DATA STRUCTURES AND ALGORITHMS

PROJECT NO. 5 – ADT MultiMap implemented on a singly linked list on an array

Name: Nistor Alexandra-Maria

Group: 915

1. **Problem Statement and Justification**

Your University needs an application for the final exam in the Anatomy class. The exam is made up of 10 questions with 5 possible answers. Each question is worth 10 points and has one or more correct answers. At the end of the quiz, the student can see only if he has passed the exam or not and his position among those who took the exam so far. In order to pass, a student must answer correctly to at least 5 questions. Moreover, to get the point for a question, a student must select all the right answers, neither less nor more.

I choose to use a MultiMap for this application, as in a MultiMap there are multiple keys, each one having one or more associated values. In this case, the keys are the questions and the values are the answers. The list of answers associated to each question consists of the right ones. The other possible answers are chosen randomly from the lists of answers associated to other questions.

1. **ADT Specification**

A MultiMap is a container in which the elements are <key, value> pairs. In a MultiMap a key can have one or more associated values, therefore they are not always unique. The values can be accessed only by using the keys. There are no positions in a MultiMap.

M = {m | m is a MultiMap with elements e = (k, v) where k ∈ TKey and v ∈ TValue}

1. **ADT Representation**

TElem:

key: TKey

value: TValue

SLLANode:

info: TElem

next: Integer

SLLArray:

elems: SLLANode[]

firstEmpty: Integer

head: Integer

capacity: Integer

MultiMap:

sll: SLLArray

* TKey: String
* TValue: String

1. **Interface**

Interface TElem:

init(elem, key, val)

destroy(elem)

getKey(elem)

getValue(elem)

operator==(element,element2)

operator!=(element, element2)

Interface SLLANode:

init(node, info, next)

destroy(node)

getInfo(node)

getNext(node)

setInfo(node, newInfo)

setNext(node, newNext)

Interface SLLArray:

init(array)

destroy(array)

getHead(array)

getFirstEmpty(array)

search(array, elem)

insertFirst(array, elem)

deleteElement(array, elem)

getFromPosition(array, pos) getNodeFromPosition(array,pos)

Interface MultiMap:

init(mm)

destroy(mm)

add(mm, key, value)

remove(mm, key, value)

search(mm, key)

values(mm)

keys(mm)

size(mm)

getMultiMapIterator(mm)

1. **Implementation**
2. TElem

subalgorithm init(elem, key, val):

Complexity: θ (1)

description: creates a new TElem

pre: key ∈ TKey, val ∈ TValue

post: elem ∈ TElem, elem is a TElem with the given key and value

elem.key <- key

elem.value <- val

end-subalgorithm

subalgorithm destroy(elem):

Complexity: θ (1)

description: destroys a TElem

pre: elem ∈ TElem

post: elem was destroyed

end – subalgorithm

function getKey(elem):

Complexity: θ (1)

description: gets the key of a TElem

pre: elem ∈ TElem

post: returns the key of elem

getKey <- elem.key

end - function

function getValue(elem):

Complexity: θ (1)

description: gets the value of a Telem

pre: elem ∈ TElem

post: returns the value of elem

getValue <- elem.value

end – function

function operator== (element, element2):

Complexity: θ (1)

description: compares two TElems by key and value

pre: element, element2 ∈ TElem

post: returns True if element and element2 have the same key and the same

value, False otherwise

if element.key == element2.key and element.value == element2.value then

operator== <- true

else

operator== <- false

end-function

function operator!= (element, element2):

Complexity: θ (1)

description: compares two TElems by key and value

Pre: element, element2 ∈ TElem

Post: returns True if element and element2 have different values, false otherwise

if element.value != element2.value then

operator!= <- true

else

operator!= <- false

end-function

1. SLLANode

subalgorithm init(node, info, next):

