ADT SparseMatrix - representation using triples (value ≠ 0). Implementation on a doubly linked list with dynamic allocation.

ADT SparseMatrix:

The ADT Matrix is a container that represents a two-dimensional array. Each element has a unique position, determined by two indexes: its line and column.

If the Matrix contains many values of 0 (or 0TElem), we have a **sparse matrix**, where it is more (space) efficient to memorize only the elements that are different from 0.

Domain

MAT = {mat|mat is a matrix with elements of the type TElem}

Interface.

- init(mat, nrL, nrC):
  - **descr:** creates a new matrix with a given number of lines and columns
  - **pre:** nrL $\in N*$ and nrC $\in N*$
  - **post:** mat $\in$ MAT , mat is a matrix with nrL lines and nrC columns
  - **throws:** an exception if nrL or nrC is negative or zero

- nrLines(mat):
  - **descr**: returns the number of lines of the matrix
  - **pre**: mat $\in$ MAT
  - **post**: nrLines $\leftarrow$ returns the number of lines from mat

- nrCols(mat):
  - **descr**: returns the number of columns of the matrix
  - **pre**: mat $\in$ MAT
  - **post**: nrCols $\leftarrow$ returns the number of columns from mat

- element(mat, i, j):
  - **descr**: returns the element from a given position from the matrix (assume 1-based indexing)
  - **pre**: mat $\in$ MAT , $1 \leq i \leq$ nrLines, $1 \leq j \leq$ nrColumns
  - **post**: element $\leftarrow$ the element from line i and column j
  - **throws**: an exception if the position (i, j) is not valid (less than 1 or greater than nrLines/nrColumns)

- modify(mat, i, j, val):
  - **descr**: sets the element from a given position to a given value (assume 1-based indexing)
  - **pre**: mat $\in$ MAT , $1 \leq i \leq$ nrLines, $1 \leq j \leq$ nrColumns, val $\in$ TElem
  - **post**: the value from position (i, j) is set to val. modify $\leftarrow$ the old value from position (i, j)
  - **throws**: an exception if position (i, j) is not valid (less than 1 or greater than nrLine/nrColumns)

- iterator(mat, it):
  - **descr**: returns an iterator over the sparse matrix
  - **pre**: mat $\in$ MAT
  - **post**: it $\in$ I, it is an iterator over mat

ADT SparseMatrix - operation implementation

subalgorithm init(mat, nrL, nrC) is:
mat.head ← NULL
mat.tail ← NULL
mat.nrL ← nrL
mat.nrC ← nrC
Complexity : θ(1)

function nrLines(mat) is:
nrLines <- mat.nrL
Complexity : θ(1)

function nrColumns(mat) is:
nrColumns <- mat.nrC
Complexity : θ(1)

function element(mat, i, j) is:
if i < 1 OR j < 1 OR i > mat.nrLines OR j > mat.nrColumns **then**
@throw exception
end-if
current ← mat.head
//look for the element. Stop if we passed the position where it should be
while current != NULL AND ([current].line < i OR ([current].line = i AND [current].column < j)) **execute**
current ← [current].next
end-while
//check if we found the element
if current != NULL AND [current].line = i AND [current].column = j **then**

```
            element ← [current].info
        else
            element ← NULL TVALUE


        end-if
    End-function
```

Complexity:

Best case: the current element is NIL => : $\theta(1)$

Worst case: the element is on the last position =>O(number of non-zero elements)

Average case: O(number of non-zero elements)

```
    function modify(mat, i, j, value) is:
        if i < 1 OR j < 1 OR i > mat.nrLines OR j > mat.nrColumns then
                @throw exception
        end-if
                    current ← mat.head

        while current != NULL AND ([current].line < i OR ([current].line
    = i AND [current].column < j)) execute
                current ← [current].next
        end-while

        if current != NULL AND [current].line = i AND [current].column =
j then
            old ← [current].info
            [current].info ← value
            return old

        else
            modify ← NULL TVALUE
        end-if
```

end-function

Complexity:

Best case: the head is NIL => : θ(1)

Worst case: the element is on the last position =>O(number of non-zero elements)

Average case: O(number of non-zero elements)

Iterator domain

I = { it | it is an iterator over a sparse matrix }

Iterator interface

- init(it, mat):
  - **descr**: creates a new iterator for a multimap
  - **pre**: mat is Sparse Matrix
  - **post**: it ∈ I
- getCurrent(it):
  - **descr**: returns the current element from the iterator
  - **pre**: it ∈ I, it is valid
  - **post**: getCurrent ←<i, j ,v> , i is a line(int), j is a column(int), v is a TValue
  - **throws**: an exception if it is invalid
- next(it):
  - **descr**: moves the current element from the container to the next triple or makes the iterator invalid if no triples are left
  - **pre**: it ∈ I, it is valid

- **post**: the current element from it points to the next element from the sparse matrix
- **throws**: an exception if it is invalid

- valid(it):
  - descr: verifies if the iterator is valid
  - pre: it ∈ I
  - post: valid ←
    - True, if it points to a valid element from the container
    - False, otherwise

Sparse Matrix representation

Node:

info: TElem
line: Integer
column: Integer
next: ↑ Node
prev: ↑ Node

Sparse Matrix:

head: ↑ Node
tail:↑ Node
nrLines: Integer

nrColumns: Integer

Iterator representation

Iterator:

mat: SparseMatrix

currentElement: ↑ Node

Problem Statement:

A hotel has a huge parking lot. This parking lot works in the following way: a client checks in, the valet takes the car and parks it wherever he finds a free spot, and assigns to that position the license plate number . When the client wants to check out, he goes to the reception and asks the receptionist where the car with his unique license plate is. Now, the receptionist needs this awesome application that lets her iterate only through the occupied parking spots, until she finds the client's car

Justification:

A SparseMatrix is a smart choice, because this parking lot is set out like a matrix, with lines , columns and a value: the license plate number. Also, we are only interested in the occupied positions in order to find a car fast and save memory.