COL106 : 2021-22 (Semester I)
Project: Module 4

Saurabh Narendra Inge Raj Kamal Venkata Koppula

September 25, 2021

Notations

For two strings a, b, we denote $a.\texttt{concat}(b)$ by $a + b$ (that is, if $a =$ "Hello", and $b =$ "World", then $a + b =$ "HelloWorld"). For any natural number n, $[n]$ denotes the set $\{1, 2, \ldots, n\}$.

Instructions The lab-submission problem (marked ♠) is to be submitted via Moodle by 11:59PM on the day that you have your lab.

1 Introduction

In the last module, we saw how Merkle trees can be used for creating authenticated sets. However, there were a couple of restrictions:

- The number of elements had to be a power of 2.
- There were no insertions/deletions.

In this assignment, we will address these two restrictions. In particular, the server **stores the set in sorted order**,[1] and should be able to handle the following queries:

- `QueryDocument`: This is same as in the previous lab-module. However, since our Merkle tree is not a complete binary tree, we will need additional attributes to recover the i^{th} element and its sibling-coupled-path-to-root).
- `InsertDocument`: This method takes as input a document (which is a string), and it inserts it in the appropriate position (in the sorted set), and outputs the updated Merkle tree's root value.
- `DeleteDocument`: This method takes as input a document (which is a string), and if the document is present in the set, it deletes the document, and outputs the updated Merkle tree's root value.

For simplicity, we are only focusing on inserts only (that is, no deletions). Also, note that to build a Merkle tree on n documents, one can repeatedly call `InsertDocument` n times.

1.1 Additional Attributes in `TreeNode`

Every node in the tree is either a leaf-node, or an internal node. This is denoted by the attribute `isLeaf`, which is set to True for leaf nodes, and False for internal nodes. As before, every node has a `value` attribute. For leaf nodes, this is equal to the document (string) stored at the leaf. For internal nodes, this is `CRF.Fn(lv+` "#" + `rv)`, where `lv` (resp. `rv`) is the `value` attribute of the left child (resp. right child).

We will add a few attributes to `TreeNode`, which will be useful for the insert operation.

[1] Using some fixed ordering, e.g. lexicographic ordering

- `numberLeaves`: the number of leaves in the subtree rooted at a particular node. This will be useful for recovering the i^{th} document.
- `maxleafval`: the maximum value of all leaves in the subtree rooted at a particular node. This will be useful for inserting a new document.
- `minleafval`: the minimum value of all leaves in the subtree rooted at a particular node. This will be useful for inserting a new document.
- `balanceFactor`: we will describe this attribute in Section 1.3. It is used to ensure that the Merkle tree is *balanced*.

1.2 Simple Insert Operation

For simplicity, we will assume that all elements in the authenticated set are distinct, and every new element inserted is also distinct (and not present in the authenticated set). Let z be the document (string) to be inserted. There are two cases:

1. z is less (in lexicographic ordering) than all elements in the set.
2. The set contains strings that are less than z.

Let us consider the second case (the other case will be similarly handled). Let z_1 be the largest value in the auth. set that is smaller than z. In order to insert z, we will replace the leaf node corresponding to z_1 with an internal node whose left child is z_1 and right child is z. The `value` of this internal node will be set to `CRF.Fn`$(z_1 + \text{“\#”} + z)$. After this, the values at all the ancestor nodes of z are updated one by one. Finally, the root value is updated, and the updated root value is the output of the simple insertion operation.

Consider the Merkle tree given in Fig. 1.

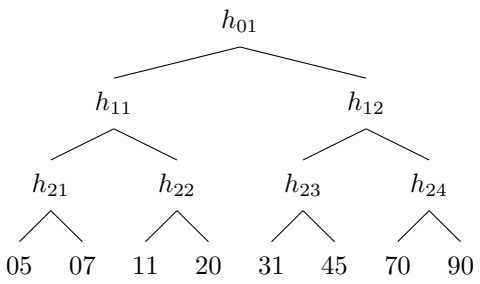


Figure 1: Given Merkle tree

After inserting 25, we get the Merkle tree as shown in Fig. 2. Note that the internal node values h_{01}, h_{11}, and h_{22} get updated to h'_{01}, h'_{11}, and h'_{22} respectively.