Complexity: θ (1)

description: creates a new SLLANode

pre: info ∈ TElem, next ∈ Integer

post: node ∈ SLLANode, node is a SLLANode with the corresponding info and

value of next

node.info <- info

node.next <- next

end - subalgorithm

subalgorithm destroy(node):

Complexity: θ (1)

description: destroys a SLLANode

pre: node ∈ SLLANode

post: node was destroyed

end – subalgorithm

function getInfo(node):

Complexity: θ (1)

description: gets the info corresponding to a SLLANode

pre: node ∈ SLLANode

post: returns the info of node

getInfo <- node.info

end - function

function getNext(node):

Complexity: θ (1)

description: gets the value of next corresponding to a SLLANode

pre: node ∈ SLLANode

post: returns the value of next corresponding to node

getNext <- node.next

end – function

subalgorithm setInfo(node, info):

Complexity: θ (1)

description: modifies the info corresponding to a SLLANode

pre: node ∈ SLLANode, info ∈ TElem

post: info corresponding to node was modified

node.info <- info

end – subalgorithm

subalgorithm setNext(node, next):

Complexity: θ (1)

description: modifies the value of next corresponding to a SLLANode

pre: node ∈ SLLANode, next ∈ Int

post: value of next corresponding to node was modified

node.next <- next

end – subalgorithm

function operator== (node, node2):

Complexity: θ (1)

description: compares two SLLANodes by info and next

pre: node, node2 ∈ SLLANode

post: returns True if node and node2 have the same info and the same value of next, False otherwise

if node.info == node2.info and node.next == node2.next then

operator== <- true

else

operator== <- false

end-function

1. SLLArray

subalgorithm init(sll):

Complexity: θ (n)

description: creates a new empty SLLArray

pre: True

post: sll ∈ SLLArray, sll is an empty SLLArray

sll.capacity <- 100

sll.head <- -1

for i<- 0, sll.capacity-2 execute

sll.elems[i].next <- i + 1

end - for

sll.elems[sll.capacity-1].next <- -1

sll.firstEmpty <- 0

end - subalgorithm

subalgorithm destroy(sll):

Complexity: θ (1)

description: destroys a SLLArray

pre: sll ∈ SLLArray

post: sll was destroyed

end – subalgorithm

function getHead(array):

Complexity: θ (1)

description: gets the head of a SLLArray

pre: sll ∈ SLLArray

post: returns the value of head corresponding to sll

getHead <- sll.head

end – function

function getFirstEmpty(sll):

Complexity: θ (1)

description: gets the value of firstEmpty of a SLLArray

Pre: sll ∈ SLLArray

Post: returns the value of firstEmpty corresponding to sll

getFirstEmpty <- sll.firstEmpty

end - function

function search(sll, elem):

Best Case: θ (1)

Worst Case: θ (n)

Average Case:



description: searches for the given element in the sll

pre: sll ∈ SLLArray, elem ∈ TElem

post: returns True, if elem is in sll, False otherwise

current <- sll.head

while current != -1 and sll.elems[i].info != elem execute

current <- sll.elems[current].next

end – while

if current != -1 do

return True

else

return False

end – if

end – function

subalgorithm insertFirst(sll, elem):

Complexity: θ (1)

description: adds a new element to the beginning of sll

pre: sll ∈ SLLArray, elem ∈ TElem

post: sll’ ∈ SLLArray, sll’ = sll ∪ { elem }, elem is the first element in the sll

if sll.firstEmpty = -1 then

@exception: Capacity reached its maximum

end - if

newPos <- sll.firstEmpty

sll.elems[newPos].info <- elem

sll.firstEmpty <- sll.elems[firstEmpty].next

sll.elems[newPos].next <- sll.head

sll.head <- newPos

end – subalgorithm

subalgorithm deleteElement(sll, elem):

Best Case: θ (1)

Worst Case: θ (n)

