IMPERATIVE PROGRAMMING HT2018

SHEET 6

GABRIEL MOISE

Question 1

/*

We can replace each bucket in our hash table with a binary search tree, which will be balanced if we add another hash function which is completely independent from the initial one. Thus, we can maintain a balanced binary search tree, so we can add data in O(log2(N)) time and also search in O(log2(N)). We should maintain the property hash2(leftchild) < hash2(parent) < hash2(rightchild) in every binary search tree we create.

*/

Question 2

/*

Each item has a probability of 1/N to go in any specific bucket because their hash functions are uniform on [0..N). Therefore, the distribution of the number of items that go into a specific bucket is Bin(n,1/N), and the mean of this is n/N. For an unsuccessful search, we compare the item with every item from the bucket, so the distribution of the number of comparisons for an unsuccessful search is the same, with the expected number of comparisons n/N. For a successful search, let's suppose that we have k keys to retrieve and the number of operations needed to retrieve all of them is 1 + 2 + ... + k = k(k+1)/2. So, if the number of items is X, which is a random variable, the expectation for this is: E = sum for k (k(k+1)/2 * P(X = k)). Since X has binomial distribution Bin(n,1/N), we get: E = 1/2*(E(X^2) + E(X)) = n/N * ((n-1)/2*N - 1) So, if we want to get the mean cost per key retrieved we divide by n and to obtain the total expected cost we multiply by N to get : E = (n-1)/2*N - 1

*/

Question 3

```
// A hash table, representing a bag of words; i.e., for each word we record how many times the word is stored.


object ArrayBag{

  private class Node (val word : String, var count : Int)

}


// DTI : table(hash(word)) = Node(word,count) if it was null when we added the word, or table(pos) = Node(word,count) where pos is the first null position found starting from hash(word) and wrapping around at MAX && there are size_ words in the table and size_ <= MAX. So, basically we have between hash(word) and pos all places non-null. Also, a node containing word can only appear once in table!


class ArrayBag{

  // The hash function we will use

  private def hash(word: String) : Int = {

    def f(e: Int, c: Char) = (e*41 + c.toInt) % MAX

    word.foldLeft(1)(f)

  }


  private val MAX = 100

  private var size_ = 0 // # distinct words stored
```

```
private val table = new Array[ArrayBag.Node](MAX)
```

// Given a word and h = hash(word), this function finds the index in table where word is or if it is not, the first empty place to put the word in(if we find an empty space, then after it there cannot be any appearance of the word we are looking for as that word would have been placed in the first empty space found the previous time). We can also find a place where table(n).count = 0, meaning there was a number of appearances of a specific string which got deleted until count became 0, so we can reuse that space.

```
private def find (word: String, h : Int) : Int =
  {
    var n = h
    while ((table(n) != null) && (table(n).word != word) && (table(n).count != 0)) n = (n+1) % MAX
    n
  }

/** Add an occurrence of word to the table */
def add(word: String) = {
  val h = hash(word)
  var n = find(word, h)
  if ((table(n) == null) || (table(n).count == 0)) table(n) = new ArrayBag.Node (word, 1) // word was not in table or got deleted
    else table(n).count += 1 // word was already in table
  size_ += 1
}

/** The count stored for a particular word */
def count(word: String) : Int = {
  val h = hash(word)
  val n = find(word, h)
  if (table(n) != null) return table(n).count
    else return 0
}

// return the size
def size = size_

/** Deletes word if it is in the bag and returns true if the word was found in the bag and false otherwise*/
def delete(word: String) : Boolean=
  {
```

```
    val h = hash(word)
    var n = find(word,h)
    if ((table(n) != null) && (table(n).count > 0))
     {
       table(n).count -= 1
       return true
     }
    else return false
  }
}
```

