IMPERATIVE PROGRAMMING HT2018

SHEET 5

GABRIEL MOISE

**Question 1**

object Question1

{

var myList : Node = null

class Node (var datum : Int, var next : Node)

{

// (b)

override def toString : String =

{

var str = ""

var pos = myList

// Invariant I : the string str contains numbers from the head of the list until pos.datum

while (pos != null)

{

if (pos.next != null) str = str + pos.datum + " -> "

else str = str + pos.datum // The last element doesn't have a "->"

pos = pos.next

}

// I && pos = null => str contains every number from the list

str

}

// >scala Question1.scala

// List is 12 -> 11 -> 10 -> 9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1.

}

// (c)

def reverse =

{

// Reversing the order of the linked list by reversing the direction in which the list is linked

var prev : Node = null

var current = myList

var next : Node = null

// Invariant : the linked list is reversed up to prev

while (current != null)

{

/\* Store next node \*/

next = current.next

/\* Change the direction of the current node \*/

current.next = prev // the linked list is reversed up to prev.next

/\* Move prev to point to the next node \*/

prev = current // the linked list is reversed up to prev && prev = current

/\* Continue the procedure for the next node \*/

current = next // I

}

// The invariant holds => current = null and because prev.next = null,

// the list is fully reversed, therefore we begin the list from prev:

myList = prev

}

def main (args: Array[String]) =

{

// (a) Here, we add each element to the head of myList

for (i <- 1 to 12) myList = new Node(i,myList)

// (c)

//reverse

println("List is "+myList.toString+".")

// >scala Question1.scala

// List is 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12.

}

}

**Question 2**

/\*\* Add the maplet name -> number to the mapping \*/

def store (name: String, number: String) =

{

val n = find(name)

if (n.next == null) // store the name at the end of the list

n.next = new Node (name, number, null)

else n.next.number = number // modify the number associated to the found name

}

**Question 3**

// Representing the phone book using a linked list with a dummy header and keeping the names in alphabetical order

// Abstraction function : book = {(n.name -> n.number) | n is in L(list.next)}, where L(a,b) = [] if a=b and L(a,b) = a : L(a.next,b), otherwise. Also, L(a) = L(a,null) as an abbreviation (from the lecture)

// DTI : L(list.next) is finite, and the names are distinct and sorted alphabetically

class LinkListHeaderBookOrd extends Book

{

private class Node (var name : String, var number : String, var next: Node)

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

// list represents the mapping composed of (n.name -> n.number) maplets,

// when n is a node reached by following 1 or more next references and

// the names in list are sorted alphabetically.

/\*\* Return the node before the one containing name.

\* Post: book = book\_0 && returns n s.t. n in L(list) &&

\* (n.next.name=name or n.next=null if no such Node exists)\*/

// Since we cannot use binary search on a linked list (we can, but it is slightly more inefficient than the usual finding method in O(n)), we will stick to a usual linear search

private def find (name: String) : Node =

{

var n = list

// Invariant: name does not appear in the nodes up to and including n; we suppose that "?" will never be introduced as a name in the phone book

// i.e., for all n1 in L(list.next, n.next), n1.name != name

while (n.next != null && n.next.name != name) n = n.next

n

}

/\*\* Is name in the book?

\* Post: book = book\_0 && returns if we found n such that n.next.name = name \*/

def isInBook(name: String): Boolean = find(name).next != null

/\*\* Return the number stored against name \*/

def recall(name: String) : String = {

val n = find(name); assert(n.next != null); n.next.number

}

/\*\* Add the maplet name -> number to the mapping maintaining the alphabetical order \*/

def store (name : String, number : String) =

{

val n = find(name) // We have n.next.name = name or n.next = null

// If the name we want to add is not in the list, we must add it in the correct place to maintain the DTI

if (n.next == null)

{

// We will search for the position of where the name should be put so that we maintain the DTI

var prev = list

var current = list.next

// We will consider that "?" is smaller than any name we would want to add

// Invariant I : name is bigger than every name up to, but not including current.name && current = prev.next

while ((current != null) && (name > current.name))

{

prev = prev.next

current = current.next

}

// From the invariant, we know that name is bigger than every name up to, but not including the current node, so we should put the name in a node that will be introduced between prev and current.

var n1 = new Node (name, number, current)

prev.next = n1

}

else n.next.number = number

// Finding the node that have node.next.name = name and then skipping node.next

def delete (name : String) : Boolean =

{

val n = find(name)

if (n.next != null) {n.next = n.next.next ; true}

else false

}

}

**Question 4**

/\*

(a)

The expected amount of work done by a recall function is E, given by the formula sum from i=0 to (n-1) of work(i) \* p(i), where work(i) is the number of operations needed to reach the ith node of the linked list, which, in our case of a linear algorithm of searching, will be (i+1) and p(i) is the probability that the ith name would be recalled. Also, we have to add to E the work needed in the case when we recall a name that doesn't exist in the list, and that is w(none) = n and the probability to recall such a name, q = 1 - (p(0) + p(1) + ... + p(n-1))

Therefore, we have E = p(0) + 2\*p(1) + 3\*p(2) + ... + (n-1)\*p(n-2) + n\*p(n-1) + n\*q, which would obviously be minimized when p(0)>=p(1)>= ... >=p(n-1).

\*/

// (b)

// The interface to the phone book

// When a name is recalled, we search for it, and then we save its data separately, create a new node that will be put at the head of the list, and then the node where we found the name will be deleted.

