Problem Sheet 6

Question 1

To answer this question, we first need to ask what kinds of datatypes allow for a faster lookup than linear time. Here are some examples:

- Arrays - these are only good to use if the approximate maximum size of the `hashTable` is known. If we know there to be a million items approximately, and we have 100 of these buckets, we can confidently size each array to be of size 10,000. Lookup in an array is constant time.
- Binary search trees - We can introduce a binary search tree in which we don't need to know or specify the maximum size of it when we initialise it. However, the cost of doing so results the lookup time to be reduced from constant to logarithmic - which is still exponentially better than linked lists. A problem with this data-type is that the type we store inside must have an implementation of the `compare` method in order for this to work.

Question 2

We can state some basic probabilities by interpreting the question. For a given $i \in [0, N]$, the probability that the i-th bucket was picked will be $P(X = i) = \frac{1}{N}$.

Given that we can have k many different items in each bucket, and that the chance that each item goes in the i-th bucket is essentially just a Bernoulli trail - we can use some probability facts to state that the sum of each of the k_i items sum to a Binomial distribution. Therefore, our probability distribution for all of the k items in each bucket is:

$$\mathbb{P}[k : n, N] = \binom{n}{k}\left(\frac{1}{N}\right)^k\left(1 - \frac{1}{N}\right)^{n-k}$$

And by that, the expected number of searches would be

$$\mathbb{E}[k : n, N] = \frac{n}{N}$$

and for an unsuccessful search, just one more search than the expected number of searches would be required in order to see that the item doesn't exist.

Question 3

```scala
class BagOfWords(val MAX: Int) {
  // Entry class defined within BagOfWords
  class Entry(var word: String, var count: Int, var status: EntryStatus.Value)

  // Companion object for EntryStatus enumeration within BagOfWords
  object EntryStatus extends Enumeration {
    val Active, Deleted, Empty = Value
  }

  private val table = Array.fill[Entry](MAX)(new Entry(null, 0,
EntryStatus.Empty))

  def add(word: String): Unit = {
    val index = findIndex(word, insert = true)
    if (index != -1) {
      if (table(index).status != EntryStatus.Active) {
        table(index).word = word
        table(index).count = 1
        table(index).status = EntryStatus.Active
      } else {
        table(index).count += 1
      }
    } else {
      println("Table is full. Cannot add more words.")
    }
  }

  def find(word: String): Int = {
    val index = findIndex(word, insert = false)
    if (index != -1 && table(index).status == EntryStatus.Active)
table(index).count else 0
  }

  def delete(word: String): Unit = {
    val index = findIndex(word, insert = false)
    if (index != -1 && table(index).status == EntryStatus.Active) {
      table(index).status = EntryStatus.Deleted
    }
  }

  private def findIndex(word: String, insert: Boolean): Int = {
    var hash = word.hashCode % MAX
    if (hash < 0) hash += MAX // Ensure positive index
    var steps = 0
    while (steps < MAX && table(hash).status != EntryStatus.Empty &&
           !(table(hash).word == word && table(hash).status ==
EntryStatus.Active)) {
      hash = (hash + 1) % MAX
      steps += 1
    }
    if (steps >= MAX) -1 // Table full or not found
    else hash
  }
```

```
}
```

Question 4

Part a:

```scala
class Tree(var word: String, var left: Tree, var right: Tree)

object Tree {
  // Helper method to create new Tree instances more easily
  def apply(word: String, left: Tree, right: Tree): Tree = new Tree(word, left,
right)

  // Recursive procedure to print a tree in prefix order with indentation
  def printTree(tree: Tree, depth: Int = 0): Unit = {
    if (tree != null) {
      println("." * depth + tree.word) // Print current node with indentation
      printTree(tree.left, depth + 1)  // Recursively print left subtree
      printTree(tree.right, depth + 1) // Recursively print right subtree
    } else {
      println("." * depth + "null") // Print "null" for empty branches
    }
  }
}
```

