# excersize2_1

February 14, 2018

## 1 PH 360 - Excersize 2

### 1.1 By: Shifra Abittan & Guy Bar Yosef

### 1.2 Part 1: Merkle Tree

```
In [2]: import hashlib as sh
        import numpy as np
```

Our merkle tree is constructed out of merkle nodes and hash pointers. These two classes are defined below:

```
In [3]: class hashPointer:
            ''' A struct that holds two attributes:
                    pointer : A pointer to any data structure
                    myhash  : The hash of the data strcture being pointer to.

                    NOTE: The hash pointer does not hash the data itself,
                    rather the hash has to be inputted during the object instantiation.
            '''
            def __init__(self, data, inputhash):
                self.pointer = data;
                self.myhash = inputhash # simply reads in the hash provided as a paramater

            def __str__(self):          # prints out the value being pointer to.
                return "%s" % self.pointer
```

The Merkle Node is the building block of the merkle tree. Each block includes two hash pointers, either to the leafs at the bottom of the tree or to other merkle nodes, as well as the hash of the concatination of the hashes of the two objects being pointed to. The reason to include the hash of the current merkle node as a variable is to simplify hashing the two hashpointers together without including the other object's metadeta in said hash. If there is only one input to the Node, instead of two, as would happen in an odd-sized tree, the rightmost chain of merkle nodes will all have a 'None' as their right hashPointer and the hash of the merkleNode will be the hash of the concatenation of the left hashPointer with itself.

```
In [4]: class merkleNode:
            ''' The blocks of the merkle tree.

                Attributes:
                    - left   : left hash pointer.
                    - right  : right hash pointer.
                    - myhash : the hash of the concatination of the two hash pointers.
            '''
            def __init__(self, left, right = None):
                self.left = hashPointer(left, left.myhash)
                self.right = right
                # If amount of leaf nodes is odd, the last merkleNode (of the last data point) will have only one input.
                # In that case, hash the single input twice to obtain the node's hash.
                if self.right == None:
                    self.myhash = sh.sha256(left.myhash.encode() + left.myhash.encode() ).hexdigest()
                else:
                    self.right = hashPointer(right, right.myhash)
                    self.myhash = sh.sha256(left.myhash.encode() + right.myhash.encode() ).hexdigest()


            def printNode(self):
                '''Goes through the nodes of the tree recursevly and prints out the leafs of the tree, left to right.
                '''
                if not hasattr(self.left.pointer, "left"): # Reached the lowest Node in tree, which has a left leaf
                    print (self.left)
                    if self.right != None:  # Will print out the right leaf if it exsists
                        print(self.right)
                else:                         # we are not yet at the bottom of the tree
                    self.left.pointer.printNode()
                    if self.right != None:
                        self.right.pointer.printNode()
```

Merkle tree implementation. Methods included: - printTree(void) : Prints the tree leafs, from left to right.

- verifyMembership(data, branch) : Takes in the data that one wants to confirm is in the tree as well as the relevant audit proof hashes (given the variable name 'branch'). This method will compute the hashes to reconstruct the merkle root and return 'True' if this is the resulting merkle root is correct and 'False' otherwise.
- dataBranch(data) : Completes two jobs. Takes in a data point as well as its location in the tree.
  - First, it will return a 'False' if the data point is not in the location specified, so it checks whether the data point is in the tree or not.
  - Secondly, if the data is in the tree, the method will return said data's audit proof hashes, allowing the 'verifyMembership' method to take in the data point and the audit proof hashes to recreate the merkle root.

```
In [5]: class MerkleTree:
            ''' Merkle tree.

                Attributes:
                    - list      : Ordered list of leafs.
                    - tableroot : Hash pointer to the Merkle Root.
            '''
            def __init__(self, sortedList):
                self.size = len(sortedList)

                intermediate = sortedList
                temp =[]
                for i in intermediate:
                    temp.append( hashPointer(i, sh.sha256(i.encode()).hexdigest() ) )
                intermediate = temp

                while (len(intermediate) > 1):
                    temp = []
                    for i in range(len(intermediate)):
                        if (i % 2 == 0 ):
                            if i+1 >= len(intermediate):
                                temp.append( merkleNode(intermediate[i]) )
                            else:
                                temp.append( merkleNode(intermediate[i], intermediate[i+1]) )
                    intermediate = temp
                self.tableroot = hashPointer(intermediate[0], sh.sha256(intermediate[0].myhash.encode() ).hexdigest() )



            def printTree(self):
                ''' Prints out the tree leafs, left to right.'''
                self.tableroot.pointer.printNode()



            def verifyMembership (self, datapoint, branch):
                ''' Prove membership of inputed 'datapoint' when hashed with the hashes provided in 'branch' list.

                    Branch list is indexed starting at the leafs and continuing up the tree.
                    The Branch is a list of lists, with each inner list consisting of the side that its hash is supposed
                    to be in the current merkle node ("right" OR "left") and the hash itself.
                '''
                attempt = sh.sha256( datapoint.encode() ).hexdigest()
                for i in range(0, len(branch)):
                    if branch[i][0] == "right":
                        attempt = sh.sha256( attempt.encode() + branch[i][1].encode() ).hexdigest()
                    else:
                        attempt = sh.sha256( branch[i][1].encode() + attempt.encode() ).hexdigest()

                if self.tableroot.pointer.myhash == attempt:
                    return True
                else:
                    return False



            def dataBranch (self, datapoint, location):
                ''' Takes in a data point that may or may not be in the tree.
                    If data is in the tree, returns its audit proof hashes. Otherwise returns a False.
                    The audit proof hashes list is indexed starting at the leafs of the tree.
                '''
                found = False
                it = self.tableroot.pointer
                branch = []
                first = 0
                last = self.size - 1
```

```
        if self.size % 2 == 1: # size of tree is odd
            # if the looked for leaf is in the rightmost leaf, it is in right half of tree.
            if location == self.size -1:
                # first go into the right half of tree and then continue on left hashPointers
                # as all the right hash pointers in the right half of tree are 'None'
                branch.insert(0, ["left", it.left.pointer.myhash])
                it = it.right.pointer
                while hasattr(it, "left"):
                    branch.insert(0, ["left", it.left.pointer.myhash])
                    it = it.left.pointer
            else:
                branch.insert(0, ["right", it.right.pointer.myhash])
                it = it.left.pointer
                last = self.size - 2

        # the same for even size of tree and odd size of tree when not looking for rightmost leaf
        while hasattr(it, "left"):
                    mid = (first + last)//2
                    if location > mid:
                        if hasattr(it.left.pointer, "myhash"):
                            branch.insert(0, ["left", it.left.pointer.myhash])
                        else:
                            branch.insert(0, ["left", it.left.myhash])
                        it = it.right.pointer
                        first = mid + 1
                    elif location <= mid:
                        if hasattr(it.right.pointer, "myhash"):
                            branch.insert(0, ["right", it.right.pointer.myhash]) ### work on last time w/ the leaf
                        else:
                            branch.insert(0, ["right", it.right.myhash])
                        it = it.left.pointer
                        last = mid

        if it.pointer == datapoint:
            return branch
        else:
            return False
```

