Homework 5

Out: 10.26.23
Due: 11.6.23

1. [Selection, 10 points]
   Let A be a list of n (not necessarily distinct) integers. Describe an O(n) algorithm to test whether any item occurs more than ⌈n/2⌉ times in A.

2. [Hashing, 24 points]
   Insert the following keys into an initially empty hash table of size 10:
   4371, 1323, 6173, 4199, 4344, 9679, 1989.
   Show the resulting hash table, with the hash function h(x) = x mod 10.
   a. Using open hashing (i.e. chaining).
   b. Using closed hashing (open addressing) with linear probing.
   c. Using closed hashing (open addressing) with quadratic probing, C0=C1=0, C2=1.
   d. Using double hashing with the secondary hash function h2(x) = 7 – (x mod 7).

3. [Binary Search Tree, 16 points]
   a. Draw the binary search tree which results from inserting the following numbers, one after the other, from left to right: 3 12 2 8 1 1.5 6 0.5 7 20.
   b. Show the resulting binary search tree after deleting 8. Explain.
   c. Show the resulting binary search tree after deleting 8 and 12. Explain.

4. ~~[Merkle Tree, 50 points]~~
   Introduction Merkle trees are used to validate data integrity by detecting data tempering. To do so, Merkle trees utilize one-way collision resistant hash functions. Leaf nodes contain hashes of data blocks, and internal nodes contain hashes of the concatenated hashes of their children. The root node serves as the fingerprint for the entire tree and should be stored in a secure location.

   In our implementation, the input data is stored as a vector of integers, and a 4-ary Merkle tree is built on top of the data. Each leaf contains the hash of an individual data point, and each internal node contains the hash of the concatenated hashes of its 4 child nodes. The root of the tree will thus contain the signature hash of all the data, and a change at any data point will result in a substantial change to the root. To ensure that the root can be used for verification, the root is saved in a secure location accessible only to authorized parties, while the data protected by the Merkle tree (as well as the rest of the tree) is volatile and may be tampered with.
   Every read from the data vector is verified by recomputing the root and comparing it to the saved root. A mismatch is an indication that the data has been altered and cannot be trusted.

   The Hash function The data vector is a vector of integers, and the hash stored at each node is a string. We utilize FNV1a hashing as it gives the same results on all platforms. It is used as follows:

//call hash function on a string input, store result in "hashed_message" variable
string hashed_message = fnv1a(to_string(value_to_be_hashed));

For internal nodes, value_to_be_hashed is the concatenation of the hashes of child nodes, left to right. This is already implemented in the *concatenateHash* method.

Task description You are provided with a partial implementation of a Merkle tree in *merkle_tree.h* and *merkle_tree.cpp*, as well as sample *main.cpp* and *merkle_tree_input.txt* files. The input file contains one integer per line. The header file contains a *Node* class and a *MerkleTree* class. The nodes contain a key field, which stores the concatenated hash of of the children nodes, and a vector of pointers to the children of the node. The Merkle Tree class contains declarations of all required methods that you need to implement. You may add members to the classes but do not modify existing class members.

Implement a Merkle tree that supports building a tree either from a given data vector of integers or by inserting data points one at a time. Changes to leaf nodes should cause all the keys of all relevant parent nodes to be updated as well. For example, if your data vector has 7 values, and an 8th value is added, the new data point should become the rightmost child of the right parent node. The hash of the right parent should then be updated, as well as the root value.

The Merkle tree class contains the following methods and members:

- ~~data – Vector of ints to store the data that the tree is built on.~~
- ~~root – Pointer to the root node (private member, "saved in secure location").~~
- ~~merkleTree() – Constructor of an empty Merkle tree.~~
- ~~merkleTree(vector<int>& data) – Constructor of a tree from a data vector.~~
- ~~concatenateHash – Provided method that utilizes the provided hash function, concatenates child hashes from left to right and returns the hash of the concatenation.~~
- ~~printTree – Method to visualize the Merkle tree and its content (not provided).~~
- ~~verify – Saves current root, then rebuild the tree over the data vector, compares original root hash to new hash, and returns True if they match and False otherwise.~~
- ~~overwrite – Overrides a given value with new input value and rehashes the tree. The given value must be a member of the vector. Starts by searching for the given value in the data vector.~~
- ~~insert – Inserts new value to the end of the data vector and rehashes the tree.~~
- ~~printRoot – Prints the root hash.~~

The main function receives an input file as a command-line argument. For example, suppose that you compile your program to the executable *Problem4*, your program should be run as follows with the provided input file:

> Problem4 merkle_tree_input.txt

The provided *main.cpp* should give the following output:

```
Inputted data: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Root value: 12837331367622398083
7 overwritten to 100
New root value: 7802256578728336894
Inserting: 50
New root value: 9742762515618019747
Inserting 1, 2, 3, 4, 5, 6
Level 0: 17944618359440641973
 Level 1: 7802256578728336894
 Level 1: 8170833380014453750
  Level 2: 4916309629727857311
  Level 2: 10469149479656363843
  Level 2: 4706293407091169726
  Level 2: 15007273788742128653
  Level 2: 4453533732341374910
  Level 2: 15759420036096308178
   Level 3: 12638134423997487868
   Level 3: 12638137722532372501
   Level 3: 12638136623020744290
   Level 3: 12638131125462603235
   Level 3: 12638130025950975024
   Level 3: 12638133324485859657
   Level 3: 5001448700306559868
   Level 3: 12638144319602141767
   Level 3: 12638143220090513556
   Level 3: 574369514284255396
   Level 3: 574370613795883607
   Level 3: 574371713307511818
   Level 3: 574372812819140029
   Level 3: 574365116237742552
   Level 3: 574366215749370763
   Level 3: 574367315260998974
   Level 3: 570543213818838016
   Level 3: 12638134423997487868
   Level 3: 12638137722532372501
   Level 3: 12638136623020744290
   Level 3: 12638131125462603235
   Level 3: 12638130025950975024
   Level 3: 12638133324485859657
Verifying tree...
Is data secure: True
Malicious data attack occurs!
Verifying tree...
Is data secure: False
Creating a new tree from inserts
Inserting: 20, 10, 1 to new tree
New tree final root value: 9163769391530059507
Verifying tree...
Is data secure: True
Level 0: 9163769391530059507
 Level 1: 577309608377523935
```

```
 Level 1: 574369514284255396
 Level 1: 12638134423997487868
Verifying tree...
Is data secure: True
Malicious data attack occurs!
Verifying tree...
Is data secure: False
Creating a new tree from inserts
Inserting: 20, 10, 1 to new tree
New tree final root value: 9163769391530059507
Verifying tree...
Is data secure: True
Level 0: 9163769391530059507
 Level 1: 577309608377523935
 Level 1: 574369514284255396
 Level 1: 12638134423997487868
```

Deliverables Submit your modified *merkle_tree.cpp* and *merkle_tree.h* files only. Your code must compile and run on the lab computers with the provided *main.cpp* file (or a variation of it). In a comment at the top of your *merkle_tree.cpp* file, explain how you implemented your *insert* function.