**\Map – Implementation using the AVL Binary Search Tree**

NOTE: An error was found on page 3. Part in red is the correction. However, I have not been able to verify if the code is already considering this. It is a good exercise to find that out.

1. **Introduction**

In today’s activity, we work with an implementation of the *Map* ADT using a *balanced binary search tree* data structure. In particular, we shall use binary search trees satisfying the AVL property as studied in lectures. Once again we will use the *Entry* and *Map* interfaces as specified in lectures and on previous lab sessions.

1. **The AVL Binary Tree**

Let’s first recall that is the height of a node in a binary tree. The height of a node v is: *max(height(left), height(right))+1*, where *height(null)* is defined as -1. A binary tree is also an AVL tree if every node v in the tree satisfies *Math.abs( height(left child of v) – height(right child of v) ) <= 1*.

It can be mathematically proven that the height of any binary AVL tree having *n* nodes is *O(lg n)*, where *lg* means “logarithm base 2”. Therefore, an AVL tree has asymptotically minimum height possible, and hence the length of the simple path from the root to any of the leaves in the tree is *O(lg n)*. This further implies that if we use an AVL tree to implement the Map operations (following the idea of the ones that we have studied for a bst), then the worst-case execution time that each operation has is *O(lg n),* which is optimal for the case of the binary search tree implementation. Of course, the operations *remove* and *put* may alter the height of some of its nodes, and therefore, in order to continue complying with the AVL property, the tree may require some restructuring. Fortunately, the creators of the AVL tree came with nice algorithms to detect and correct unbalancing during the *remove* and *put* operations, and those algorithms do not alter the asymptotic behavior of these two Map operations. For a complete formal theory about the AVL trees and restructuring operations, refer to the textbook for proper places in the literature or to textbooks in the area of analysis of algorithms. At this moment, we will just use the fundamental results that have been proven elsewhere. Some of them are easy to prove formally, while others may be kind of harder. But these are topics proper for another course. We discuss these restructuring strategies on the next section.

1. **An Implementation of AVL Binary Search Tree**

Our final goal is to implement the Map using the AVL binary search tree. In this case, we will follow a slightly different approach from the one followed in the textbook. However, the authors’ approach will be more suitable for code reusability when studying other types of balanced binary search trees. As we did for the previous lab (the lab about binary search trees and an implementation of the Map using such data structure), we will first implement a binary search tree that is also AVL. For this, we will eventually have a class named: LinkedAVL\_BST. Then the implementation of the Map will use one such binary search tree (as in the previous lab, but now, and AVL tree) to store its entries.

Recall the operation to *add a new element* into the tree (operation addElement(E e)). The new element is added as the element of a new node that will be an external node of the tree. A path from the root to the location where such node must be inserted is traversed downward while guided by the results of comparing the new value and the value on each node found in that path. As for the case of the operation to *remove an element* from the tree (operation removeElement(E e)), the way in which the node that is physically removed from the tree is also determined by searching downward in the tree in a similar manner. Remember that the node that is finally removed from the tree is not necessarily the node containing the value to remove from the collection. It can be a node in the right subtree of the node containing the value to remove.

Let’s define what we call a *reference position* for each of these two operations. In the case of the operation to add a new element to the tree, the reference position is the *new node* that has been added (initially as a new leaf in the tree). In the case of the operation to remove an element from the tree, that reference position is *the parent node of the one finally deleted*.

From the theoretical results that are known about the AVL binary search tree, we can summarize the following. Let’s assume that T is such a tree before one of the operations, *put* or *remove*, is applied. Also assume that the operation causes the tree to violate the AVL property. Then the following result is true for both operations: If a position in T has lost the AVL balancing property, then that node must be in the path between that reference position to the root of T. Furthermore, appropriate restructuring of nodes in that path is enough to rebalance the tree and make it be an AVL tree once again. For the sake of the discussion that follows, let’s refer to that path as the *critical path*.

Based on the previous results, all what is required is to somehow traverse that critical path from bottom to root and on the way up determine nodes in the path that are unbalanced, and, if so, rebalance them. The process also includes recompute the height of nodes along that path. We refer to the current node being explored in that path as *z*. Since z is the current node, all its descendants in the critical path must have been explored, balanced if needed to, and their corresponding heights have been properly updated. Now, if z is unbalanced, then the following can be shown regarding this node: One of its subtrees has height at least 1. It means that the height of node *z* is at least 2. Therefore we can find two descendant nodes of *z*, named node *y* and node *x*, such that *y* is the *highest child* *of* *z* and *x* is the *highest child of y*. In the case that the two children of y have equal height (this may happen only on the remove operation -- think why is this true...), then the child of y to select would be the one the is the same child of y as y is of z. (That is to say that if the two children of z have the same height, then the left child of y becomes x if y is the left child of z, and if y is the righ child of z, then x becomes the right child of y). These three nodes become the pivot nodes used in the restructuring of the subtree whose root is z. The following figure illustrates the four possible relation between nodes x, y and z.

