DESIGN-ASGN7 “The Great Firewall of Santa Cruz”

Purpose:

The purpose of this assignment is to build a hash table, bloom filter, and binary search trees to filter out the bad speech of the citizens of the GPRSC. The program takes in a list of badspeech words and newspeech words and creates a filter out of both lists made up of a bloomfilter, a hashtable, and layers of binary search trees. The program will then scan the input of the user and feed the text through the filter. If a badspeech word is detected, the user will be sent to joycamp. If only newspeech is detected, then the program will provide the list of words to change in their letter.

Node:

```
def init(oldspeak: str, newspeak: str):
      if (oldspeak):
            self.oldspeak = oldspeak
            if (!newspeak):
                  self.newspeak = NULL
            else:
                  self.newspeak = newspeak
            n.left = None
            n.right = None

def node_delete(n: Node):
      free(n.oldspeak)
      free(n.newspeak)
      free(n)
      n = None

def node_print(n: Node):
      if (!n):
            return
      if (n.oldspeak and n.newspeak):
            print(n.oldspeak+” -> “+n.newspeak+”\n”)
      else:
            print(n.oldspeak+”\n”)
```

BitVector:

```
def init(length: int):
```

```
      Bv = malloc(BitVector)
      If (bv):
            X = 0
            If (length %8):
                  X = 1
            else:
                  X = 0
            self.bytes = length/8 + (x)
            Self.vector = vector_init(uint8_t)
            Self.length = length

      else:
            return None

def bv_delete(bv: BitVector):
      if (bv and bv.vector):
            free (bv)
      if (bv):
            free(bv)
            bv = None

def bv_length(bv: BitVector):
      if (bv):
            return bv.length
      else:
            Return 0

#Cited from Professor Long
def bv_set_bit(bv: BitVector, i: int):
      if (i <= bv_length(bv) and bv and bv.vector):
            bv.vector[i/8] |= (0x1 << i % 8)
            return True
      return False

#Cited from Professor Long
def bv_clr_bit(bv: BitVector, i: int):
      if (i <= bv_length(bv) and bv and bv.vector):
            bv.vector[i/8] &= ~(0x1 << i % 8)
            return True
      return False
```

```
#Cited from Professor Long
def bv_get_bit(bv: BitVector, i: int):
      if (i <= bv_length(bv) and bv and bv.vector):
            bv.vector[i/8] >> (i % 8) & 0x1
            return True
      return False
```

BST (Binary Search Tree):

```
def init(void):
      Node bst = None
      Return bst

def bst_delete(root: Node):
      if (root):
            bst_delete(root.left)
            bst_delete(root.right)
            node_delete(root)

# Cited from Professor Long
def max(x: int, y: int):
      return x if x >y else y

# Cited from Professor Long
def bst_height(root: Node):
      if (root):
            return 1+max(bst_height(root.left), bst_height(root.right));
      Else:
            Return 0

# Cited from Professor Long
def bst_size(root: Node):
      if (root):
            return 1+max(bst_size(root.left), bst_size(root.right));
      Else:
            Return 0

# Cited from Professor Long
def bst_find(root: Node, oldspeak: str):
      if (root):
            if (root.oldspeak > oldspeak):
                  return bst_find(root.left, oldspeak)
            elif(root.oldspeak < oldspeak):
```

```
                  Return bst_find(root.right, oldspeak)
      Return root

# Cited from Professor Long
def bst_insert(root: Node, oldspeak: str, newspeak: str):
      If (!oldspeak):
            Return root
      if (!root):
            return node_create(oldspeak, newspeak)
      else:
            if (root.oldspeak > oldspeak):
                  Root.left = bst_insert(root.left, oldspeak, newspeak)
            elif(root.oldspeak < oldspeak):
                  Root.right = bst_insert(root.right, oldspeak, newspeak)
            Return root
```

