DSA – Seminar 3
Sorted MultiMap (SMM)

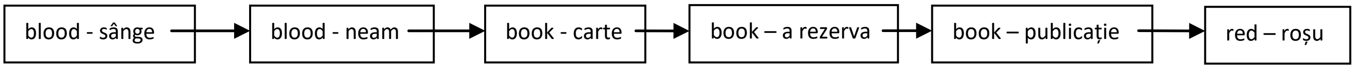
- Map – contains key-value pairs. Keys are unique, each key has a single associated value.
- MultiMap – a key can have multiple associated values (a list of values).
- Sorted MultiMap – there is a relation R defined on the keys and they are ordered based on the keys.

Interface of a SMM

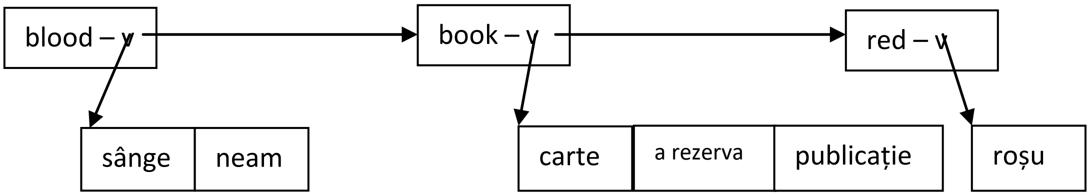
Problem: Implement the SortedMultiMap ADT – use a sinly linked representation with dynamic allocation

Ex. a multimap with the translation of different English words in Romanian
- book – carte, a rezerva, publicație
- red – roșu
- blood – sânge, neam

Representation 1: Singly linked list of <key, value> pairs. There might be multiple nodes with the same key, they will be placed one after the other (since the nodes are sorted based on the keys).



Representation 2: Singly linked list of <key, list of values> pairs. The keys are unique and sorted.



<u>TElem:</u>
k: TKey
vl: List

<u>Node:</u>
info: TElem
next: ↑Node

<u>SMM:</u>
head: ↑Node
R: Relation

$$R(k_1, k_2) = \begin{cases} true, if \text{ "}k_1 < k_2\text{" } (k_1 comes\ before\ k_2) \\ false, otherwise \end{cases}$$

<u>Iterator:</u>
We need to keep in the iterator:
- the SMM
- a reference to the current node from the SMM

- an iterator for the list of values associated to the current node

  IteratorSMM:
  smm: SMM
  current: ↑Node
  itL: IteratorList

Iterator operations: init, valid, next, getCurrent (returns a <key, value> pair).

Printing the elements of a SMM using the iterator:

```
Subalgorithm print(smm) is:
      iterator(smm, it)
      while valid(it) execute:
            getCurrent(it, <k,v>)
            @print c and v
            next(i)
      end-while
end-subalgorithm
```

The print subalgorithm looks in the same way independently of the representation of the iterator and the representation of the map!

Operations for the iterator

```
subalgorithm init (it, smm) is:
      it.smm ← smm
      it.current ← smm.head
      if it.current ≠ NIL then:
            iterator([it.smm.head].info.vl, it.itL)
      end-if
end-subalgorithm
Complexity: θ(1)


subalgorithm getCurrent(it, e) is: // e will be a <k, v> pair
      k ← [it.current].info.k
      getCurrent(it.itL, v)
      e ← <k,v>
end-subalgorithm
Complexity: θ(1)


function valid(it):
      if it.current ≠ NIL then
            valid ← true
      else
            valid ← false
end-function
Complexity: θ(1)
```

```
subalgorithm next(it) is:
      next(it.itL)
      if not valid(it.itL) then
            it.current ← [it.current].next
            if it.current ≠ NIL then
                  iterator ([it.current].info.vl, it.itL)
            end-if
      end-if
end-subalgorithm
Complexity: $\theta(1)$
```

