# Project Documentation

class: Data Structures and Algorithms

## Content:

# A. Container of the abstract data type

## 1. Specification and interface

### Iterator

Domain:

I ={it | it is an iterator over a container with elements of type TElement }

Interface:

- init (it, sm)

  description: creates a new iterator for a container

  precondition: sm is a Sparse Matrix

  postcondition: it ∈ I and it points to the first element in sm if sm is not empty or it is not valid

- getCurrent (it, e)

  description: return the current element from the iterator

  precondition: it ∈ I, it is valid

  postcondition: e ∈ TElement, e is the current element from it

- next (it)

  description: moves the current element from the container to the next element or makes the iterator invalid if no elements are left.

  precondition: it ∈ I, it is valid

  postcondition: the current element from it points to the next element from the sparse matrix

- valid (it)

  description: verifies if the iterator is valid

  precondition: it ∈ I

  postcondition: will return true if it points to a valid element of the sparse matrix and false otherwise.

## Sparse Matrix

Domain:

SM = {sm | sm is a sparse matrix with elements e = (l, c, v), where l ∈ TValue, c ∈ TValue and v ∈ TValue}

Interface:

- init (sm, noL, noC)

  description: creates a new empty Sparse Matrix

  precondition: noL ∈ TValue, noC ∈ TValue

  postcondition: sm ∈ SM, sm is an empty sparse matrix

- destroy (sm)

  description: destroys a sparse matrix

  precondition: sm ∈ SM

  postcondition: sm was destroyed

- noLines (sm)

  description: the function will return the number of lines in the sparse matrix

  precondition: sm ∈ SM

  postcondition: the function will return a TValue representing the number of the lines

- noColumns (sm)

  description: the function will return the number of columns in the sparse matrix

  precondition: sm ∈ SM

  postcondition: the function will return a TValue representing the number of the columns

- element (sm, line, column)

  description: the function will return the element on the position (line, column)

  precondition: sm ∈ SM, line ∈ TValue, column ∈ TValue

  postcondition: v ∈ TValue, where v = v', if there exist a tuple <line, column, v'> and the value "0" otherwise

- modify (sm, line, column, value)

  description: change the values of the element from line line and column column into value

  precondition: sm ∈ SM, line ∈ TValue, column ∈ TValue and value ∈ TValue

  postcondition: sm' ∈ SM, if value = 0 then sm' = sm - <line, column, value>; if value != 0 then sm' = sm ∪ <line, column, value>; if old value on position <line, column> != 0 and value != 0 then old value = value

For the interface some of the function I used in the interface are private so I will enumerate then below:

- relation (a, b)

  description: The function is the relation that we use at the container

  precondition: a ∈ TElement, b ∈ TElement

  postcondition: The function will return true is a>b or

  false otherwise

- add (sm, n, a)

  description: The function will actually add a pair to the sparse matrix

  precondition: sm ∈ SM, n ∈ Node ,a ∈ TElement

  postcondition: sm′∈ M, sm′= sm ∪ < line, column, value >

- minimum (sm, n)

  description: The function will return a not which is minimum after below the node n

  precondition: sm ∈ SM, n ∈ Node

  postcondition: minimum ← the minimum value from the tree below n

- remove (sm, n, line)

  description: The function will actually remove a pair from the sparse matrix

  precondition: sm ∈ SM, n ∈ Node ,k ∈ TValue

  postcondition: sm′∈ SM, sm′= sm \ < line, column, value >

- update (sm, line, column, value)

  description: the function that will actually change the value of a triple in the sparse matrix

  precondition: sm ∈ SM, line ∈ TValue, column ∈ TValue, value ∈ TValue

  postcondition: sm' ∈ SM, sm'= sm <line, column, value>

## 2. Representation of the abstract data type

### Representation

**TElement:**

line: integer

column: integer

value: string

**Node:**

info: TElement

left: ↑ Node

right: ↑ Node

**SparseMatrix:**

root: ↑ Node

noLines: integer

noColumns: integer

**Iterator:**

sm: ↑ SparseMatrix

s: stack<Node*>

currentNode: ↑Node

## Pseudocode for every function in the iterator:

```
subalgorithm init (it, sm) is:
    [it].sm ← sm
    @allocate Node nod
    nod ← [it].[sm].root
    while (not nod = NILL) execute
        [it].s.push(nod)
        nod ← [nod].left
    end - while
    if (not [it].s.empty()) then
        [it].currentNode ← [it].s.top()
    else
        [it].currentNode ← NILL
    end - if
end - subalgorithm
```

```
function getCurrent(it) is:
    getCurrent ← [it].[currentNode].info;
end - function
```

```
subalgorithm next(it) is:
    @allocate Node nod
    nod ← [it].s.top()
    [it].s.pop()
    if (not [nod].right = NILL) then
        nod ← [nod].right
        while (not nod = NIL) execute
            [it].s.push(nod)
            nod ← [nod].left
        end - while
    end - if
```

```
    if (not [it].s.empty()) then
        [it].currentNode ← [it].s.top()
    else
    [it].currentNode ← NILL
    end - if
end - subalgorithm
```