Question 4

```
object Question4
{
 case class Tree (var word: String, var left: Tree, var right: Tree)

 // (a)
 def printTreeRec (t : Tree, height : Int) : Unit=
  {
    var str = ". " * height
    if (t != null)
    {
      println(str + t.word)
      printTreeRec(t.left,height+1)
      printTreeRec(t.right,height+1)
    }
    else println(str + "null")
  }

 // (b)
 // Implementing a Stack with linked lists:
 object Stack // I received a warning message when I tried to use the Stack from sala.collection, so I created one for myself
 {
   class Node (var tree : Tree, var height: Int, var next : Node)
 }
 class Stack
 {
```

```
 var stack = new Stack.Node (null, 0, null)

 // we add at the front of the stack
 // Post : stack = (t,h) : stack_0
 def push (t : Tree, h: Int) =
  {
   var n1 = new Stack.Node(t, h, stack.next)
   stack.next = n1
  }

 // We get an element from the stack from the front and we delete it from the stack
 // Pre : stack is not empty
 // Post : stack = tail(stack_0) && and returns head(stack_0)
 def pop : (Tree,Int) =
 {
  require(stack.next != null)
  var (t,h) = (stack.next.tree, stack.next.height)
  stack.next = stack.next.next
  return (t,h)
 }
 // Checks if the stack is empty or not
 def isEmpty : Boolean = (stack.next == null)
}

def printTreeStack (t: Tree, height : Int) : Unit =
 {
  var st = new Stack
  st.push(t, 0)
  while (st.isEmpty == false)
  {
   var (t,h) = st.pop
   if (t == null)
   {
    var str = ". " * h
    println(str + "null")
   }
```

```
      else
      {
        st.push(t.right, h+1) // first we print the left and then the right subtree
        st.push(t.left, h+1)
        var str = ". " * h
        println(str + t.word)
      }
    }
  }

  def main (args: Array[String]) =
  {
    var tr = Tree("three", Tree("four", Tree("five",null,null), Tree("six", Tree("seven",
           Tree("one",null,null), null), null)), Tree("two",null,null))
    printTreeStack(tr,0)
  }
}
```

Question 5

```
object Question5
{
  case class Tree (var word : String, var left : Tree, var right : Tree)

  /** Function that destructively flips the tree t, exchanging left and right throughout */
  // tree.word remains the same, (t.left,t.right) becomes (t.right,t.left) and we recursively flip the two subtrees
  def flip(t: Tree) : Unit =
  {
    if (t != null)
    {
      var leftTree = t.left
      var rightTree = t.right
      t.right = leftTree
      t.left = rightTree
      flip (t.left)
      flip (t.right)
    }
  }
```

```
}
```

Question 6

```
object BinaryTreeBag

{

  private class Tree(var word: String, var count: Int, var left: Tree, var right: Tree)


  // I implemented the Stack as for question 4 too, because I get an error when I tried to use the scala.collection.mutable.Stack: warning: class Stack in package mutable is deprecated (since 2.12.0): Stack is an inelegant and potentially poorly-performing wrapper around List. Use a List assigned to a var instead.

  private class Stack

  {

    case class Node (var tree : BinaryTreeBag.Tree, var next : Node)


    var stack = new Node (null, null)


    def push (t : BinaryTreeBag.Tree) =

     {

       var n1 = new Node(t,stack.next)

       stack.next = n1

     }


    def pop : BinaryTreeBag.Tree =

    {

      require(stack.next != null)

      var t = stack.next.tree

      stack.next = stack.next.next

      return t

     }


    def isEmpty : Boolean = (stack.next == null)

  }

}


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

  // (a) Recursive definition for the function size, which adds the count fields of all the nodes from the tree
  private def sizeRec(t: Tree) : Int =
   {
     if (t != null) return t.count + sizeRec(t.left) + sizeRec(t.right)
       else return 0
   }

  // (b) Iterative version of the function size, using a stack to keep track of the parts of the tree still to be considered
  private def sizeIter (t: Tree) : Int =
   {
     var size = 0
     val stack = new BinaryTreeBag.Stack
     stack.push(t)
     // Invariant : We still need to add the count fields of the current node and of its nodes from the left and right subtrees and of the right child (if
the current node is the left child of a node)
     while(stack.isEmpty == false)
     {
      var tr = stack.pop
      if (tr != null) {size += tr.count; stack.push(tr.left); stack.push(tr.right)}
         else {}
     }
     size
   }
}
```