T1

T1

T3

T2

T2

T4

T4

T3

T1

T1

T3

T2

T2

T4

T4

T3

x

y

z

z

y

x

z

y

x

z

x

y

Figure 1. The four possible cases defined by the relationship between the three pivotal nodes involved on a rebalancing operation. The triangles represent subtrees, possibly empty; notice that they are numbered as in the order in which they would be traversed on an inorder traversal of the whole tree.

Case 1 Case 2 Case 3 Case 4

Before describing the restructuring process using those three pivot nodes, we should first state the following results, which can also be mathematically proven on the tree that was AVL and bst before the operation. In the case of the *insert* operation (one more node has been added as a descendant of *x,* which could possibly be *x* itself), once this restructuring is done for the first time (for the lowest unbalanced *z* in the critical path), the tree becomes an AVL tree again; therefore, no more rebalancing is required. In the case of the *remove* operation (one node has been removed from the path and it was part of the subtree of node *z* that is (or was if it was the only node there) rooted at the sibling of node *y*, the other child of *z*) once the restructuring is done for those three nodes, the process needs to continue upward along the critical path, since one or more nodes may still need rebalancing. But in this case, that rebalancing must continue from bottom to top from the node that wast the parent node of *z* before restructuring. The process is guaranteed to yield an AVL tree once again when the root node is reached and certified as balanced, or rebalanced when needed.

These results are the driving force for the rebalancing algorithms. As illustrated in Figure 1, there are four possible cases in terms of the relationship of nodes *y* and *x* with their respective parents (*z* and *y*, respectively). These are:

Case 1: *y* is left child of *z* and *x* is left child of *y*

Case 2: *y* is left child of *z* and *x* is right child of *y*

Case 3: *y* is right child of *z* and *x* is left child of *y*

Case 4: *y* is right child of *z* and *x* is right child of *y*

Each of these four cases requires different (but similar and simetric) actions. The following figure illustrates the actions needed on the first two cases:

Figure 2. Illustration of single and double rotations to rebalance case 1 and case 2, respectively.

The figure shows the possible outset of the first two cases. It suggests what should be done with the root nodes of the four subtrees depicted as triangles, if anything at all, in order to rebalance the whole subtree originally rooted at *z*. Notice that, in Case 1, node *y* becomes the new root, whereas for the case 2, and *x* becomes the new root. The abbreviations RR and RL mean “rotate right” and “rotate left”, respectively. The required actions must be clear from the figure. It can be also mathematically proven that, once the restructuring of the subtree is performed as suggested, the subtree becomes AVL and continues being bst. The operation to perform for the Case 3, is *RL(z, y).* For Case 4, we need to perform the sequence: *RR(y, x)* and *RL(z, x).* You should be able to derive this by first drawing the analogous subtree frame for these to cases.

T1

T3

T2

T4

Figure 3. The final rebalanced subtree on each of the four cases.

Case 1 Case 2 Case 3 Case 4

x

y

z

T1

T3

T2

T4

z

y

x

T1

T3

T2

T4

z

x

y

T1

T3

T2

T4

y

x

z

Complicated to do this? Not at all. We really need to define two operations: one for *RR* and another for *RL*. Furthermore, each of these two operations only needs to properly deal with a parent-child pair of nodes. The general algorithm for this is as follows. Let’s refer to it as *rebalance(z)*: rebalances the subtree rooted at *z* and then returns the root of the restructured tree. It assumes that the subtrees rooted at notes y and *x* (to be determined from z as described above) are AVL trees. The only unbalanced node is its root *(z)*. The algorithm is as follows:

*rebalance(z)*

1. determine nodes *y* and *x* as previously described
2. determine the case that describes the relationship of *y* and *x* with their respective parents. For each case perform the following actions:

Case 1: *RR(z, y)*, and *y* becomes the new root

Case 2: *LR(y, x)*, *RR(z, x)*, and *x* becomes the new root

Case 3: *RR(y, x)*, *LR(z, x)*, and *x* becomes the new root

Case 4: *LR(z, y)*, and *y* becomes the new root

1. return the new root of the subtree

We can blindly simplify the previous process even more by proceeding as follows: Instead of rotations, we can sort nodes *x,* *y* and *z* as per their content. Once sorted, let *a* be the first, *b* be the second, and *c* be the third. Or more specific, for each of the above four cases:

Case 1: *a = x, b = y, c = z*

Case 2: *a = y, b = x, c = z*

Case 3: *a = z, b = x, c = y*

Case 4: *a = z, b = y, c = x*

Figure 4 illustrates this further.

