Attendance

Cioroga Rares	Caravia Andrei
Chis Sergiu	Comănac Dragoș-Mihail
Capra Paul Ovidiu	Custura Stefan - Octavian
Deiac David-Mihai	Colta Paul-Stefan
Cosma Eduard	Copîndean Alexandru
Craciun Ioan-Flaviu	Crețа Florin
Cheran Bianca Paula	Carare Claudiu
Ciorbă Rareș-Nicolaie	Craiu Constantin-Tiberiu
Darida Razvan	Curila Sebastian-Bernard
Curbăt Alexandra	
Chis Matei	
Condrea Adrian	
Cimpean Andreea	
Clapou Alexandru	
Comsa Filip-Emanuel	
Cirtorosan Dragos	
Ciupe Sergiu-Călin	
Demian Ana-Maria	

ADT SortedMultiMap (SMM)

map - It contains key-value pairs. The keys have to be unique. The pairs do not have to be in any particular order (there are no positions)
multimap - a key can have several values (key can have a list of values)
sortedmultimap - the keys are sorted based on a relation R

Interface of the SMM
init(smm, Relation)
add(smm, key, value)
remove(smm, key, value)
search(smm, key) -> returns the list of all values associated to the key
size(smm)
isEmpty(smm)
iterator(smm)
destroy(smm)
Other possible operations
keys(smm) -> returns a sortedset of all the keys
values(smm) -> returns a bag of all the values
pairs(smm) -> returns a bag of all the pairs

Problem: Implement ADT SortedMultiMap - using a singly linked list with dynamic allocation

Ex: SMM with the translation of different English words in Romanian, ordered alphabetically
book - carte, a rezerva, publicatie
red - rosu
blood - sange, neam

Representation1: a singly linked list with <key, value> pairs.
Representation2: a singly linked list with unique keys and list of values.

Representation:

SortedMultiMap:

head: ↑Node
size: Integer
rel: Relation

Node:

next: ↑Node
info: TElem

TElem:

key: TKey
vl: List

k1 < k2
rel (k1, k2) - a function with 2 parameters (two keys)

- it returns true if “k1 <= k2” (k1 should be in front of k2 if we sort them, k1 and k2 are in the correct order)
- return false if k1 and k2 should be swapped

if k1 <= k2 then…
if rel(k1, k2) = true then …

Iterator

- sortedmultimap
- current key - pointer to a node
- current value -
    - index of the current value
    - iterator over the list of values - all operations run in Theta(1)
        - ADT List for the values - getElement (index)
            - dynamic array - Theta(1)
            - linked list - O(n)

IteratorSMM:

smm: SMM
currentKey: ↑Node
itL: IteratorList

```
subalgorithm init(smmit, smm) is:
    smmit.smm <- smm
    smmit.currentKey <- smm.head
    if  smm.head != NIL then
        iterator([smm.head].info.vl , smmit.itL) //function from the interface of the list
    end-if
end-subalgorithm //Theta(1)

subalgorithm next(smmit) is:
    if currentKey = NIL then
        @throw an exception
    end-if
    next(smmit.itL)
    if valid(smmit.itL) = false then
        currentKey <- [currentKey].next
        if currentKey != NIL then
            iterator([currentKey].info.vl, smmit.itL)
        end-if
    end-if
end-subalgorithm // Theta(1)

function getCurrent(smmit) is:
    If currentKey = NIL then
        @throw an exception
    End-if
    Key <- [currentKey].info.key
    Value <- getCurrent(smmit.itL)
    getCurrent <- <key, value>
End-function //Theta(1)

function valid(smmit) is:
    if currentKey = NIL then
        valid <- false
    else
        valid <- true
    end-if
end-function //Theta(1)
```

```
subalgorithm init(smm, R) is:
    smm.rel <- R
    smm.head <- NIL
    smm.size <- 0 //the number of pairs
end-subalgorithm // Theta(1)
```

```
subalgorithm destroy(smm) is:
    while smm.head != NIL execute
        current <- smm.head
        smm.head <- [smm.head].next
        destroy([current].info.vl) //destructor
        free(current) // delete[]
    end-while
end-subalgorithm
```

Complexity:
n - nr of unique keys
smm - total number of elements
Theta(n) - if destroy is Theta(1)
Theta(smm) - if destroy is Theta(nr of values)

search(smm, key)

- find a node with key and return the list from it (or return empty list)

add(smm, key, value)

- find a node with key
  - if there is such a node, add the value to the list
  - if there is no such node, add a new node (we need the previous one)

remove(smm, key, value)

- find a node with key
  - if there is no such node, we are done
  - if there is such a node, remove value from the value list
  - if the value list is empty, remove the node (we need the previous one)

auxiliary function to find a node with a given key and the previous node

```
subalgorithm searchNode(smm, key, kNode, prevNode) is:
//searchNode for “book”, kNode = book, prevNode = blood
//searchNode for “blood”, kNode = blood, prevNode = NIL
//searchNode for “day”, kNode = NIL, prevNode = book
//searchNode for “air”, kNode = NIL, prevNode = NIL
    aux <- smm.head
    prev <- prev
    found <- false
    while aux != NIL and found = false and smm.rel([aux].info.key, key) execute
        if [aux].info.key = key then
            found <- true
        else
            prev <- aux
            aux <- [aux].next
        end-if
    end-while
    if found then
        kNode <- aux
        prevNode <- prev
    else
        kNode <- NIL
        prevNode <- prev
    end-if
end-subalgorithm //O (n)
```