Question 7

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

  private class Stack
  {
    case class Node (var tree : BinaryTreeBag.Tree, var depth : Int, var next : Node)

    var stack = new Node (null, 0, null)
```

```
    def push (t : BinaryTreeBag.Tree, d: Int) =
     {
       var n1 = new Node(t,d,stack.next)
       stack.next = n1
     }

    def pop : (BinaryTreeBag.Tree, Int) =
    {
      require(stack.next != null)
      var (t,d) = (stack.next.tree,stack.next.depth)
      stack.next = stack.next.next
      return (t,d)
     }

    def isEmpty : Boolean = (stack.next == null)
  }
}

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

  // We want to calculate the minimum and the maximum depth of the tree, at any given point

  // (a) Using a recursive function

  private def depthRec (t: Tree) : (Int,Int) =
    {
     if (t == null) return (0,0)
       else
       {
         var (minLeft,maxLeft) = depthRec(t.left)
```

```
      var (minRight,maxRight) = depthRec(t.right)
      var min = 0
      if (minLeft < minRight) min = minLeft
        else min = minRight
      var max = 0
      if (maxLeft < maxRight) max = maxRight
        else max = maxLeft
      return (min+1,max+1)
    }
  }

// (b) Using an iterative function, and making use of a Stack

private def depthIter (t: Tree) : (Int,Int) =
  {
    var min = 10000000
    var max = 0
    val stack = new BinaryTreeBag.Stack
    stack.push(t,0)
    while (stack.isEmpty == false)
    {
      var (tr,depth) = stack.pop
      if (tr == null) //we reached a leaf
      {
        if (depth < min) min = depth
        if (depth > max) max = depth
      }
      else {stack.push(t.left,depth+1); stack.push(t.right,depth+1)}
    }
    return (min,max)
  }
}
```

Question 8

```
class AnagramsDictionary(fname: String){
  /** An array holding the pairs of the order permutations with the words*/
```

```
  var words = new Array [(String,String)] (120000) // the maximum number of words from knuth_words

  var count = 0

  /** Initialise anagrammatical dictionary from fname */
  private def initDict(fname: String) = {
    val allWords = scala.io.Source.fromFile(fname).getLines
    def include(w:String) = w.forall(_.isLower)
    for(w <- allWords; if include(w)) {words(count) = (w.sorted,w) ; count += 1}
  }

  initDict(fname)

  var anagrams = new Array [(String,String)] (count) // we create a new array with no empty cells so that we can sort it

  for (i<-0 until count) anagrams(i) = words(i)

  // we sort the anagrammatical dictionary
  anagrams = anagrams.sorted
}

object Question8
{
  var dict = new Dictionary("knuth_words.txt")

  // (a) The program is very slow on long strings because the time complexity is O(n^n) and the memory O(n!) for the list
  class List
  {
    case class Node (var word : String, var next : Node)

    var list = new Node ("?", null)

    var end = list

    // We add a new node at the end of the list
    def add (word : String) =
```

```
  {
    var n1 = new Node(word,null)
    end.next = n1
    end = n1
  }

 // We get a word from the head of the list, deleting its node
 // Pre : the list is not empty
 def get : String =
  {
    var word = list.next.word
    if (list.next == end) {list.next = null; end = list} // we only change end if the list consisted of one element which was removed
        else list.next = list.next.next
    word
  }
 override def toString : String =
  {
    var str = "{"
    var current = list.next
    while (current != null)
    {
      if (current.next != null) str = str + current.word + ", "
         else str = str+current.word
      current = current.next
    }
    str = str + "}"
    str
  }
}

def permutations (word : String) : List =
 {
   // In perm, we start with the last letter of the word and then we add a new letter to all the words we created so far (in all the positions) of length ln-1, which will be 2 at the first while-loop step, and add them at the end. Then we repeat until we are left with a list consisting of all the permutations of length word.length
   var N = word.length
   var perm = new List
```

