DSA – Seminar 4
Sorted MultiMap (SMM)

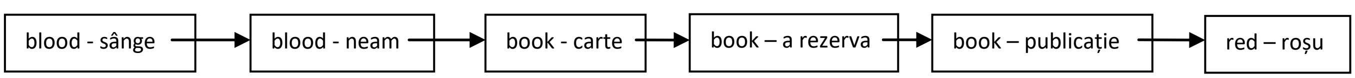
- Map – contains key-value pairs. Keys are unique, each key has a single associated value.
- MultiMap – a key can have multiple associated values (can be considered a list of values).
- Sorted MultiMap – there is a relation R defined on the keys and they are ordered based on the keys. There is no particular order of the values belonging to a key (we do not order based on the values)

Interface of a SMM

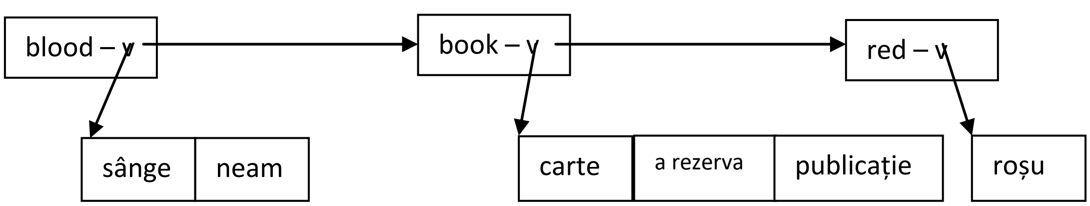
Problem: Implement the SortedMultiMap ADT – use a singly linked representation with dynamic allocation

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

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

<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

<u>IteratorSMM:</u>

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 k and v
		next(ii)
	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!

<u>Operations for the iterator</u>

```
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
	if it.current = NIL then
		@throw exception
	end-if
	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:
	if it.current = NIL then
		@throw exception
	end-if
	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: θ(1)

subalgorithm first(it) is:
	it.current ← it.smm.head
	if it.current ≠ NIL then:
		iterator([it.smm.head].info.vl, it.itL)
	end-if
end-subalgorithm
Complexity: θ(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: θ(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:
If destroy for list is θ(1) => θ(n)
If destroy for list is θ(length of list) => θ(smm)
```

//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: kNode is a ↑Node, prevNode is a ↑Node. If there is a node with k as key, kNode will be that node and prevNode will be the previous node. If there is no node with k as key, kNode will be NIL and prevNode will be the node after which the key k should be.

For the previous example (the one with the words and translations):
searchNode for „book" -> kNode the node with book, prevNode the node with blood
searchNode for „blood" -> kNode the node with blood, prevNode will be NIL
searchNode for „day" -> kNode will be NIL, prevNode the node with book
searchNode for „air" -> kNode will be NIL, prevNode will be NIL

```
subalgorithm searchNode(smm, k, kNode, prevNode) is:
	aux ← smm.head
	prev ← NIL
	found ← false
	while aux ≠ NIL and smm.R([aux].info.k, k) and  not found execute
		if [aux].info.k = k 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
Complexity: O(n)

subalgorithm search(smm, k, list) is:
	searchNode (smm, k, kNode, prevNode)
	if kNode = 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:
	searchNode(smm, k, kNode, prevNode)
	if kNode = NIL then
		addANewKey (smm, k, v, prevNode)
	else
		addEnd([kNode].info.vl, v)
	end-if
end-subalgorithm
```

```
Complexity:
//searchNode is O(n)
//addANewKey is Θ(1) operation (we will use the prevNode)
//instead of addEnd another add function can be used (so it can have Θ(1) complexity)
If addEnd (or whatever function is used for values) is Θ(1) => O(n)
If addEnd (or whatever function is used for values) is Θ(length of the list) =>
O(smm)

//auxiliary operation (not part of interface)
//pre: smm is a SMM, k is a TKey, v is a TElem/ TValue,  prevNode is a ↑Node (the
node after which the new node should be added)
//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, prevNode) is:
	allocate(newNode)
	[newNode].info.k ← k
	init ([newNode].info.vl)
	addEnd([newNode].info.vl, v)
	if prevNode = NIL then
		[newNode].next ← smm.head
		smm.head ← newNode
	else
		[newNode].next ← [prevNode].next
		[prevNode].next ← newNode
	end-if
end-subalgorithm
Complexity: Θ (1) //supposing addToEnd it Θ(1) – which is true since in this
situation we will always add an element into an empty list


function remove(smm, k, v) is:
	searchNode(smm, k, kNode, prevNode)
	if kNode ≠ NIL then
		pos ← indexOf([kNode].info.vl, v)
		if pos ≠ -1 then
			remove([kNode].info.vl, pos, e)
		end-if
		if isEmpty([kNode].info.vl) then
			removeKey(smm, k, prevNode)
		end-if
		remove ← true
	else
		remove ← false
	end-if
end-subalgorithm
Complexity: O(smm)
```

//auxiliary operation (not part of the interface)
//pre: smm is a SMM, k is a TKey, prevNode is a ↑Node, smm contains a node with key k after the node prevNode (if prevNode is NIL, then the first node if smm contains the key k). The value list of the node with key k is empty.
//post: the node containing key k is removed from smm

```
subalgorithm removeKey(smm, k, prevNode) is:
	if prevNode = NIL then
		deleted ← smm.head
		smm.head ← [smm.head].next
		destroy([deleted].info.vl)
		free(deleted)
	else
		deleted ← [prevNode].next
		[prevNode].next ← [[prevNode].next].next
		destroy([deleted].info.vl)
		free(deleted)
	end-if
end-subalgorithm
```

Complexity: Θ(1)
Destroy will destroy an empty list => Θ(1)