Tuan Nguen

Question 1:

It might be better to use an ordered array, given that buckets have more than one time in them. But since hash tables are created such that they hold only a few items, *O(1)* for lookups and *O(logn)* for insertion would not be that much faster and since their constants are probably going to be bigger, in many cases even slower than the unordered linked list.

Question 2:

The probability of an item being in the k-th bucket is $p = 1/N$, so for all *n* items, the distribution of the number of items in that bucket is binomial distribution *B(n, p)*. For the successful search, if the key that we are searching for is in a certain bucket, then the distribution of the rest of the keys is binomial *B(n-1, p)*. We know the mean of this distribution - $(n-1)p = (n-1)/N$. Suppose the average number of keys that we have to search to find our desired key is half of the number of items in the bucket, then the expected number of comparisons that we have to do is $\frac{n-1}{2N} + 1$. For the unsuccessful search, the expected number of comparisons that we have to do is the number of items in that bucket. Therefore, the expected number of comparisons is np.

Question 3:

```
class HashBag{
  private var MAX = 100
  private var table : Array[(String, Int)]=Array.fill(MAX)(("",0))
//Stores the word and its count
  private var size = 0
  //We start from the word's hash value and go through the bag until we
either find the word or reach the index from which we started

  //Abs: bag = {a=(word,count) | a <- L(table)}
  //DTI: table is finite, words are distinct

  private def hash(word: String) : Int = {  //Our hash function
    def f(e: Int, c: Char) = (e*41 + c.toInt) % MAX
    word.foldLeft(1)(f)
  }

  // Post: table = table_0 - (word, count) + (word, count+1)
  //       if word was already in the table
  // table = table_0 + (word,1) otherwise
  def add(word: String) : Unit = {
    var i = hash(word)
    // Invariant: The strings in table[hash(word)..i)
    // are non-empty and not equal to word
    while (table(i) != ("",0) && table(i)._1 != word){//Loop until empty
or equal to word
      i = (i + 1)%MAX
```

```
    }
    // Termination implies that table(i)._1 is either "" or word
    // => that is the right place to put the data
    if (table(i)._1 == ""){//If empty, put the word and set its count
to 1
      table(i)=(word,1)
      size += 1 //As we've filled another empty ?bucket
    }
    else table(i)=(word,(table(i)._2)+1)
    //If not empty, increase its count by 1
  }

  //Post: table = table_0 && returns the count of the word or 0 if the
word is not in the table
  def count(word: String) : Int = {
    var i = hash(word)
    val h = i
    if (table(i)._1 == word) return table(i)._2 //If found, return
immediately
    i = (i + 1) % MAX
    //  Invariant: The strings in table[hash(word)..i)
    // (may wrap around) are non-empty and not equal to word
    while (table(i)._1 != word && i != h) {
      //Loop until empty or equal to word
      i = (i + 1) % MAX
    }
    // Termination implies that either we've reached the index
    // where we started and therefore word is not in the table
    // Or table(i)._1==word and then we just return its count
    if (i == h) return 0
    else return table(i)._2
  }

  // Post: table = table_0 - (word, count) + (word, count+1)
  //       if word was in the table and its count>1
  // table = table_0 - (word, 1) + ("",0)
  //       if there was only one word hashed
  // table = table_0
  //       if word was not in the table
  def delete(word: String) : Unit = {
    var i = hash(word)
    val h = i
    if (table(i)._1 == word) {
      if (table(i)._2 == 1) {
        table(i) = ("", 0)
        size-=1 //As we've emptied the last element in a bucket
```

```
      }
      else table(i) = (word, (table(i)._2) - 1) //If it's not going to
be empty after we remove 1, decrease the count by 1
    }
    else {
      i = (i + 1) % MAX
      // Invariant: The strings in table[hash(word)..i)
      // (may wrap around) are non-empty and not equal to word
      while (table(i)._1 != word && i != h) {
        //Loop until empty or equal to word
        i = (i + 1) % MAX
      }
      if (i != h) { //i!=h => table._1 is the word
        if (table(i)._2 == 1) {
          table(i) = ("", 0)
          size -= 1 //As we've emptied the last element in a bucket
        }
        else table(i) = (word, (table(i)._2) - 1)
      }//if i==h => The word was not there to begin with => do nothing
    }
  }
}
```