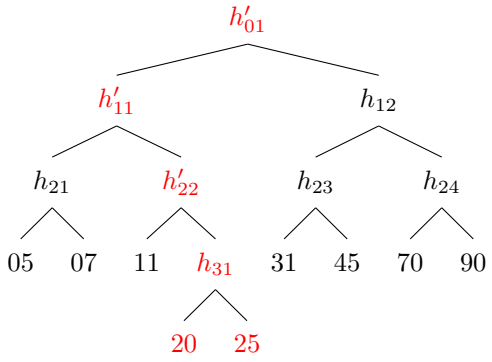


Figure 2: Merkle tree after insertion of 25

Again after inserting 27, we get the Merkle tree as shown in Fig. 3. Note that the internal node values h'_{01}, h'_{11}, h'_{22} and h_{31} get updated to h''_{01}, h''_{11}, h''_{22} and h'_{31} respectively.

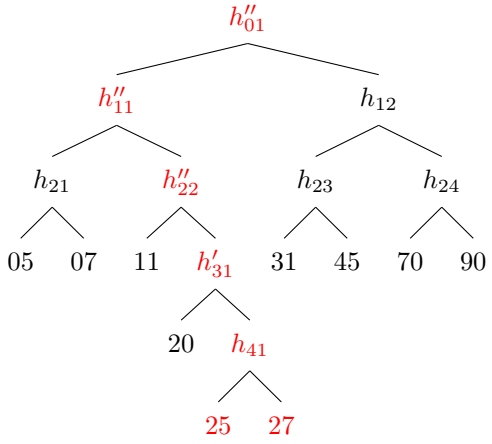


Figure 3: Merkle tree after insertion of 27

1.3 Re-balancing the Tree: Single Rotation and Double Rotation

The tree may become unbalanced after insertion/deletion of elements. A node in a Merkle tree is balanced if height of its left subtree and right subtree differ by -1, 0 or 1. A node which is not balanced is called unbalanced. For instance, in Figure 2, every node is balanced. However, after adding 27, the Merkle tree in Figure 3 has unbalanced nodes (the nodes corresponding to h''_{22}, h''_{11} and h''_{01} are unbalanced).

To make this merkle tree of Fig. 3 balanced, we 'rotate' the subtree given in Fig. 4a to obtain the subtree given in Fig. 4b.

(a) original subtree

(b) rotated subtree

Figure 4: subtree rotation

The final balanced merkle tree is given in Fig. 5. Note that the internal node values h''_{01} and h''_{11} get updated to h'''_{01} and h'''_{11} respectively.

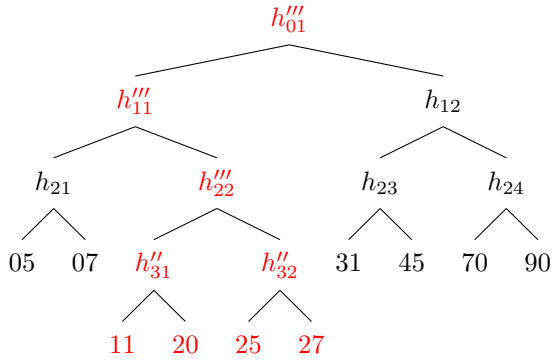
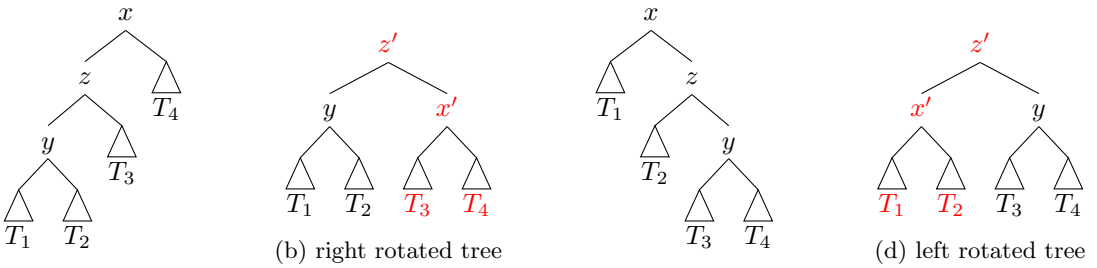


Figure 5: Final balanced Merkle tree after rotation

1.3.1 Single Rotation

Strictly speaking, unlike AVL trees, this is not a rotation; however the balancing operation is very similar to the 'single rotation' operation used for balancing AVL trees, hence we will call it rotation.