Part b:

```scala
import scala.collection.mutable

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

object Tree {
  def apply(word: String, left: Tree, right: Tree): Tree = new Tree(word, left,
right)

  // Iterative procedure to print a tree using a stack
  def printTreeIterative(root: Tree): Unit = {
    case class TreeNodeDepth(node: Tree, depth: Int)

    val stack = mutable.Stack[TreeNodeDepth]()
    stack.push(TreeNodeDepth(root, 0))

    while (stack.nonEmpty) {
      val current = stack.pop()
      val depth = current.depth
      val node = current.node
      if (node != null) {
        println("." * depth + node.word)
        // Push right child first so that left child is processed first
        stack.push(TreeNodeDepth(node.right, depth + 1))
        stack.push(TreeNodeDepth(node.left, depth + 1))
      } else {
        println("." * depth + "null")
      }
    }
  }
}
```

I've used the already given implementation of the Stack from the mutable collection. In this case, we define a helper case class `TreeNodeDepth` so we can composite both the depth and the Tree node with each other to make this implementation easier.

We keep popping out of the stack after we push our tree into it, and print out the nodes in order of its depth.

Question 5

Here is a simple recursive implementation of the `flip` function:

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

def flip(t: Tree): Unit = {
  if (t != null) {
    // Swap the left and right children
    val temp = t.left
    t.left = t.right
    t.right = temp

    // Recursively flip the subtrees
    flip(t.left)
    flip(t.right)
  }
}
```

Question 6

Part a:

```scala
case class Tree(word: String, count: Int, left: Tree = null, right: Tree = null)

def totalWords(t: Tree): Int = {
  if (t == null) {
    0
  } else {
    t.count + totalWords(t.left) + totalWords(t.right)
  }
}
```

Part b:

```
import scala.collection.mutable.Stack

case class Tree(word: String, count: Int, left: Tree = null, right: Tree = null)

def totalWordsIterative(root: Tree): Int = {
  val stack = Stack[Tree]()
  var total = 0

  // Initial push
  if (root != null) stack.push(root)

  // Loop invariant: At any point, the stack contains nodes that are yet to be
visited.
  // Loop variant: The size of the stack decreases with each iteration until it's
empty.
  while (stack.nonEmpty) {
    val current = stack.pop()
    total += current.count

    if (current.right != null) stack.push(current.right)
    if (current.left != null) stack.push(current.left)
  }

  total
}
```

Question 7

To answer this question, we can decompose the problem into two steps: One step is to generate the permutations, and the next is to filter the permutations with the use of a dictionary.

Here is the procedure for generating the permutations for the words:

```
def generatePermutations(word: String): Set[String] = {
  if (word.length <= 1) {
    Set(word)
  } else {
    for {
      i <- word.indices.toSet
      perm <- generatePermutations(word.take(i) + word.drop(i + 1))
    } yield word(i) + perm
  }
}
```

It uses sets to make it easier to deal with duplicates. And here is the procedure for finding the anagrams.

```
def findAnagrams(word: String, dictionary: Set[String]): Set[String] = {
  val permutations = generatePermutations(word)
  permutations.intersect(dictionary)
}
```

Simply put, this algorithm is not efficient since it does unnecessarily more steps than it needs to. There isn't a need to "brute force" all of the permutations of the string and cause our algorithm to run in $\Theta(n!)$. A better solution may involve dynamic programming, or as hinted in the next part - looking up from a dictionary of anagrams.

Part b: Just as we did for the first part, we can decompose our problem into three parts - one for generating all distinct permutations (where order matters), implementing our data structure to help us with this, and then creating said an agrammatical dictionary.

Here is how we could create our dictionary:

```
def createAnagramDictionary(words: Seq[String]): Map[String, Seq[String]] = {
  words.groupBy(_.sorted)
}
```

and then use it along with the following:

```
def generatePermutations(word: String): Seq[String] = {
  if (word.isEmpty) Seq("")
  else for {
    i <- 0 until word.length
    perm <- generatePermutations(word.take(i) + word.drop(i + 1))
  } yield word(i) + perm
}

def findAnagrams(word: String, dictionary: Set[String]): Set[String] = {
  generatePermutations(word).filter(dictionary.contains).toSet
}
```

Although the `generatePermutations` function still runs in factorial time - if used in succession with the dictionary as well as with a filter - the time complexity can jump from n! to n^2, since instead of considering permutations, we consider ordered sequences, ideally speaking.