DATA STRUCTURES (AND ALGORITHMS)

ADT Set, ADT Map, ADT Matrix

PhD. Diana-Lucia Miholca

2023 - 2024

Babeş - Bolyai University, Faculty of Mathematics and Computer Science

In Lecture 3...

- Linked Lists
  - Singly linked lists
  - Doubly linked lists
  - Iterator for linked lists

Today

- Abstract Data Types
  - ADT Set
  - ADT Map
  - ADT Matrix

Problem: Student Council Election

Student Council Election: Consider that we have to organize a Student Council voting event and, in order to **avoid multiple voting**, we want to develop a software program which stores the personal numeric codes (IDs) of the actual voters.

What are the characteristics of a container suitable for storing the IDs of the actual voters?

What are the characteristics of a container suitable for storing the IDs of the actual voters?

- Do the elements have to be unique?

What are the characteristics of a container suitable for storing the IDs of the actual voters?

- Do the elements have to be unique?
  - Yes, they have to.

What are the characteristics of a container suitable for storing the IDs of the actual voters?

- Do the elements have to be unique?
  - Yes, they have to.
- Is the order of the elements important?

What are the characteristics of a container suitable for storing the IDs of the actual voters?

- Do the elements have to be unique?
  - Yes, they have to.
- Is the order of the elements important?
  - No, the order is irrelevant.

D **ADT Set** is a container in which the elements have to be unique and their order is not important.

E Examples:

- ✓ {1, 5, 3}
- ✗ {**1**, 5, **1**}
- ✓ {"data", "stucture", "type"}
- ✗ {**"data"**, "stucture", **"data"**, "type"}

- **Domain** of the ADT Set:

Domain of the ADT Set:

$\mathcal{S} = \{s \mid \text{s is a set with (unique) elements of the type TElem}\}$

Domain of the ADT Set:

$\mathcal{S} = \{s \mid \text{s is a set with (unique) elements of the type TElem}\}$

For the *Student Council Election* problem, TElem $=$ String

- The interface of ADT Set should contain operations for:

- The interface of ADT Set should contain operations for:
  - $\emptyset$ creating an empty set

- The interface of ADT Set should contain operations for:
  - creating an empty set
  - destroying an existing set

- The interface of ADT Set should contain operations for:
  - creating an empty set
  - destroying an existing set
  - adding a **new** element

- The interface of ADT Set should contain operations for:
  - creating an empty set
  - destroying an existing set
  - adding a **new** element
  - removing an element

The interface of ADT Set should contain operations for:

- creating an empty set
- destroying an existing set
- adding a **new** element
- removing an element
- searching for a given element

- The interface of ADT Set should contain operations for:
  - ∅ creating an empty set
  - destroying an existing set
  - adding a **new** element
  - removing an element
  - searching for a given element
  - finding the size

- The interface of ADT Set should contain operations for:
  - ∅ creating an empty set
  - destroying an existing set
  - adding a **new** element
  - removing an element
  - searching for a given element
  - finding the size
  - returning an iterator

$\emptyset$ init(s)

- **desc:** creates a new empty set
- **pre:** true
- **post:** $s \in \mathcal{S}$, s is an empty set.

destroy(s)

- **desc:** destroys a set (frees the memory occupied by it)
- **pre:** $s \in S$
- **post:** the set s has been destroyed.

- add(s, e)
  - **desc:** adds a new element into the set if it is not already in the set
  - **pre:** $s \in \mathcal{S}$, $e \in TElem$
  - **post:**
    - $s' \in \mathcal{S}$, $s' = s \cup \{e\}$ (e is added only if it is not in s yet; otherwise, no change is made)

$$add(s, e) = \begin{cases} True, & \text{if } e \text{ has been actually added to s } (not(e \in s)) \\ False, & \text{otherwise} \end{cases}$$