```
    var pos = N-1 // the current position of the letter we add to all the words in perm

    perm.add(word.drop(N-1))

    var ln = 1

    // Invariant : perm contains all the possible permutations of word[(N-ln)..N)

    while (ln < N)

    {

     ln += 1

     var ch = word(N-ln)

     // We get nodes from the list, we put ch in each position to form new sub-permutations and we add them to perm, until we get to a node with length ln (one that was added during this while iteration)

     while (perm.list.next.word.length == ln - 1)

     {

      var subword = perm.get

      // We insert ch in every position (we form ln words)

      for (i<-0 until ln) {var newWord = subword.take(i) + ch + subword.drop(i); perm.add(newWord)}

     }

    }

    // Now we have in perm all the permutations of the initial word

    perm

   }


 // Given a list of words, it returns the list containing all the words from the list that are in the dictionary

 def spell_check (perm : List) : List =

   {

    var wordsDict = new List // the list that will the correct words (from our dictionary)

    var current = perm.list.next

    while (current != null)

    {

     if (dict.isWord(current.word)) wordsDict.add(current.word)

     current = current.next

    }

    wordsDict

   }


 // (b)


 var anagramsDict = new AnagramsDictionary("knuth_words.txt")
```

```
val sizeDict = anagramsDict.anagrams.size


// We search the sorted permutation of the given word in the array (we consider a to be the array consisting of only the first entry of each tuple
from our initial array)

def search(x: String) : Int = {

  // invariant I: a[0..i) < x <= a[j..sizeDict) && 0 <= i <= j <= sizeDict

  var i = 0; var j = sizeDict

  while(i < j){

    val m = (i+j)/2 // i <= m < j

    if(anagramsDict.anagrams(m)._1 < x) i = m+1 else j = m

  }

  // I && i = j, so a[0..i) < x <= a[i..N)

  i

}


val value = 18 // by searching for anagrams in the anagramDict.anagrams array we found the the max length is 18, so we print those with length
18


def main (args: Array[String]) =

  {

    val word = scala.io.StdIn.readLine

    // (a)

    println(spell_check(permutations(word)).toString)


    // (b)

    var pos = search(word.sorted)

    print("{"+ anagramsDict.anagrams(pos)._2 + ", ")

    // From pos+1 we search for the anagrams of word

    var j = pos + 1

    while (anagramsDict.anagrams(j)._1 == anagramsDict.anagrams(pos)._1)

    {

      if (anagramsDict.anagrams(j+1)._1 == anagramsDict.anagrams(pos)._1) print(anagramsDict.anagrams(j)._2 + ", ")

      else print(anagramsDict.anagrams(j)._2)

      j = j + 1

    }

    println("}")
```

```
    /* Finding the longest anagrams from the knuth dictionary:
    var i = 0
    while (i < sizeDict)
    {
      if ((i < sizeDict - 1) && (anagramsDict.anagrams(i)._1 == anagramsDict.anagrams(i+1)._1))
      {
        if (anagramsDict.anagrams(i)._2.size == value) print(anagramsDict.anagrams(i)._2+" ")
        while ((i < sizeDict - 1) && (anagramsDict.anagrams(i)._1 == anagramsDict.anagrams(i+1)._1))
        {
          if (anagramsDict.anagrams(i)._2.size == value) print(anagramsDict.anagrams(i+1)._2+" ")
          i = i + 1
        }
        if (anagramsDict.anagrams(i)._2.size == value) println()
      }
      else i = i + 1
    }
    */
    // It turns out that the anagrams of length 18 from knuth_words are pathophysiological and physiopathological.
    // It also turns out that the longest set of anagrams is:
    // {least, setal, slate, stale, steal, stela, tales, teals, tesla}
    // which has 9 anagrams.
  }
}
```