Average Case: O (n)

description: removes a given element from sll

pre: sll ∈ SLLArray, elem ∈ TElem

post: sll’ ∈ SLLArray, sll’ = sll \ { elem } ( if the element is not in the sll, no change is made)

prevNode <- -1

currentNode <- sll.head

while currentNode != -1 and sll.elems[currentNode].info != elem execute

prevNode <- currentNode

currentNode <- sll.elems[currentNode].next

end – while

if currentNode != -1 do

if currentNode == sll.head do

sll.head <- sll.elems[currentNode].next

else

sll.elems[prevNode].next<- sll.elems[currentNode].next

end – if

sll.elems[currentNode].next <- sll.firstEmpty

sll.firstEmpty <- currentNode

end – if

end – subalgorithm

function getFromPosition(sll, pos):

Complexity: θ (1)

description: gets the element on a given position in sll

pre: sll ∈ SLLArray, pos ∈ Integer

post: returns the element in the sll on the position pos

getFromPosition <- sll.elems[pos].info

end – function

function getNodeFromPosition(sll, pos):

Complexity: θ (1)

description: gets the SLLANode on a given position in sll

pre: sll ∈ SLLArray, pos ∈ Integer

post: returns the node in the sll on the position pos

getNodeFromPosition <- sll.elems[pos]

end – function

1. MultiMap

subalgorithm init(mm):

Complexity: θ (1)

description: creates a new empty MultiMap

pre: true

post: mm ∈ MultiMap, mm is an empty MultiMap

init(sll<SLLANode>)

mm.sll <- sll

end – subalgorithm

subalgorithm destroy(mm):

Complexity: θ (1)

description: destroys a MultiMap

pre: mm ∈ MultiMap

post: mm was destroyed

end – subalgorithm

subalgorithm add (mm, key, value):

Complexity: θ (1)

description: adds a new <key, value> pair to mm

pre: mm ∈ MultiMap, key ∈ TKey, value ∈ TValue

post: mm’ ∈ MultiMap, mm’ = mm ∪ <key, value>

init(elem, key, value)

insertFirst(mm.sll, elem)

end – subalgorithm

subalgorithm remove(mm, key, value):

Complexity: O (n)

description: removes a given <key, value> pair from mm

pre: mm ∈ MultiMap, key ∈ TKey, value ∈ TValue

post: mm’ ∈ MultiMap, mm’ = mm \ <key, value> (if the element having key key and value value does not exist in mm, no change is made)

init(elem, key, value)

deleteElement(mm.sll, elem)

end – subalgorithm

function search(mm, key):

Complexity: θ (n)

description: searches in mm for the values associated to the given key

pre: mm ∈ MultiMap, key ∈ TKey

post: returns a listOfStrings, containing the elements from mm having the key key

@initialize res, an empty listOfStrings

iterator <- getMultiMapIterator(mm)

while isValid(iterator) execute

elem <- getCurrent(iterator)

if getKey(elem) == key then

addString(res, getValue(elem))

end – if

next(iterator)

end – while

search <- res

end - function

function values(mm):

Complexity: θ (n)

description: searches in mm for all the values

pre: mm ∈ MultiMap

post: returns a listOfStrings, containing all the values from mm

@initialize res, an empty listOfStrings

iterator <- getMultiMapIterator(mm)

while isValid(iterator) execute

elem <- getCurrent(iterator)

addString(res, getValue(elem))

next(iterator)

end – while

values <- res

end – function

function keys(mm):

Complexity: θ (n)

description: searches in mm for all the keys

pre: mm ∈ MultiMap

post: returns a listOfStrings, containing all the keys from mm (if a key appears more than once in mm, it will be added only once in the l listOfStrings)

@initialize res, an empty listOfStrings

iterator <- getMultiMapIterator(mm)

while isValid(iterator) execute

elem <- getCurrent(iterator)

addString(res, getKey(elem))

next(iterator)

end – while

values <- res

end – function

function size(mm):