Operations for the sorted multi map

Notations for the complexities:

    n – number of distinct keys
    smm – total number of elements

```
subalgorithm init(smm, R) is:
      smm.R ← R
      smm.head ← NIL
end-subalgorithm
Complexity: $\theta(1)$
```

```
subalgorithm destroy(smm) is:
      while smm.head ≠ NIL execute:
            aux ← smm.head
            smm.head ← [smm.head].next
            destroy([aux].info.vl)
            free(aux)
      end-while
end-subalgorithm
Complexity: $\theta(smm)$ (or $\theta(n)$ – if the lists for the values do not need to be
destroyed)
```

```
//auxiliary function that will help us with the other operations (private function,
it is not part of the interface).
//pre: smm is  SMM, k is a Tkey
//post: searchNode returns the address of the node that contains k as key, or NIL if
no key with k exists.
function searchNode(smm, k) is:
    aux ← smm.head
    found ← false
    while aux ≠ NIL and smm.R(k, [aux].info.k) and  not found execute
        if [aux].info.k = k then
            found ← true
        else
            aux ← [aux].next
        end-if
    end-while
    if found then
        searchNode ← aux
    else
        searchNode ← NIL
    end-if
end-function
Complexity: O(n)

subalgorithm search(smm, k, list) is:
    aux ← searchNode (smm, k)
    if aux = NIL then
        init(list) // return an empty list
    else
        list ← [aux].info.vl
    end-if
end-subalgorithm
Complexity: O(n)


subalgorithm add(smm, k, v) is:
    aux ← searchNode(smm, k)
    if aux = NIL then
        addANewKey (smm, k, v)
    else
        if search([aux].info.vl, v) = false then
            addEnd ([aux].info.vl, v)
        end-if
    end-if
end-subalgorithm
Complexity: O(smm)
//searchNode, addANewKey are Θ(n) operations
//instead of addEnd another add function can be used (so it can have Θ(1) complexity)
//search is linear with the length of the value list.
```

```
//auxiliary operation (not part of interface)
//pre: smm is a SMM, k is a TKey, v is a TElem/ Tvalue
//post: a new node with key k and value v is added to the smm. The order of the keys will respect the relation.
subalgorithm addANewKey (smm, k, v) is:
    if smm.head = NIL then
        allocate (smm.head)
        [smm.head].info.k ← k
        init ([smm.head].info.vl)
        addEnd ([smm.head].info.vl, v)
    else
        c ← smm.head
        allocate(aux)
        [aux].info.k ← k
        init ([aux].info.vl)
        addEnd([aux].info.vl, v)
        if smm.R(k, [smm.head].info.k) then
            [aux].next ← smm.head
            smm.head ← aux
        else
            while [c].next ≠ NIL and not smm.R(k, [[c].next].info.k) execute
                c ← [c].next
            end-while
            [aux].next ← [c].next
            [c].next ← aux
        end-if
    end-if
end-subalgorithm
```

Complexity: O(n) //supposing addToEnd it θ(1) – which is true since in this situation we will always add an element into an empty list

```
subalgorithm remove(smm, k, v) is:
    aux ← searchNode(smm, k)
    if aux ≠ NIL then
        pos ← indexOf([aux].info.vl, v)
        if pos ≠ -1 then
            remove([aux].info.vl, pos, e)
        end-if
        if isEmpty([aux].info.vl) then
            removeKey(smm, k)
        end-if
    end-if
end-subalgorithm
```

Complexity: O(smm)

```
//auxiliary operation (not part of the interface)
//pre: smm is a SMM, k is a TKey, smm contains a node with key k
//post: the node containing key k is removed from smm
subalgorithm removeKey(smm, k) is:
      if [smm.head].info.k = k then
            deleted ← smm.head
            smm.head ← [smm.head].next
            destroy([deleted].info.vl)
            free(deleted)
      else
            aux ← smm.head
            while  [[aux].next].info.k ≠ k execute
                  aux ← [aux].next
            end-while
            deleted ← [aux].next
            [aux].next ← [[aux].next].next
            destroy([deleted].info.vl)
            free(deleted)
      end-if
end-subalgorithm
Complexity: O(smm)
```