treeGen.py

Description: This is the main module of the project, it is the starting point of execution. This module is responsible to connect all other sub modules and interact with them to run the whole process like creating the tree, generating the bloom filter for each node of the tree and then initiating the search process to find all the elements in the given range query.

Input: SNum(Data items of the data provider), Bits(# of bits to represent the max number from the range or SNum, whichever is bigger), theRange(This is from the data user end who provides the range to be queried)

Complexity: $O(n^2)$ (It involves the worst case of dividing one element in each level and time taken to split the node into two parts)

Output: Generates the tree and searches for the element in the tree, then result is printed (queryResult)

Code:

```
import math
from functools import reduce # Used to generate permute list

# This is the main function which takes the list and breaks it into two sublists based on algorithm!
def superMain(SNum, Bits):
    sub1, sub2  = [], [] # Final two sublists!
    preSet = [] # This will contain the prefix families of individual binary element
    dPreSet = {} # This stores all (key, value) pairs of (binary, prefix family)
    binRange = []
    bitLen = Bits
    dNumToBin = {} # Helper dict to convert Number to Binary
    dBinToNum = {} # Helper dict to convert Binary to Number
    S, S1, S2, L = [], [], [], [] # Important Variables of the algorithm!
    string = '{0:0' + str(bitLen) + 'b}' # static BitLength

    # Converts the numeric to binary values!
    for n in SNum:
        bNum = string.format(n)
        S.append(bNum)
        dNumToBin[n] = bNum # Adds Binary value to Number key

    for x in SNum:
        dBinToNum[dNumToBin[x]] = x # Adds Number value to Binary key

    def createPrefix(s, preSet):
        savedS = s # Copy of the S
        temp = []  # temp list to generate the Prefix Set of a binary value

        temp.append(s)

        s = list(s)
        for x in range(1, len(s) + 1):
            s[-x] = '*'
            temp.append(''.join(s))
```

```
        dPreSet[savedS] = temp # Saves the prefix family to the binary key!
        preSet += temp

        return preSet

    for element in S:
        createPrefix(element, preSet)

    pastLCP = [] # This list keeps track of the past Longest common prefix.
    def checkLCP(S, pastLCP):
        LCP = ""
        prefixF = []

        for i in range(len(S)):
            prefixF.append(dPreSet[S[i]])

        # Finds the intersection between prefix families of the subset
        LCP = list(reduce(set.intersection, [set(item) for item in prefixF]))

        # Checking if the current LCP is unique by removing the redundant LCP by comparing it to
PastLCP
        for x in LCP:
            if x in pastLCP:
                LCP.remove(x)

        pastLCP += LCP # Adding the unique LCP to the pastLCP list for futher comparision

        ## The below block finds the Longest Common Prefix by checking for least number of * in it.
        c = 0
        countL = []

        for x in LCP:
            countL.append((x.count("*"), x))

        LCPrefix = min(countL)[1]

        LCP0 = list(LCPrefix)
        LCP1 = list(LCPrefix)

        # Replaces the first star with 0 or 1 to divide the elements of S to S1 and S2
        for x in range(len(LCP0)):
            if LCP0[x] == "*":
                LCP0[x] = "0"
                break

        for x in range(len(LCP1)):
            if LCP1[x] == "*":
                LCP1[x] = "1"
                break

        LCP0 = "".join(LCP0)
```

```
    LCP1 = "".join(LCP1)

    L0 = [] # Empty lists which will be filled by elements which have 0 in the location of the first *
    L1 = [] # Empty lists which will be filled by elements which have 1 in the location of the first *

    for i in range(len(S)):
        prefixFamily = dPreSet[S[i]] # Pulling the prefix family of individual element

        # Checking if that prefix family has LCP0 or LCP1, if present then we add to corresponding
L0/L1
        if LCP0 in prefixFamily:
            L0.append(S[i])
        else:
            L1.append(S[i])

    return L0, L1 # returns the two subsets for further computation


  pastS3LCP = [] # This list keeps track of the past Longest common prefix.