Consider the Fig. 6. We assume that the subtrees T_1, T_2, T_3, and T_4 have same height h. The tree given in Fig. 6a is not balanced. To make this tree balanced, we do right rotation as shown in Fig. 6b. Note that the internal node values x and z get updated to x' and z' respectively. Similarly, the tree given in Fig. 6c is not balanced. To make this tree balanced, we do left rotation as shown in Fig. 6d. Note that the internal node values x and z get updated to x' and z' respectively.

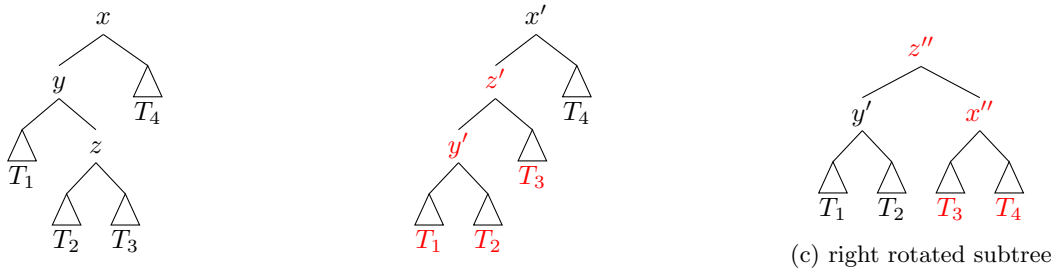


(a) original tree (b) right rotated tree (c) original tree (d) left rotated tree

Figure 6: single rotation

1.3.2 Double Rotation

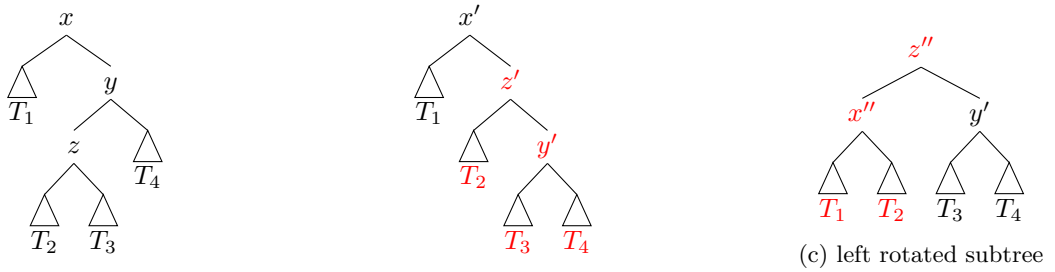
Consider the tree given in Fig. 7a. We assume that the subtrees T_1, T_2, T_3, and T_4 have same height h. Then the tree given in Fig. 7a is not balanced. To make this tree balanced, we first do left rotation as shown in Fig. 7b. Note that the internal node values x, y and z get updated to x', y' and z' respectively. Then we do right rotation as shown in Fig. 7c. Note that the internal node values x' and z' get updated to x'' and z'' respectively. The tree given in Fig. 7c is balanced.



(a) original tree (b) left rotated subtree (c) right rotated subtree

Figure 7: left-right double rotation

Similarly, consider the tree given in Fig. 8a. We assume that the subtrees T_1, T_2, T_3, and T_4 have same height h. Then the tree given in Fig. 8a is not balanced. To make this tree balanced, we first do right rotation as shown in Fig. 8b. Note that the internal node values x, y and z get updated to x', y' and z' respectively. Then we do left rotation as shown in Fig. 8c. Note that the internal node values x' and z' get updated to x'' and z'' respectively. The tree given in Fig. 8c is balanced.



(a) original tree (b) right rotated subtree (c) left rotated subtree

Figure 8: right-left double rotation

2 Assignment Questions

You must solve all the problems below, preferably before your lab session next week. The lab test questions will build upon these assignment questions, and therefore it is strongly recommended that you try these questions beforehand.
You are given the following classes:

- `public class Pair<A,B>` : Objects of this class would be used to store each element in the sequence *Sibling-Coupled Path to Root*. The attributes of the class are:
  - `public A First`: the first element stored in the object.
  - `public B Second`: the second element stored in the object.