Question 4:

```
import scala.collection.mutable.Stack;

case class Tree(var word: String, var left: Tree, var right: Tree){

  // Abs: tree = {word | word <- T(tree)}
  // DTI: T(tree) - finite and each tree has max 2 children

  // Post: tree = tree_0 && prints the tree in prefix order,
  // using indentation
  def recPrintTree() = recursivePrintTree(this, 0)
  private def recursivePrintTree(tree : Tree, depth : Int) : Unit = {
    for(i<-0 until depth) print(" . ")
    if (tree == null){
      println("null")
    }
    else {
      println(tree.word)
      recursivePrintTree(tree.left, depth + 1)
      recursivePrintTree(tree.right, depth + 1)
    }
  }
```

```
  //Post: tree = tree_0
  // && prints the tree in prefix order, using indentation
  //The stack version of printing the tree
  def stackPrintTree() : Unit = {
    val x : Stack[(Tree, Int)] = Stack[(Tree, Int)]()
    //Stack that will hold the roots of all the
    //subtrees that we still haven't printed out yet
    x.push((this, 0))//Start with the root(at the top)
    while (!x.isEmpty){ //So that it prints all values
      val (tree, depth) = x.top
      x.pop
      for(i<-0 until depth) print(" . ")
      if (tree == null) //When empty print out "null"
        println("null")
      else{
        println(tree.word)
        x.push((tree.right, depth+1))
        x.push((tree.left, depth+1))
        // We put the left node second, since that way
        // it will be reached first, when coming
        // from the top of the stack
      }
    }
  }
}
```

Question 5:

This is the implementation of binary tree and followed by the function that flips the tree.

```
case class Tree(var word: String, var left: Tree, var right: Tree){
  private var root : Tree = null

  private def add(word: String) = {
    if(root == null) root = Tree(word, null, null)
    else{
      var t = root
      while(word < t.word && t.left != null ||
        word > t.word && t.right != null){
        if(word < t.word) t = t.left else t= t.right
      }
      if(word < t.word) t.left = Tree(word, null, null)
      else if t.right = Tree(word, null, null)
      // Do nothing if the word is already in the tree
    }
  }
```

```
  def print = printTree(root)

  private def printTree(t: Tree): Unit = {
    if (t != null) {
      printTree(t.left)
      println(t.word + " -> " + t.count)
      printTree(t.right)
    }
  }
}

object Test{
  //Post: tree = flipped tree
  def flip(tree: Tree) : Unit = {
    if (tree != null) {
      val temp: Tree = tree.left
      tree.left = tree.right
      tree.right = temp; //Swap the left and right subtrees
      flip(tree.left)
      flip(tree.right) //Recurse on the subtrees
    }
  }
}
```