  def checkS3LCP(S, pastS3LCP):
    LCP = ""
    prefixF = []

    for i in range(len(S)):
        prefixF.append(dPreSet[S[i]])

    # Finds the intersection between prefix families of the subset
    LCP = list(reduce(set.intersection, [set(item) for item in prefixF]))

    # Checking if the current LCP is unique by removing the redundant LCP by comparing it to
PastLCP
    for x in LCP:
        if x in pastS3LCP:
            LCP.remove(x)

    pastS3LCP += LCP # Adding the unique LCP to the pastLCP list for futher comparision

    ## The below block finds the Longest Common Prefix by checking for least number of * in it.
    c = 0
    countL = []

    for x in LCP:
        countL.append((x.count("*"), x))

    LCPrefix = min(countL)[1]

    # Replaces the first star with 0 and 1
    LCP0 = list(LCPrefix)
    LCP1 = list(LCPrefix)

    # Replaces the first star with 0 or 1 to divide the elements of S to S1 and S2
```

```
    for x in range(len(LCP0)):
        if LCP0[x] == "*":
            LCP0[x] = "0"
            break

    for x in range(len(LCP1)):
        if LCP1[x] == "*":
            LCP1[x] = "1"
            break

    LCP0 = "".join(LCP0)
    LCP1 = "".join(LCP1)

    L0 = [] # Empty lists which will be filled by elements which have 0 in the location of the first *
    L1 = [] # Empty lists which will be filled by elements which have 1 in the location of the first *

    # Checking if that prefix family has LCP0 or LCP1, if present then we add to corresponding L0/L1
    for i in range(len(S)):
        prefixFamily = dPreSet[S[i]]
        if LCP0 in prefixFamily:
            L0.append(S[i])
        else:
            L1.append(S[i])

    return L0, L1 # returns the two subsets for further computation

# A type of overloaded function which prints the values as required by the algorithm
def printTheNumbers(L):
    temp1 = L[0]
    temp2 = L[1]
    temp1 = [dBinToNum[x] for x in temp1]
    temp2 = [dBinToNum[x] for x in temp2]

    # Checks which list is bigger, keeps the bigger list in temp1 and smaller in temp2
    if len(temp1) >= len(temp2):
        pass
    else:
        # Swaps if temp1 is smaller than temp2
        temp = temp1[:]
        temp1 = temp2[:]
        temp2 = temp[:]

    return temp1, temp2 # Returns the results

n = len(S) # Length of the main S

# This function goes over List L and checks if there is a possibility of combining two lists
# and still satisfy the condition of being less than or equal to the half of the initial List!
def checkCombo(L):
    i = -1 # default value
    j = -1 # default value
```

```
    for x in range(len(L)-1):
        for y in range(x + 1, len(L)):
            if (len(L[x]) + len(L[y])) <= math.ceil(n/2):
                i = x
                j = y
    return i, j

# A sub main function which runs the algorithm
def main(S, pastLCP, pastS3LCP):

    global sub1, sub2 # Stores the result.

    # This while splits the S ==> (S1, S2)
    while (len(S) > math.ceil(n/2)):
        S1, S2 = checkLCP(S, pastLCP)

        if len(S1) >= len(S2):
            L.append(S2)
            S = S1
        else:
            L.append(S1)
            S = S2

    L.append(S)

    # Checks if there are any sublists which can be combined, if they can then they will be combined
    while (checkCombo(L)):

        i, j = checkCombo(L) # Gets the index of the sublists which can be combined

        # Checking for default value, if so then we stop the combination process as there
        # are no sublists that can be combined!
        if i == -1 or j == -1:
            break

        # Copies the sublists to a and b
        a = L[i]
        b = L[j]

        Lij = L[i] + L[j] # Combining the two lists in temporary Lij
        tempD = {}

        del L[i] # Deleting the first sublist at index i

        # Deleting the second sublist at index j-1 as i has been deleted in the previous statement
        # so the index of j decreases by 1
        del L[j-1]

        L.append(Lij) # Adding the temporary combined L[i] and L[j]
```

```
# If there are only 2 sublists after the previous step then we return the result
if len(L) == 2:
    sub1, sub2 = printTheNumbers(L)
    return sub1, sub2 # two sublists which are the final result are returned
else:

    pickS3List = [] # Stores the intersection and sublists as a tuple from which S3 is picked