T1

T1

T3

T2

T2

T4

T4

T3

T1

T1

T3

T2

T2

T4

T4

T3

**a**=x

**b**=y

**c**=z

**c**=z

**a**=y

**b**=x

**a**=z

**c**=y

**b**=x

**a**=z

**c**=x

**b**=y

Figure 4. Identification of **a**, **b**, and **c**, on each of the four possible cases previously discussed.

Case 1 Case 2 Case 3 Case 4

Then, to restructure the subtree originally rooted at *z*, we just need to accomplish what is depicted in Figure 5.

T1

T3

T2

T4

a

b

c

Figure 5. Final rebalanced subtree after *a,* *b*, and *c* are determined. The root of the balanced subtree is now *b*.

From the previous discussion and illustrations, we can see that once a is identified, in all cases, its left subtree does not change throughout the rebalancing process. Same thing happens with the right subtree of node c. Hence, in terms of the four subtrees T1, T2, T3, and T4, we only need to identify T2 and T3, and, once identified, make them become the right subtree of a and the left subtree of c, respectively. The above can be achieved by the following method:

|  |
| --- |
| **private void restructure(Node<E> a, Node<E> b, Node<E> c) {**  **//** the following identifies root nodes for T2 and T3  **Node<E> t2Root = (a.getParent() == b ? a.getRight() : b.getLeft());**  **Node<E> t3Root = (c.getParent() == b ? c.getLeft() : b.getRight());**    **attachRight(a, t2Root); //** attach T2 as right subtree of node a (method of class LinkedAVL\_BST)  **attachLeft(c, t3Root); //** attach T3 as left subtree of node c (method of class LinkedAVL\_BST)  **attachSubtrees(b, a, c); //** attach subtrees rooted at a and c as new left and right subtrees  **//** of node b (method of class LinkedAVL\_BST)  **}** |

NOTE: The above method can be improved, and it is left as an exercise for you to practice. For example, notice that it may reasign what is already properly assigned. This is not an error, but it is enefficient.

1. **Some Implementation Details**

Let’s consider a formal implementation of this strategy to finally implement the Map ADT based on an AVL binary search tree. As we did on the previous lab, we implement the AVL binary search tree as a separate class; then, the Map implementation will have and instance field that references one such tree, and operations of the map will properly invoke operations on the tree.

The algorithms to search on an AVL binary search tree are the same as were originally studied for the binary search tree. The operations that need further actions are: adding a new element to the tree and removing an element from the tree. In each of those, the process needs to traverse the critical path, from bottom to top (the root) take the following actions:

* Determine unbalanced nodes and balance them if necessary.
* Recompute height of each node in the path or surrounding nodes that get involved in any rebalancing process.

As was previously discussed, for each of the two operations, the process identifies a *reference node* that becomes the lowest node in the critical path from where the process begins. In the case when the new element is added to an empty tree, its new node becomes the root and the only node in the tree; hence, for that case, there is no need of further actions for the tree to be an AVL tree. For the case when a node is removed from the tree, if the parent node of the finally deleted node is null, then the removed node was the root of the tree. Then, its only subtree, if any, becomes the new tree and no further action is required. In any other case in both operations, the analysis and possibly rebalancing of nodes in the critical path is required. Also, as nodes are traversed along that path, once balanced, their respective heights are recomputed. All this is accomplished by the method upwardBalance(...). At the end, the method returns the root of the final AVL tree; which ends up being the same root that the tree has when the process begins or one of its original successors if rebalancing at level 0 is also required. This is implemented inside class LinkedAVL\_BST<E>, which is a subclass of LinkedBinarytree<E>. Also, notice that class Node<E> (inside class LinkedBinaryTree<E> has been modified inorder to include a field to store the height of each node in the tree.

1. **Exercise for this Lab Session about Map**

Import to Eclipse the partial project included in the zip file. Just study the code as it is to see how the previously described processes have been implemented in class LinkedAVL\_BST<E>. This is supposed to be a working version, if you find any error, please let us know. However, as said on the statement in red at the beginning of this document, the correction stated on page 3 (statements in red) as to how the node x is selected may not be implemented properly. A good exercise would be for you to find out if that is the case; if so, you can make the correction as per the clarification given.

Several testers have been included. But you may want add your own in order to understand better and better certify that the implementation is correct... The display method for the AVL binary tree will also display, for each position (node), the height of that node in the tree and its balance factor (0, 1 or -1).

Most of the methods have been properly commented, so, you can follow those comments to understand better the code given and match it with the above discussions. However, some methods can be improved by providing more efficient implementations. For example, the previously shown method (**restructure**) can be improved. If you look at the figures, the process does not always need to reassigng t2Root and t3Root. See how you can improve it...

1. FIN …