Question 6:

```
import scala.collection.mutable.Stack;

object BinaryTreeBag{
  private class Tree(var word: String, var count: Int, var left: Tree,
var right: Tree)
}

class BinaryTreeBag {
  private type Tree = BinaryTreeBag.Tree

  private def Tree(word: String, count: Int, left: Tree, right: Tree) =
    new BinaryTreeBag.Tree(word, count, left, right)

  private var root: Tree = null

  private def countInTree(word: String, t: Tree): Int = {
    if (t == null) 0
    else if (word == t.word) t.count
    else if (word < t.word) countInTree(word, t.left)
    else countInTree(word, t.right)
  }
```

```
def count(word: String): Int = {
  var t = root
  while (t != null && t.word != word)
    if (word < t.word) t = t.left else t = t.right
  if (t == null) 0 else t.count
}

private def addToTree(word: String, t: Tree): Tree = {
  if (t == null) Tree(word, 1, null, null)
  else if (word == t.word) {
    t.count += 1; t
  }
  else if (word < t.word) {
    t.left = addToTree(word, t.left); t
  }
  else {
    t.right = addToTree(word, t.right); t
  }
}

def add(word: String) = {
  if (root == null) root = Tree(word, 1, null, null)
  else {
    var t = root
    while (word < t.word && t.left != null ||
      word > t.word && t.right != null)
      if (word < t.word) t = t.left else t = t.right
    if (word == t.word) t.count += 1
    else if (word < t.word) t.left = Tree(word, 1, null, null)
    else t.right = Tree(word, 1, null, null)
  }
}

def print = printTree(root)

private def printTree(t: Tree): Unit = {
  if (t != null) {
    printTree(t.left)
    println(t.word + " -> " + t.count)
    printTree(t.right)
  }
}

def printIterative = {
  var t = root
```

```
  val stack = new scala.collection.mutable.Stack[Tree]
  while (t != null || !stack.isEmpty) {
    if (t != null) {
      stack.push(t); t = t.left
    }
    else {
      val t1 = stack.pop
      println(t1.word + " -> " + t1.count)
      t = t1.right
    }
  }
}

def delete(word: String): Unit = root = deleteFromTree(word, root)

private def deleteFromTree(word: String, t: Tree): Tree = {
  if (t == null) null
  else if (word < t.word) {
    t.left = deleteFromTree(word, t.left); t
  }
  else if (word > t.word) {
    t.right = deleteFromTree(word, t.right); t
  }
  else if (t.count > 1) {
    t.count -= 1; t
  }
  else if (t.left == null) t.right
  else if (t.right == null) t.left
  else {
    val (w, c, newR) = delMin(t.right)
    t.word = w;
    t.count = c;
    t.right = newR
    t
  }
}

private def delMin(t: Tree): (String, Int, Tree) = {
  if (t.left == null) (t.word, t.count, t.right)
  else {
    val (w, c, newL) = delMin(t.left)
    t.left = newL;
    (w, c, t)
  }
}
```

```
  def countAllNodes(): Int = countNodes(this.root)

  // Post: tree = tree_0 && returns total number of words in tree
  private def countNodes(tree: Tree): Int = {
    if (tree == null) return 0
    return tree.count + countNodes(tree.left) + countNodes(tree.right)
    // returns the count in the given tree
    // + the counts of its children, given recursively
  }

  def stackCountAllNodes(): Int = stackCountNodes(this.root)

  // Post: tree = tree_0 && returns total number of words in tree
  private def stackCountNodes(tree: Tree): Int = {
    val x: Stack[Tree] = Stack[Tree]()
    // x is going to hold the the roots of all the subtrees
    // that we still haven't added to the overall count
    var count = 0
    x.push(tree)
    // Invariant: x holds the roots of the subtrees
    // (at the beginning the whole tree) that we haven't dealt with yet
    // Variant: Number of uncounted nodes
    // => it decreases by 1 at each step and we will terminate when we've
    // dealt with all=> when the stack is empty
    while (!x.isEmpty) { //So that we add the counts of all the words
      val next = x.top
      x.pop
      if (next != null) {
        count = count + next.count //Add the top of the stack's count
        x.push(next.right)
        x.push(next.left)
        // Adds the two nodes that are the roots of the subtrees of the
        // top element, since they are the
      }
    }
    return count
  }
}
```