Complexity: θ (n)

description: gets the number of <key, value> pairs from mm

pre: mm ∈ MultiMap

post: returns the number of <key, value> pairs from mm

length <- 0

iterator <- getIterator(mm)

while isValid(iterator) execute

length <- length + 1

next(iterator)

end – while

size <- length

end – function

function getMultiMapIterator(mm):

Complexity: θ (1)

description: returns an iterator for a MultiMap

pre: mm ∈ MultiMap

post: iterator ∈ I, iterator is an interator over mm

init(iterator, mm.sll)

getMultiMapIterator <- iterator

end – function

1. Representation and implementation of the additional element listOfStrings

listOfStrings:

elems: string[100]

length: Integer

capacity: Integer

function createlistOfStrings(int capacity):

Complexity: θ (1)

description: creates a new empty listOfStrings

pre: capacity ∈ Integer

post: returns a listOfStrings of capacity capacity

init(list)

list.capacity <- capacity

list.length <- 0

createlistOfStrings <- list

end – function

function addString(list, str):

Complexity: θ (1)

description: adds a new string to list

pre: list ∈ listOfStrings, str ∈ String

post: list’ ∈ listOfStrings, list’ = list U { str }

list.elems[list.length] <- str

list.length <- list.length + 1

addString <- list

end – function

function getElement(list, pos):

Complexity: θ (1)

description: gets the string on position pos from list

pre: list ∈ listOfStrings, pos ∈ Integer

getElement <- list.elems[pos]

end – function

function deleteString(list, pos):

Best Case: θ (1)

Worst Case: θ (n)

Average Case: O (n)

description: removes the element on position pos from a listOfStrings

pre: list ∈ listOfStrings, pos ∈ Integer

post: list’ ∈ listOfStrings, list’ = list \ { list.elems[pos] } (if position is not valid, no change is made)

if pos < 0 || pos >= list.length then

deleteString <- list

end – if

for i <- pos, list.length – 2 execute

list.elems[i] <- list.elems[i+1]

end – for

list.length <- list.length – 1

deleteString <- list

end – function

function searchString(list, str):

Best Case: θ (1)

Worst Case: θ (n)

Average Case: O (n)

description: checks if a string exists in list

pre: list ∈ listOfStrings, str ∈ String

post: returns True if str is in list, False otherwise

for i<-0, list.length-1 execute

if getElement(list, i) == elem then

searchString <- True

end – if

end – for

searchString <- False

end – function

function equality(list1, list2):

Best Case: θ (1)

Worst Case: θ (n)

Average Case: O (n)

description: checks if two listOfStrings are identical

pre: list1 ∈ listOfStrings, list2 ∈ listOfStrings

post: returns true if list1 and list2 contain the same strings, False otherwise

if list1.length != list2.length then

equality <- False

else

for i<-0, list1.length execute

str <- getElement(list1, i)

if searchString(list2, str) == false then

equality <- False

end – if

end – for

end – if

equality <- True

end - function

1. Implementation of the quiz

function initQuiz(source)

Complexity: θ(n)

description: initializes a new MultiMap, reading data from a file

pre: source ∈ String

post: quiz ∈ MultiMap

init(quiz);

@while reading <question, answer>pairs from the file source,

(question, answer ∈ String)

add(quiz, question, answer)

initQuiz <- quiz

end - function

function initVector(vect, str)

Complexity: θ(n)

description: initialize a vector of maximum 100 elements, in which each position is equal to the value on that position

pre: vect ∈ Integer, str ∈ listOfStrings

post: returns a pointer to vect

for i <- 0, length(str) do

vect[i] <- i

end – for

initVector <- vect

end - function

function randomize(vect, str)

Complexity: θ(1)

description: initialize a vector of maximum 100 elements and then shuffles them

pre: vect ∈ Integer, str ∈ listOfStrings

post: returns a pointer to vect

initVect(vect, str)

@shuffle the elements from vect

randomize <- vect

end – function

function removeRightAnswers(right, wrong)