- `public class TreeNode`: This class has the following attributes:
  - `public TreeNode parent`: the parent node
  - `public TreeNode left`: the left child node
  - `public TreeNode right`: the right child node
  - `public String val`: the value contained in this node
  - `public boolean isLeaf`: indicates whether the node is a leaf node or internal node
  - `public int numberLeaves`: the number of leaves in the sub-tree corresponding to this `TreeNode`.
  - `public String maxleafval`: the maximum value among all leaves of the subtree rooted at this `TreeNode`.
  - `public String minleafval`: the minimum value among all leaves of the subtree rooted at this `TreeNode`.
  - `public int balanceFactor`: (height of left subtree) - (height of right subtree)

  This class has the following methods:

  To be implemented:
  - `public TreeNode SingleRightRotation()`: this method takes no input, performs right rotation, and outputs the new root of the subtree. For instance, if `nd` is the node corresponding to the root in Figure 6a, then `nd.SingleRightRotation()` should output a `TreeNode` whose left-left-subtree corresponds to T_1. The balance factor of this output node must be 0 for this example. You must also update the other attributes of this output node and other nodes in this subtree.
  - `public TreeNode SingleLeftRotation()`: this method takes no input, performs left rotation, and outputs the new root of the subtree. For instance, if `nd` is the node corresponding to the root in Figure 6c, then `nd.SingleLeftRotation()` should output a `TreeNode` whose left-left-subtree corresponds to T_1. The balance factor of this output node must be 0 for this example. You must also update the other attributes of this output node and other nodes in this subtree.
  - `public TreeNode DoubleLeftRightRotation()`: this method takes no input, performs left-right rotation, and outputs the new root of the subtree. For instance, if `nd` is the node corresponding to the root in Figure 7a, then `nd.DoubleLeftRightRotation()` should output a `TreeNode` whose left-left-subtree corresponds to T_1. The balance factor of this output node must be 0 for this example. You must also update the other attributes of this output node and other nodes in this subtree.

- `public TreeNode DoubleRightLeftRotation()`: this method takes no input, performs right-left rotation, and outputs the new root of the subtree. For instance, if `nd` is the node corresponding to the root in Figure 8a, then `nd.DoubleRightLeftRotation()` should output a `TreeNode` whose left-left-subtree corresponds to T_1. The balance factor of this output node must be 0 for this example. You must also update the other attributes of this output node and other nodes in this subtree.

- `public class MerkleTree`: This class has the following attributes:
  - `public TreeNode rootnode`: pointer to the root node of the Merkle Tree
  - `public int numdocs`: number of documents (n) stored in this Merkle tree.

  This class has the following methods:

  To be implemented:
  - `public String InsertDocument(String document)`: This method takes as input a document, and inserts it into the authenticated set.
    * If there are no existing documents in the auth. set, then the method creates a new `TreeNode`, and the root is also a leaf-node in this special case. The string returned will be the value of the root node, which is equal to the string inserted.
    * If there is exactly one document in the auth. set, then the resulting Merkle tree (after insertion) will have three nodes: the root node, with two children which are leaf nodes.
    * If there are at least two documents in the auth. set, then insertion works as follows:
      1. First, find the leaf node `nd` whose value is the largest value less than the document to be inserted. (If there is no such leaf node, then that means the inserted document is less than all documents in the set – act accordingly.)
      2. Replace `nd` with an internal node whose left child is `nd` and right child contains the document to be inserted.
      3. Update the relevant attributes of all the ancestor nodes.
      4. Starting with the newly created node, if there are unbalanced nodes on this path to the root, then balance those (using the four rotation methods). Note that after the rotation, you must also update the CRF values of the ancestor nodes.
    * Finally, output the value at the root.

    You must not store the authenticated set as an array/list. This is because we want efficient (logarithmic time) updates in the worst case.
  - `public String DeleteDocument(String document)`: This method deletes the input document from the authenticated set (if present), and outputs the new root value. If the document is not present, it throws an exception.

2.1 Exercises

1. ♠ **Inserting documents into Merkle tree.**

   Implement the method `InsertDocument`(as explained above).

2. (Optional) **Implement** `DeleteDocument`.

3 Instructions

- Do not change the accessibility, names or signatures of the attributes and methods in the driver code. You may add your own attributes and methods to any of the above classes as and when required.
- The default constructor is used to instantiate objects of all the above classes. It is your responsibility to ensure appropriate initialization of the attributes of a newly created object.
- You are **not allowed** to use any other preexisting classes/data structures.
- **Submission instructions for lab-submission problem:** You must submit `TreeNode.java` and `MerkleTree.java` on Moodle. Please follow the instructions on Moodle regarding file name.

Acknowledgements

This is Version 0 of the document. If you find any errors, please send an email to `kvenkata@iitd.ac.in`, `csz188013@cse.iitd.ac.in` and `mcs202469@cse.iitd.ac.in` (and participate in the 'error-finding competition').