An implementation of merge sort that we copy pasted straight out of 'Problem Solving with Algorithms and Data Structures using Python' by Brad Miller and David Ranum. - Link: http://interactivepython.org/runestone/static/pythonds/SortSearch/TheMergeSort.html

```
In [6]: def mergeSort(alist):
            if len(alist)>1:
                mid = len(alist)//2
                lefthalf = alist[:mid]
                righthalf = alist[mid:]

                mergeSort(lefthalf)
                mergeSort(righthalf)

                i=0
                j=0
                k=0
                while i < len(lefthalf) and j < len(righthalf):
                    if lefthalf[i] < righthalf[j]:
                        alist[k]=lefthalf[i]
                        i=i+1
                    else:
                        alist[k]=righthalf[j]
                        j=j+1
                    k=k+1

                while i < len(lefthalf):
                    alist[k]=lefthalf[i]
                    i=i+1
                    k=k+1

                while j < len(righthalf):
                    alist[k]=righthalf[j]
                    j=j+1
                    k=k+1
```

Some examples to present the Merkle Tree class built above:

```
In [7]: def main():

            datalist = ["input6", "input1", "input3", "input4", "input5", "input2", "input7", "input8"]
```

```
        print("original list: ")
        print( datalist)
        mergeSort(datalist)

        print("The list is now sorted using Merge Sort:")
        print( datalist)
        print("\n")


        # part (a)
        print("Description of Merkle tree and some of its methods:")
        help(MerkleTree)
        tree = MerkleTree(datalist)


        print("\n\nPrinting out the tree, left to right:")
        tree.printTree()


        print("\n\nPart C: Finding the branch needed to verify a leaf: ")
        validbranch = tree.dataBranch("input5", 4)
        try:
            for i in validbranch:
                print(i[1])
        except:
            print("Input is not in tree.")

        try:
            tree.verifyMembership("input5", validbranch)
            print("Input5 was verified using the above branch:")
        except:
            print("Can't verify with inputed branch.")


        print("\n\n")


        print("Part D: Verifying non-membership of a leaf: ")
        invalidbranch = tree.dataBranch("This won't work!", 4)
        print("Attempting to print out hashes in proof: ")
        try:
            for i in invalidbranch:
                print(i[1])
        except:
            print("Input is not in tree.")


    if __name__ == "__main__":
        main()
```

```
original list:
['input6', 'input1', 'input3', 'input4', 'input5', 'input2', 'input7', 'input8']
The list is now sorted using Merge Sort:
['input1', 'input2', 'input3', 'input4', 'input5', 'input6', 'input7', 'input8']


Description of Merkle tree and some of its methods:
Help on class MerkleTree in module __main__:

class MerkleTree(builtins.object)
 |  Merkle tree.
 |
 |  Attributes:
 |      - list      : Ordered list of leafs.
 |      - tableroot : Hash pointer to the Merkle Root.
 |
 |  Methods defined here:
 |
 |  __init__(self, sortedList)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  dataBranch(self, datapoint, location)
 |      Takes in a data point that may or may not be in the tree.
 |      If data is in the tree, returns its audit proof hashes. Otherwise returns a False.
 |      The audit proof hashes list is indexed starting at the leafs of the tree.
 |
 |  printTree(self)
 |      Prints out the tree leafs, left to right.
 |
 |  verifyMembership(self, datapoint, branch)
 |      Prove membership of inputed 'datapoint' when hashed with the hashes provided in 'branch' list.
 |
 |      Branch list is indexed starting at the leafs and continuing up the tree.
 |      The Branch is a list of lists, with each inner list consisting of the side that its hash is supposed
 |      to be in the current merkle node ("right" OR "left") and the hash itself.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)



Printing out the tree, left to right:
input1
input2
input3
input4
input5
input6
input7
input8


Part C: Finding the branch needed to verify a leaf:
a6f2925a2f6d441876c76c13e60fe448b790b96d1efc6a63d5f362a57b3d9fb3
17f1e679a8d8dc33f64242df9b386a811968d73f26b8a71fe8962b06a09de3f1
d8aa53c70d8ab86a05f97be7cd856211902a235259b17ba40286a505484b172a
Input5 was verified using the above branch:



Part D: Verifying non-membership of a leaf:
Attempting to print out hashes in proof:
Input is not in tree.
```