Complexity: θ(n)

description: removes from wrong all the common elements of right and wrong

pre: right ∈ listOfStrings wrong ∈ listOfStrings

post: wrong’ ∈ listOfStrings, wrong’ = wrong \ {common elements with right} (if they don’t have any common elements, no change is made)

for i<- 0, length(wrong) execute:

elem <- getElement(wrong, i);

if searchString(right, elem) == true then

wrong <- deleteString(wrong, i)

i -= 1

end – if

end – for

removeRightAnswers <- wrong

end – function

function completeAnswersList(complete, wrong, nr, vect)

Complexity: θ(n)

description: adds to complete all the elements of wrong

pre: complete ∈ listOfStrings, wrong ∈ listOfStrings, nr ∈ Integer, vect ∈ Integer

post: complete’ ∈ listOfStrings, complete’ = complete ∪ {elements from right}

for i <- 0, nr-1 execute

elem <- getElement(wrong, vect[i])

complete <- addString(complete, elem)

end – for

completeAnswersList <- complete

end - function

subalgorithm printQuestion(i, key, current, vector)

Complexity: θ(1)

description: prints the current question and its possible answers to the user

pre: i ∈ Integer, key ∈ String, current ∈ listOfStrings, vector ∈ Integer

post: True

@print question

for i <- 0, 4 do

elem <- getElement(current, vect[i])

@print elem

end – for

end - subalgorithm

function userAnswer(current, vect)

Complexity: θ(n)

description: gets the answer from the user and return the corresponding list of strings

pre: current ∈ listOfStrings, vect ∈ Int

post: userFinal ∈ listOfStrings

@get the user’s answer

userFinal <- createlistOfStrings(6)

@adds to userFinal, the answers corresponding to the digits entered by the user

userAnswer <- userFinal

end - function

subalgorithm decide(user, right)

Complexity: θ(1)

description: tells the user if he chose the right answer (or answers) or not

pre: user ∈ listOfStrings, right ∈ listOfStrings

post: True

if equality(user, right) then

@print “Correct! + 10points” and adds 10 points to the total score

else

@print “Auch.Wrong answer”

end – if

end – subalgorithm

function startQuiz(quiz)

Complexity: θ(n)

description: shows 10 questions to the user, gets his answers, checks if they are right and computes his total score

pre: quiz ∈ MultiMap

post: return the total score of the user

init(keys)

init(currentAnswers)

init(RAnswers)

init(currentWAnswers)

randomize(vectK, keys)

for i <- 0, 9 execute

currentKey <- getElement(keys, vectK[i])

currentAnswers <- search(quiz, currentKey)

RAnswers <- currentAnswers

currentWAnswers <- values(quiz)

removeRightAnswers(currentAnswers, currentWAnswers)

randomize(vectWA, currentWAnswers)

nr < - 5 – length(currentAnswers)

currentAnswers <- completeAnswersList(currentAnswers, currentWAnswers, nr, vectWA)

randomize(vectRA, currentAnswers)

printQuestion(i, currentKey, currentAnswers, vectRA)

init(userFinal)

userFinal = userAnswer(currentAnswers, vectRA)

decide(userFinal, RAnswers)

end – if

startQuiz <- totalScore

end - function

subalgorithm seeTop(source, totalScore)

Complexity: θ(n)

description: shows to the user its position in the top

pre: source ∈ String, totalScore ∈ Integer

post: True

@reads data from the file containg the top of people who took the quiz and counts the number of people who got a higher score than the user

@ prints the position of the user

end – subalgorithm

subalgorithm addScore(source, totalScore, name)

Complexity: θ(1)

description: adds the score and the name of the user to the top

pre: source ∈ String, totalScore ∈ Integer, name ∈ String

post: True

@write to the file name and totalScore

end – subalgorithm

function main()

description: asks for the users name, starts the quiz, tells the user if he passed or not and allows him to see his position in the top if he did pass

pre: True

post: True

@ask for user’s name

quiz <- initQuiz(“inputData.txt”)

score <- startQuiz(quiz)

while true execute

if score >= 50 then

@shows the user his score, allows him to exit the quiz or to see his position in the top

else

@tells the user he failed

end - if

end – while

end - function

1. **Iterator**

An iterator is a structure that is used to iterate through the elements of a container, in our case through the elements of a MultiMap.