- **function** valid(it) is:

```
    if ([it].currentNode = NILL) then
        valid ← false
    else
        valid ← true
    end - if
end - function
```

## Pseudocode for every function in the container:

- **subalgorithm** init(sm, noLines, noColumns) is:

```
    [sm].root ← NILL
    [sm].noC ← noColumns
    [sm].noL ← noLines
end - subalgorithm
```

- **function** noLines (sm) is:

```
    noLines ← [sm].noL
end - function
```

- **function** noColumns (sm) is:

```
    noColumns ← [sm].noC
end - function
```

- **function** element (sm, line, column) is:

```
@ allocate TElement something
something.line ← line
something.column ← column
something.value ← "0"
@allocate node currentNode
currentNode ← sm.root
while (not [currentNode] = NIL and not [[currentNode].info].line = line )
        if (not relation([currentNode].info, something))
                currentNode ← [currentNode].right
        else
                currentNode ← [currentNode].left
        end - if
end - while
while (not [currentNode] = NIL and not [[currentNode].info].column
                                                        = column )
        if (not relation([currentNode].info, something))
                currentNode ← [currentNode].right
        else
                currentNode ← [currentNode].left
        end - if
end - while
if (currentNode = NILL)
        element ← "0"
element ← [[currentNode].info].value
end - if
```

**end - function**

- **function** add(sm, node, e)

```
if node = NIL then
        @allocate(node)
        [node].info ← e
        [node].left ← NIL
        [node].right ← NIL
```

```
    else if relation([node].info, e) then
            add ([node].left, e)
        else
            add ([node].right, e)
        end - if
    end - if
end - function
```

- **function** update (sm, line, column, value) is:

```
    @ allocate TElement something
    something.line ← line
    something.column ← column
    something.value ← "0"
    @allocate node currentNode
    currentNode ← sm.root
    while (not [currentNode] = NIL and not [[currentNode].info].line = line )
        if (not relation([currentNode].info, something))
            currentNode ← [currentNode].right
        else
            currentNode ← [currentNode].left
        end - if
    end - while
    while (not [currentNode] = NIL and not [[currentNode].info].column =
                                                                    column )
        if (not relation([currentNode].info, something))
            currentNode ← [currentNode].right
        else
            currentNode ← [currentNode].left
        end - if
    end - while
    if ([[currentNode].info].column = column and [[currentNode].info].line
                                                                    = line)
        [[currentNode].info].value = value;
end - function
```

- **function** minimum(sm, n) is:

```
@allocate Node currentNode
currentNode ← n
while (not [currentNode].left = NIL)
        currentNode ← [currentNode].left
end - while
minimum ← currentNode
```

**end - function**

- **function** remove(sm, n, line) is:

```
@allocate int isRoot
isRoot ← 0
if (n == NILL) then
        removeRec ← n else
if (line < [n].info.line) then
        [n].left ← removeRec([n].left, line)
else if (line > [n].info.line) then
        [n].right ← removeRec([n].right, line)
else:
        if ([n].left = NIL and [n].right = NIL) then
                if ( n = [sm].root)
                [sm].root = NIL
                @delete n
                n ← NILL
        else if ([n].right = NIL) then
                if (n = [sm].root) then
                        isRoot ← 1
                        @allocate Node aux
                        aux ← n
                        n ← [n].left
                        delete aux
                        if (isRoot = 1) then
                                [sm].root ← n
                else if ([n].left = NIL) then
                        if (n = [sm].root) then
```

```
                    isRoot ← 1
                    @allocate Node aux
                    aux ← n
                    n ← [n].right
                    delete aux
                    if (isRoot = 1) then
                        [sm].root ← n
        else
            if (n = [sm].root) then
                isRoot ← 1
                @allocate Node aux
                aux ← [sm].minimum([n].right)
                [n].info = [aux].info
                [n].right = removeRec([n].right, [aux].info.line)
                if (isRoot = 1) then
                    [sm].root ← n
            end - if
        end - if
    removeRec ← n
```
**end - function**

- **subalgorithm** modify (sm, line, column, value)

```
    @allocate string oldValue
    oldValue ← element(line, column)
    @allocate TElement somehting
    [something].line ← line
    [something].column ← column
    [something].value ← value
    if (oldValue = "0")
        if (value = "0")
            @break
        else
            add(sm.root, something)
        end - if
    else
        if (value = "0")
```

```
                remove(sm.root, line)
            else
                update(line, column, value)
            end - if
        end - if
end - subalgorithm
```