// Abstraction function : book = {(n.name -> n.number) | n is in L(list.next)}, where L(a,b) = [] if a=b and L(a,b) = a : L(a.next,b), otherwise. Also, L(a) = L(a,null) as an abreviation (from the lecture)

// DTI : L(list.next) is finite, the names are distinct and sorted according to the "most recently used" rule (the last recalled is at the head of the list)

class LinkedListProbabilityBook extends Book{

private var list = new LinkedListProbabilityBook.Node("?", "?", null)

private def find(name:String) : LinkedListProbabilityBook.Node = {

var n = list

while(n.next != null && n.next.name != name) n = n.next

n

}

def isInBook(name: String): Boolean = find(name).next != null

// When we recall name, we move the node which contains it to the head of the list

def recall(name: String) : String = {

val n = find(name);

require (n.next != null)

// Preserving the recalled number

val number = n.next.number

// Deleting the node from the current position

n.next = n.next.next

// Adding the node to the head of the list

list.name = name; list.number = number

list = new LinkedListProbabilityBook.Node("?", "?", list)

// Returning the desired number

return number

}

/\*\* Add the maplet name -> number to the mapping \*/

def store(name: String, number: String) = {

val n = find(name)

if(n.next == null){ // store new info in current list header

list.name = name; list.number = number

list = new LinkedListProbabilityBook.Node("?", "?", list)

}

else n.next.number = number

}

/\*\* Delete the number stored against name (if it exists);

\* return true if the name existed. \*/

def delete(name: String) : Boolean = {

val n = find(name)

if(n.next != null){ n.next = n.next.next; true }

else false

}

}

// Companion object

object LinkedListProbabilityBook{

private class Node(var name:String, var number:String, var next:Node)

}

**Question 5**

class ArrayQueue extends Queue[Int]

{

val MAX = 100 // max number of pieces of data

// The implementation using a "circular array"

// Abstraction function : queue = data [head..(out+ln)) if out+ln < MAX

// queue = data [head..MAX) ++ [0..(head+ln)%MAX) if out+ln>=MAX

// DTI : 0 <= ln <= MAX var data = new Array [Int] (MAX)

var head = 0 // where the queue begins

var ln = 0 // the length of the queue

// If ln < MAX, then we can add x in data()(head+ln)%MAX) and then increase ln by 1, but if we get to ln = MAX, then the queue is full, so we cannot add more elements to it

def enqueue (x : Int) =

{

require (ln < MAX) // or we can say require (!isFull)

data((head+ln)%MAX) = x

ln = ln + 1

}

// The head of the list is data(head) if the list is not empty, and it doesn't exist if ln = 0

def dequeue : Int =

{

require (ln > 0) // or we can say require (!isEmpty)

val result = data(head)

head = (head + 1) % MAX

ln = ln - 1

result

}

// The queue is empty if ln = 0, therefore we have no elements in the queue

def isEmpty : Boolean = (ln == 0)

// The queue is full when we get to ln = MAX, therefore we reached the maximum size allowed for the queue

def isFull : Boolean = (ln == MAX)

}

**Question 6**

class IntQueue

{

// Abstraction function : queue = L(list.next), L(null) = {}, L(x) = x.datum:L(x.next)

// DTI : L(list.next) is finite and ends in end

private type Node = IntQueue.Node

private def Node(datum:Int, next:Node) = new IntQueue.Node(datum,next)

private var list = Node(0,null)

private var end = Node(0,null)

list.next = end

// Instead of the dummy end we place the new node and we create a new dummy end afterwards, updating end

def enqueue (x : Int) =

{

end.datum = x

end.next = Node(0,null)

end = end.next

}

// First, we need that the queue is not empty, which happens when isEmpty = true. Then, if not, we keep the data of the first node after the dummy header, and then we delete it.

def dequeue : Int =

{

require (! isEmpty)

var result = list.next.datum

list.next = list.next.next

result

}

// The queue is empty if we have list.next = end

def isEmpty : Boolean = (list.next == end)

}

// Companion object

object IntQueue{

private class Node(var datum:Int, var next:Node)

}

**Question 7**

class DoubleEndedQueue

{

// Abstraction function : queue = L(list.next), L(null) = {}, L(x) = x.datum:L(x.next)

// DTI: L(list.next) is finite and ends in end (we do not count the dummy end)

private type Node = DoubleEndedQueue.Node

private def Node(datum:Int, prev:Node, next:Node) = new DoubleEndedQueue.Node(datum,prev,next)

private var list = Node (0,null,null)

private var end = Node (0,null,null)

list.next = end

end.prev = list

// state : s: seq Int

// init : s = {}

/\*\* Is the queue empty? \*/

// Post: list = list\_0 && return list.next == end

def isEmpty : Boolean = (list.next == end)

/\*\* add x to the start of the queue. \*/

// Post : list = x : list\_0

def addLeft(x:Int) =

{

list.datum = x

list.prev = Node(0,null,list)

list = list.prev

}

/\*\* get and remove element from the start of the queue. \*/

// Pre : list is non-empty

// Post : list = tail list\_0 && return head list\_0

def getLeft : Int =

{

require (! isEmpty)

var result = list.next.datum

list.next = list.next.next

list.next.prev = list

result

}

/\*\* add element to the end of the queue. \*/

// Post : list = list\_0 ++ [x]

def addRight(x: Int) =

{

end.datum = x

end.next = Node (0,end,null)

end = end.next

}

/\*\* get and remove element from the end of the queue. \*/

// Pre : list is non-empty

// Post : list = init list\_0 && return last list\_0

def getRight : Int =

{

require (! isEmpty)

var result = end.prev.datum

end.prev = end.prev.prev

end.prev.next = end

result

}

}

// Companion object

object DoubleEndedQueue{

private class Node(var datum:Int, var prev:Node, var next:Node)

}