    # Find S3 via finding the subset who's prefix families share the least number of prefixes!
    for subset in L:
        preFamilies = []
        for x in subset:
            preFamilies.append(dPreSet[x])
        interSection = list(reduce(set.intersection, [set(item) for item in preFamilies]))
        pickS3List.append((len(interSection), subset))

    S3 = min(pickS3List)[1] # Picking the S3 which has the least interSection
    nS3 = len(S3) # stores len of S3


    while (len(L) == 3):
        try:
            L.remove(S3) # Removes S3 and then runs the algorithm!
        except:
            """ SPECIAL CASE - BRUTE FORCED THE SOLUTION """
            del L[-1]
            sub1, sub2 = printTheNumbers(L)
            return sub1, sub2

        # Splits the S3 ==> (S31, S32)
        while (len(S3) > math.ceil(nS3/2)):
            S31, S32 = checkS3LCP(S3, pastS3LCP)

            if len(S31) >= len(S32):
                L.append(S32)
                S3 = S31
            else:
                L.append(S31)
                S3 = S32

        ## Assigns bigger list to S1 and vice versa!
        if (len(L[0]) >= len(L[1])):
            S1 = L[0]
            S2 = L[1]
        else:
            S1 = L[1]
            S2 = L[0]

        ## Assigns smaller list to S31 and vice versa!
        # Combines S31 with S1/S2 and S32 with S2/S1 based on the algorithm!
        if (len(S31) <= len(S32)):
```

```
                pass
            else:
                temp = S31[:]
                S31 = S32[:]
                S32 = temp[:]

            if (len(S1) + len(S31)) <= math.ceil(n/2):
                S1 += S31
                if (len(S2) + len(S32)) <= math.ceil(n/2):
                    S2 += S32
                else:
                    S3 = S32[:]
                    L.append(S3)
            else:
                S2 += S31
                S3 = S32[:]
                L.append(S3)

        sub1, sub2 = printTheNumbers(L)
        return (sub1, sub2) # returns result to SuperMain

    res1, res2 = main(S, pastLCP, pastS3LCP)
    return res1, res2 # Returns results to the function call superMain(S, Number_of_Bits)

"""THIS IS THE START OF THE EXECUTION OF THE ALGORITHM"""

import sys
import random as rand # Used to generate random numbers
from binarytree import * # this will import the binary tree file
queryResult = [] # stores the final result of the query search

# This is the main function which will run the whole program
def main(SNum, Bits, theRange):
    import sys
    import hashlib
    sys.path.insert(0, 'C:/Users/Stark/Desktop/Programming/Coursework/Secure-Data-Analysis/Final
Project/MinPrefixCode/')
    import minPrefixGen as mp # file that generates the minPrefix

    # This function will return the trapDoor
    def getTrapDoor():

        # SHA-1 Hash function!
        def hashIt(s):
            s = str.encode(s)
            return hashlib.sha1(s).hexdigest()

        # Extract the keys from the treeGen function
        randK = getPrivateKeys()

        # prints theRange of the query
```

```
    print("The Range is:       " + str(theRange))
    print()

    # Extracts the minPrefixSet from the minPrefixGen module
    minPrefixSet = mp.main(theRange, Bits)

    # Takes the union of the generated minPrefixSet
    minPrefixSet = list(set(minPrefixSet))
    print("The MinPrefixSet is: " + str(minPrefixSet))
    print()

    trap = []
    tempTrap = []
    # For every Prefix in minPrefixSet it will create the trapDoor,
    # that is, one time hash for that node.
    for prefix in minPrefixSet:
        l = []
        ll = []

        for k in randK:
            element = k + prefix
            l.append(hashIt(element))
            ll.append(element)
        trap.append(l) # Contains the hashed values!
        tempTrap.append(ll) # Appends the normal element an primarily used to display

    return trap # Returns the one time hashed values of that nodes minPrefixSet


sys.path.insert(0, 'C:/Users/Stark/Desktop/Programming/Coursework/Secure-Data-Analysis/Final Project/BloomFilter/')
import bloomfilter as bf # file that will generate the bloom filter!