I = {it | it is an interator over a MultiMap with elements e = (k, v) where k ∈ TKey and v ∈ TValue}

**IteratorMultiMap:**

multiMap: MultiMap

currentElement: Integer

**Interface:**

init(it, sll)

destroy(it)

getCurrent(it, e)

valid(it)

next(it)

**Implementation:**

subalgorithm init(it, sll):

Complexity: θ (1)

description: creates an interator over sll

pre: sll ∈ SLLArray

post: it ∈ I, it is in iterator over the sll

it.sll <- sll

it.currentPosition <- getHead(sll)

end – subalgorithm

subalgorithm destroy(it):

Complexity: θ (1)

description: destroys a given iterator

pre: it ∈ I

post: it was destroyed

function getCurrent(it):

Complexity: θ (1)

description: gets the current node in the SLLArray

pre: it ∈ I

post: returns the current SLLANode in the sll

getCurrent<- getFromPos(sll.it, sll.currentPos)

end – function

function valid(it):

Complexity: θ (1)

description: checks if the iterator is valid

pre: it ∈ I

post: returns True if the iterator is valid, False otherwise

if it.currentPosition != -1 then

return True

else

return False

end – if

end – function

subalgorithm next(it):

Complexity: θ (1)

description: moves the iterator to the next node in sll

pre: it ∈ I

post: it’ ∈ I, it’ points to the next node in sll

node <- getNodeFromPosition(it.sll, it.currentPosition)

it.currentPosition <- getNext(node)

end – subalgorithm

1. **Tests**

class Tests {

public:

Tests() {}

~Tests() {}

void testSllArray();

void testMultiMap();

};

void Tests::testSllArray() {

SLLArray sll;

assert(sll.getFirstEmpty() == 0);

assert(sll.getHead() == -1);

sll.insertFirst(TElem("question1", "answ1"));

sll.insertFirst(TElem("question2", "answ2"));

sll.insertFirst(TElem("question3", "answ3"));

assert(sll.getHead() == 2);

assert(sll.getFirstEmpty() == 3);

assert(sll.getFromPosition(1) == TElem("question2", "answ2"));

assert(sll.search(TElem("question1", "answ1")) == true);

sll.deleteElement(TElem("question1", "answ1"));

assert(sll.search(TElem("question1", "answ1")) == false);

sll.deleteElement(TElem("question3", "answ3"));

assert(sll.getHead() == 1);

}

void Tests::testMultiMap() {

MultiMap mm;

mm.add("question1", "answ1");

mm.add("question2", "answ2");

mm.add("question3", "answ3");

assert(mm.size() == 3);

MultiMapIterator it = mm.getMultiMapIterator();

assert(it.getCurrent().getValue() == "answ3");

assert(it.getCurrent().getKey() == "question3");

TElem elem1 = it.getCurrent();

it.next();

assert(it.getCurrent().getValue() == "answ2");

assert(it.valid() == true);

TElem elem2 = it.getCurrent();

assert(elem1.operator==(elem2) == false);

it.next();

it.next();

assert(it.valid() == false);

assert(mm.keys().length == 3);

mm.remove("question2", "answ2");

assert(mm.values().length == 2);

assert(mm.size() == 2);

assert(mm.search("question1").length == 1);

listOfStrings l1 = mm.search("question1");

mm.add("question4", "answ1");

listOfStrings l2 = mm.search("question4");

assert(equality(l1, l2) == true);

l1 = deleteString(l1, 0);

assert(l1.length == 0);

}