BF(Bloom Filter):

```
def init(size):
      BloomFilter bf = malloc(sizeof(BloomFilter))
      bf.primary[0] = salt_primary_lo
      bf.primary[1] = salt_primary_hi
      bf.secondary[0] = salt_secondary_lo
      bf.secondary[1] = salt_secondy_hi
      bf.tertiary[0] = salt_tertiary_lo
      bf.tertiary[1] = salt_tertiarry_hi
      bf.filter =bv_create(size)
      Return bf

def bf_delete(bf: BloomFilter):
      bv_delete(bf.filter)
      free(bf)
      Bf = None

def bf_size(bf: BloomFilter):
      return bv_length(bf.filter)

def bf_insert(bf: BloomFilter, oldspeak: str):
      Hashed = hash(bf.primary, oldspeak) %bv_length(bf.filter)
      bv_set_bit(bf,filter, hashed)

      Hashed = hash(bf.secondary, oldspeak) %bv_length(bf.filter)
      bv_set_bit(bf,filter, hashed)
```

```
      Hashed = hash(bf.tertiary, oldspeak) %bv_length(bf.filter)
      bv_set_bit(bf,filter, hashed)


def bf_probe(bf: BloomFilter, oldspeak: str):
      Hashed = hash(bf.primary, oldspeak) %bv_length(bf.filter)
      if (!bv_get_bit(bf.filter, hashed)):
            return False

      Hashed = hash(bf.secondary, oldspeak) %bv_length(bf.filter)
      if (!bv_get_bit(bf.filter, hashed)):
            return False

      Hashed = hash(bf.tertiary, oldspeak) %bv_length(bf.filter)
      if (!bv_get_bit(bf.filter, hashed)):
            return False

      return True

def bf_count(bf: BloomFilter):
      Count = 0
      for i in range(len(bf_size(bf)):
            if (bv_get_bit(bf.filter, i):
                  count+=1
      return count
```

HT(HashTable):

```
def init(size: int):
      HashTable ht = malloc(sizeof(HashTable))
      if (ht):
            Ht.size = size
            Ht.salt[0] = SALT_HASHTABLE_LO
            Ht.salt[1] = SALT_HASHTABLE_HI
            Ht.trees = calloc(size, sizeof(node))
            if (!ht.trees):
                  Free ht
                  Ht = None
      Return ht

def ht_delete(ht: HashTable):
      if (ht and ht.trees):
```

```
            for i in range(len(ht.size)):
                  bst_delete(ht.trees[i])
            free(ht.trees)
            free(ht)
            ht = None

def ht_size(ht: HashTable):
      return ht.size

def ht_lookup(ht: HashTable, oldspeak: str):
      Index = hash(ht.salt, oldspeak) % ht.size
      Node n= bst_find(ht.trees[index], oldspeak)
      Return n

def ht_insert(ht: HashTable, oldspeak: str, newspeak: str):
      if (ht):
            Index = hash(ht.salt, oldspeak) % ht.size
            Ht.trees[i] = bst_insert(ht.trees[i], oldspeak, newspeak)
      Return

Def ht_count(ht: HashTable):
      Count = 0
      for i in range(len(ht.size)):
            if (ht.trees[i]):
                  count +=1
      Return count

def ht_avg_bst_size(ht: HashTable):
      avg, sum, denom
      for i in range(len(ht.size)):
            sum += bst_size(ht.trees[i])
            if (ht.trees[i]):
                  denom += 1
      avg = sum/denom
      return avg

def ht_avg_bst_height(ht: HashTable):
      avg, sum, denom
      for i in range(len(ht.size)):
            sum += bst_height(ht.trees[i])
            if (ht.trees[i]):
                  denom += 1
```

```
avg = sum/denom
return avg
```

Banhammer:

1) Opens the files, sets the hash size, bloom size, set commands, makes three lists to the data from the files
2) Opts an arguments from the user
3) If the user opted for help command, prints out the help instructions and frees the constructed lists
4) Creates bloom filter, hash table, badspeak bst and newspeak bst to store the bad words and newspeak words detected by the user
5) Inserts the badspeak words to the bloom filter and hashtable in lexicographical order and does the same for newspeak
6) Iterates through the user's input and checks if a word is in the bloom filter and hashtable. If it is and the word has a newspeak then the program will mark the user for a reprimand. However if the word has no newspeak then the user will be sent to joycamp
7) Stats will be printed if user opted
8) Will print out the list of words detected as well as the appropriate letter output
9) Frees memory