sys.path.insert(0, 'C:/Users/Stark/Desktop/Programming/Coursework/Secure-Data-Analysis/Final Project/SearchTree/')
import searchElement as search # Includes the search algorithm module

randK = [] # Generates the random k's
for i in range(7):
    randK.append(str(rand.randint(1, 1000)))

# This function returns the Privates Keys (randK) when called
def getPrivateKeys():
    return randK

# Tree Data structure Utilization
mytree = tree() # Creates a tree by name mytree
root = Node(SNum) # The root is set to the SNum initially
data = SNum # Data which will be divided into left and right child recursively
start = root # Head of the tree
parent = root # Initially parent is the head of the tree
```

```
# This function will recursively build the tree using SuperMain
def recBuildTree(x, data, parent):
    if len(x.value) == 1: # Checks if the node is leaf or not
        return parent # if its a leaf then it returns its parent
    else:
        # the SuperMain splits the data into two parts which will now become left and right child
        left, right = superMain(data, Bits)
        x.left = Node(left) # The left child
        x.right = Node(right) # The right child
        parent = x # Its the parent now for next iterations

        # this is a recursive algorithm which will divide the data into a tree
        recBuildTree(x.left, left, parent), recBuildTree(x.right, right, parent)

# Preorder Traversal through the tree to create the bloomFilters for each node!
def preorder(tree):
    if tree:
        # Retrieves the bloom filter for the node data
        bloom, vr, unionSET, secondhashed = bf.getBloomFilter(tree.value, Bits, randK)

        # Assigns the Bloom Filter, vr (Private key of the node) to the node data
        tree.value = (bloom, vr, unionSET[0], unionSET[1], secondhashed)

        # Runs recursively doing the same to the left and right children
        preorder(tree.left)
        preorder(tree.right)

trap = getTrapDoor()


def searchIT(tree):

    global queryResult
    if tree:
        data = tree.value # Gets the bloom filter and VR from the tree

        # Searches for the element in the bloom filter by calling the search module
        # if it returns PASS it means that the element was found in the bloom filter
        x = search.searchForME(data[0], data[1], data[4], trap, data[2])

        # Checks if the element found is child or not
        if x == "PASS":

            # if the element found is child then it appends it to the query results
            # thats the final result of the query search algorithm
            if not(tree.left or tree.right): # Verifies for leaf nodes
                if len(queryResult) == 0:

                    # Data[2] is the numerical representation of the element
                    # which is actually a bloom filter now.
```

```
                queryResult = data[2]
            else:
                queryResult += data[2]

        searchIT(tree.left), searchIT(tree.right)

    # Generating the tree, and running the whole build and search process
    def getTree():
        global queryResult
        print("Data is:          " + str(SNum))
        print()

        recBuildTree(root, data, parent) # Recursively builds the tree!
        pprint(start) # Prints the tree

        # Traverses the tree and replaces the data list with its bloom filter and vr
        preorder(start)

        # Recursively searches for the elements in the tree and generates the queryResult
        searchIT(start)
        print()

print("_____
_____
_____")
        print()
        print("Search Result(s): "  + str(sorted(queryResult))) # Prints the final Result of the query

print("_____
_____
_____")

    # This function call will create the tree normally and then traverse through it in Preorder
    # fashion and replace the nodes with the bloom filters!
    getTree() # <========= Starts the exectution

# Data and Search Query
theRange = [990000,1000000] # Search Query
SNum = [1, 6, 7, 9, 10, 11, 12, 16, 1000000] # Data items
Bits = max(max(theRange), max(SNum)).bit_length() # Selects the number of bits to the max value in
Range!

main(SNum, Bits, theRange) # This is the main function call to Run the whole program!
```