- **function** iterator (sm) is:

    @ allocate Iterator{sm} to it

    iterator ← it

**end - function**

## 3. Complexity of the operations

### Complexity of all the functions

A) Container

- ~ **init** has the general complexity Θ(1)
- ~ **destroy** has the general complexity Θ(1)
- ~ **relation** has the general complexity Θ(1)
- ~ **noLines** has the general complexity Θ(1)
- ~ **noColumns** has the general complexity Θ(1)
- ~ **element** has the general complexity Θ(n)
- ~ **add** has the general complexity Θ(n)
- ~ **update** has the general complexity Θ(n)
- ~ **minimum** has the general complexity Θ(n)
- ~ **remove** has the general complexity Θ(n)
- ~ **modify** has the general complexity Θ(1)
- ~ **iterator** has the general complexity Θ(1)

B) Iterator

- ~ **init** has the general complexity Θ(n)
- ~ **getCurrent** has the general complexity Θ(1)
- ~ **next** has the general complexity Θ(n)
- ~ **valid** has the general complexity Θ(1)

## Computed complexity for the element function

**Best Case:**

The best case is when the element that we are searching is first position in our binary search tree, having then the complexity $\Theta(1)$. Just the first number is checked, no matter how large the binary tree is.

**Worst Case:**

The worst case possible is that the element we are searching is actually on the last position on the binary tree, then the function will have the complexity $\Theta(n)$. We have to check all numbers from the binary tree.

**Average Case:**

The average case is computed by the formula: $\sum_{I \in D} P(I) \cdot E(I)$ where:

- D is the domain of the problem, the set of every possible input that can be given to the algorithm, in our case {a..z}x{a..z} because k can takes values from a to z and the same v.
- I is the input data
- P(I) is the probability that we will have I as input
- E(I) is the number of operation performed by the algorithm for input I

For our example D would be the set of all possible binary trees with n leafs:

For our example I could be a subset of D in which:

- One I represents all the binary trees where the first element being the one that we are looking for
- One I represents all the binary trees where the second element is the one that we are looking for ...

P(I) is usually considered equal for every I

So the complexity would be something like:

$$\sum_{i=1}^{n} (n+10) = 11 + 12 + 13 + \ldots = \text{(aprox) } n$$

So the average case is actually O(n)

# 4. Test Coverage and Tests

//To be completed for the deadline of the project

# B. Application

## 1. Problem Statement

The problem that was assigned for me to solve:

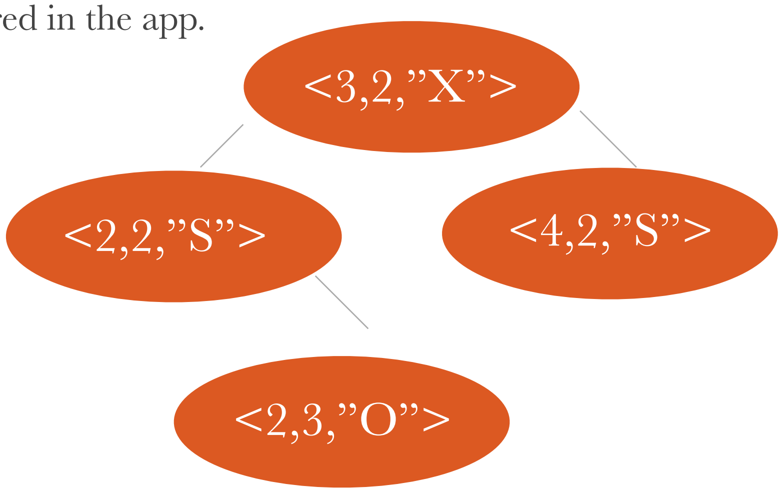
27. ADT SparseMatrix – representation using <line, column, value> triples (value ≠ 0). Implementation on a binary search tree.

The solution that I thought for solving the problem:

I would like to recreate the **Battleship** game which is a guessing game for two players on which the players' fleets of ships (including battleships) are marked. The locations of the fleet are concealed from the other player. Players alternate turns calling "shots" at the other player's ships, and the objective of the game is to destroy the opposing player's fleet.

The reason why I chose this method to solve the problem is because the hole game is played on a matrix. At the beginning there are lots of empty spaces so because of that the sparse matrix is the perfect abstract data type for solving this problem, also because of the data type (binary search tree) the search will be so much faster than searching element by element. Down below I putted an example of how things will be stored in the app.

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | | | | | |
| 2 | | S | O | | |
| 3 | | X | | | |
| 4 | | S | | | |
| 5 | | | | | |




Above is an example of how the app how the app will look like in the end and how is suppose to store things inside of it.

# 2. Solutions to the chosen problem

//To be completed for the deadline of the project

# 3. Complexity for the operations

//To be completed for the deadline of the project