Question 7:

```
import scala.collection.mutable.Stack;

object BinaryTreeBag{
  private class Tree(var word: String, var count: Int, var left: Tree,
var right: Tree)
}
```

```
class BinaryTreeBag {
  private type Tree = BinaryTreeBag.Tree

  private def Tree(word: String, count: Int, left: Tree, right: Tree) =
    new BinaryTreeBag.Tree(word, count, left, right)

  private var root: Tree = null

  private def countInTree(word: String, t: Tree): Int = {
    if (t == null) 0
    else if (word == t.word) t.count
    else if (word < t.word) countInTree(word, t.left)
    else countInTree(word, t.right)
  }

  def count(word: String): Int = {
    var t = root
    while (t != null && t.word != word)
      if (word < t.word) t = t.left else t = t.right
    if (t == null) 0 else t.count
  }

  private def addToTree(word: String, t: Tree): Tree = {
    if (t == null) Tree(word, 1, null, null)
    else if (word == t.word) {
      t.count += 1; t
    }
    else if (word < t.word) {
      t.left = addToTree(word, t.left); t
    }
    else {
      t.right = addToTree(word, t.right); t
    }
  }

  def add(word: String) = {
    if (root == null) root = Tree(word, 1, null, null)
    else {
      var t = root
      while (word < t.word && t.left != null ||
        word > t.word && t.right != null)
        if (word < t.word) t = t.left else t = t.right
      if (word == t.word) t.count += 1
      else if (word < t.word) t.left = Tree(word, 1, null, null)
      else t.right = Tree(word, 1, null, null)
```

```
  }
}

def printT = printTree(root)

private def printTree(t: Tree): Unit = {
  if (t != null) {
    printTree(t.left)
    println(t.word + " -> " + t.count)
    printTree(t.right)
  }
}

def printIterative = {
  var t = root
  val stack = new scala.collection.mutable.Stack[Tree]
  while (t != null || !stack.isEmpty) {
    if (t != null) {
      stack.push(t); t = t.left
    }
    else {
      val t1 = stack.pop
      println(t1.word + " -> " + t1.count)
      t = t1.right
    }
  }
}

def delete(word: String): Unit = root = deleteFromTree(word, root)

private def deleteFromTree(word: String, t: Tree): Tree = {
  if (t == null) null
  else if (word < t.word) {
    t.left = deleteFromTree(word, t.left); t
  }
  else if (word > t.word) {
    t.right = deleteFromTree(word, t.right); t
  }
  else if (t.count > 1) {
    t.count -= 1; t
  }
  else if (t.left == null) t.right
  else if (t.right == null) t.left
  else {
    val (w, c, newR) = delMin(t.right)
    t.word = w;
```

```
      t.count = c;
      t.right = newR
      t
    }
  }

  private def delMin(t: Tree): (String, Int, Tree) = {
    if (t.left == null) (t.word, t.count, t.right)
    else {
      val (w, c, newL) = delMin(t.left)
      t.left = newL;
      (w, c, t)
    }
  }

  def depth() : Int = maxDepth(this.root)

  def maxDepth(root: Tree) : Int = {
    if(root == null) 0
    else Math.max(maxDepth(root.left), maxDepth(root.right)) + 1
  }



  def depths() : (Int,Int) = getDepths(this.root)

  //Post: tree = tree_0 and returns (min depth of the tree, max depth
of tree)
  private def getDepths(tree: Tree): (Int, Int) = {
    // Base case
    if(tree.left == null && tree.right == null) (1, 1)
    else if(tree.left == null){
      var (min, max) = getDepths(tree.right)
      (min + 1, max + 1)
    }
    else if(tree.right == null){
      var (min, max) = getDepths(tree.left)
      (min + 1, max + 1)
    }
    else{
      val (leftmin, leftmax) = getDepths(tree.left) //Get the max and
      val (rightmin, rightmax) = getDepths(tree.right) //(right subtree
min depth, right subtree max depth)
      var min = Math.min(1 + leftmin, 1 + rightmin)
      var max = Math.max(1 + leftmax, 1 + rightmax)
```

```
      return (min, max) //add 1 since we have to count the level we're
at
    }
  }

  def stackDepths() : (Int,Int) = getStackDepths(this.root)

  //Post: tree = tree_0 and returns (min depth of tree, max depth of
tree)
  private def getStackDepths(tree: Tree): (Int, Int) = {
    if (tree == null) return (0, 0)  //Case when we can't add a tree to
the stack
    val x: Stack[(Tree, Int)] = Stack[(Tree, Int)]()
    x.push((tree, 1))
    var k = tree
    var i = 1 //Value to hold a depth from the top to a leaf
    //To get the depth from the top to the leftmost leaf to use as basis
for comparison
    while(k.left != null){
      k = k.left
      i = i+1
    }
    var max: Int = i //Default set to i
    var min: Int = i //Default set to infinity

    //Invariant: x = stack of all the roots of the subtrees, we have to
go through
    // max = max(i, maximum depth of a leaf that's in the tree, but not
in the subtrees that x defines)
    // min = min(i, minimum depth of a leaf that's in the tree, but not
in the subtrees that x defines)
    while (!x.isEmpty) {
      val (next, depth): (Tree, Int) = x.top
      x.pop
      if (next.left == null && next.right == null) { //This would mean
that next is a leaf=> we compare the depth of that leaf to the current
min and max
        if(depth > max) max = depth
        if(depth < min) min = depth
      }
      if (next.right != null) { //If there is a subtree to the right,
add its root to the stack
        x.push((next.right, depth + 1))
      }
      if (next.left != null) { //If there is a subtree to the left, add
its root to the stack
```

```
        x.push((next.left, depth + 1))
      }
    }
    (min,max)
  }

  //Post: tree = tree_0 && prints the tree in prefix order, using
indentation
  def recPrintTree() = recursivePrintTree(root, 0)
  private def recursivePrintTree(tree : Tree, depth : Int) : Unit = {
    for(i<-0 until depth)print(" . ")
    if (tree == null){
      println("null")
    }
    else {
      println(tree.word)
      recursivePrintTree(tree.left, depth + 1)
      recursivePrintTree(tree.right, depth + 1)
    }
  }
}
```

Question 8:

```
object Question8{
  private val fileName = "C:\\Users\\TUAN\\Desktop\\knuth_words"
  private val words1 = new scala.collection.mutable.HashSet[String]

  private def initDict(fname: String) = {
    val allWords = scala.io.Source.fromFile(fname).getLines
    for(w <- allWords) words1 += w
  }

  initDict(fileName)

  def perm(s: String) : Unit = {
    var allPossiblePermutations = s.permutations // List of all
permutations
    for(w <- allPossiblePermutations){
      if(words1.contains(w)) println(w)
    }
  }
}
```

This approach is rather slow for longer string, because it needs to calculate all the permutations of the string. For a string of length *n*, the number of permutations is *n!*, which grows with order *O(n^n)*.

```
object Question8.2{
  private val fileName = "C:\\Users\\TUAN\\Desktop\\knuth_words"
  val bufferedSource = io.Source.fromFile(filename)
  val words = (for (line <- bufferedSource.getLines())
         yield (line.sorted, line)).toList.sorted
  //words is an array that stores (sorted word, word) for all words in
the knuth_words file
  bufferedSource.close

  //Post: s=s0 and returns -1 if there isn't an anagram to s, otherwise
returns x s.t. words(x)._1=s
  def findAnagram(s: String): Int = {
    var l = 0
    val ssorted = s.sorted
    var r = words.size
    var k = (l + r) / 2
    while (l < r - 1) {//Binary search in the first argument in words
      k = (l + r) / 2
      val (a, b) = words(k)
      if (ssorted.compareTo(a) > 0) {
        l = k
      }
      else r = k
    }
    val (a, b) = words(k + 1)
    if (ssorted == a) k + 1
    else -1
  }

  //post: s=s0 and returns an array of strings, containing all the
anagrams of s
  def allAnagrams(s: String) : Array[String] = {
    var k = findAnagram(s) //find an anagram to s
    if (k != -1) {
      val ssorted = s.sorted
      while (k > 0 && words(k - 1)._1 == ssorted) {//find the first
element that is an anagram to s
        k = k - 1
      }
      var t = k
      while (words(t)._1 == ssorted) {//find the last element that is an
anagram
        t = t + 1
      }
      var all = new Array[String](t - k)
```

```
      for(i<-0 until t - k) all(i) = words(i + k)._2 //add the words
from k to t to all, since they are anagrams to s
      return all
    }
    Array() //if findAnagram gave -1 then there is no anagram to the
given word
  }
}
```

Extra Question:

```
object Question9{
  /** Prints all the permutations of an array in lexicographic order */
  def permute(a: Array[Int], l: Int, r: Int) : Unit = {
    if(l == r){
      for(i <- a) print(i + " ")
      println
    }
    else{
      var i = l

      print("This is before while: ")
      for(i <- a) print(i + " ")
      println

      /** Invariant: a[l..i] = a0[l..i] && a[i..r] are yet to be permuted
        *         && l <= i <= r */
      while(i <= r){
        var swap = a(l)
        a(l) = a(i)
        a(i) = swap
        permute(a, l+1, r)

        print("This is before backtracking: ")
        for(i <- a) print(i + " ")
        println

        // Backtracking
        swap = a(l)
        a(l) = a(i)
        a(i) = swap
        i += 1
      }
    }
  }

  /** Partition the segment a[l..r)
```

```
    * Post: return k s.t. a[l..k) < a[k..r) and l <= k < r */
  def partition(a: Array[Int], l: Int, r: Int) = {
    /** Invariant: a[l+1..i) < x = a(l) <= a[j..r) && 1 < i <= j <= r
      *         && a[0..l) = a0[0..l) && a[r..N) = a0[r..N)
      *         && a[l..r) is a permutation of a0[l..r) */
    var x = a(l)
    var i = l+1; var j = r
    while(i < j){
      if(a(i) < x) i += 1
      else{ val swap = a(i); a(i) = a(j-1); a(j-1) = swap; j -= 1 }
    }
    // Swap pivot into position
    a(l) = a(i-1); a(i-1) = x
    i - 1 // Position of the pivot
  }

  /** Sorts an array in ascending order
    * Post: a = a0 && a is a permutation of a0
    *    && a(i) <= a(j) for l <= i < j <= r
    *    && a[0..l) = a0[0..l) && a[r..N) = a0[r..N) */
  def QSort(a: Array[Int], l: Int, r: Int) : Unit = {
    if(r - l > 1){
      val k = partition(a, l, r)
      QSort(a, l, k); QSort(a, k+1, r)
    }
  }
}
```