- remove(s, e)
  - **desc:** removes an element from the set.
  - **pre:** $s \in \mathcal{S}$, $e \in TElem$
  - **post:**
    - $s' \in \mathcal{S}$, $s' = s \setminus \{e\}$ (if e is not in s, s is not changed)

$$remove(s, e) = \begin{cases} True, & \text{if } e \text{ has been actually removed } (e \in s) \\ False, & \text{otherwise} \end{cases}$$

search(s, e)

desc: verifies if an element is in the set or not.

pre: $s \in \mathcal{S}$, $e \in TElem$

post:

$$search(s, e) = \begin{cases} True, & \text{if } e \in s \\ False, & \text{otherwise} \end{cases}$$

- size(s)
  - **desc:** returns the number of elements from a set
  - **pre:** $s \in \mathcal{S}$
  - **post:** size = |s| (the number of elements from *s*)

iterator(s, i)

desc: returns an iterator over a given set

pre: $s \in \mathcal{S}$

post:

- $i \in \mathcal{I}$[1], i is an iterator over the set s
- i refers a first element from s (if s is not empty; otherwise, i is invalid)

---

[1] $\mathcal{I} = \{i \mid i$ is an iterator over a set s $\in \mathcal{S}\}$

Other possible operations:

- $\cup$ union of two sets
- $\cap$ intersection of two sets
- $\setminus$ difference of two sets

- Possible representations for ADT Set:
  - Dynamic Array
  - Linked List

Other possible representations for ADT Set:

Hash Tables

Examples:

- *std::unordered_set* in C++ STL
- *HashSet* in Java Collections API
- Python's sets ({ })

Binary Search Trees

Examples:

- *std::set* in C++ STL
- *TreeSet* in Guava (Google Core Libraries for Java)

- How can we represent a Set using a Dynamic Array?
  - What are the specific fields for a Dynamic Array?
  - Is there anything extra we need for representing a Set?

ADT Set ▶ Representation on Dynamic Array

- How can we represent a Set using a Dynamic Array?
  - What are the specific fields for a Dynamic Array?
  - Is there anything extra we need for representing a Set?

ADT Set's representation on a Dynamic Array:

<u>Set:</u>

elems: TElem[]
len: int
cap: int

How can we implement the *add* operation? What cases[1] should we consider?

[1] Remember that the postconditions of the *add* operation ensure that an element is actually added to a set only if it the element is not already present.

How can we implement the *add* operation? What cases[1] should we consider?

We distinguish two cases:

1. If the element is in the set, we return false and add nothing
2. If the element is not in the set, we should actually add it

[1] Remember that the postconditions of the *add* operation ensure that an element is actually added to a set only if it the element is not already present.

How can we implement the *add* operation? What cases[1] should we consider?

We distinguish two cases:

1. If the element is in the set, we return false and add nothing
2. If the element is not in the set, we should actually add it
   - Where should we add the element into the array?

[1] Remember that the postconditions of the *add* operation ensure that an element is actually added to a set only if it the element is not already present.

How can we implement the *add* operation? What cases[1] should we consider?

We distinguish two cases:

1. If the element is in the set, we return false and add nothing
2. If the element is not in the set, we should actually add it

   Where should we add the element into the array?

   ✓ At the end

[1] Remember that the postconditions of the *add* operation ensure that an element is actually added to a set only if it the element is not already present.

How can we implement the *add* operation? What cases[1] should we consider?

We distinguish two cases:

1. If the element is in the set, we return false and add nothing
2. If the element is not in the set, we should actually add it

Where should we add the element into the array?

✓ At the end

We have to consider the case when the array is full and we have to resize it.

[1] Remember that the postconditions of the *add* operation ensure that an element is actually added to a set only if it the element is not already present.

ADT Set ▶ Representation on Dynamic Array ▶ *add* ▶ Example

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

Capacity: 10

Length: 6

- *Add the element 49 to the set*

ADT Set ▶ Representation on Dynamic Array ▶ *add* ▶ Example

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

Capacity: 10
Length: 6

▶ *Add the element 49 to the set*

51	32	19	31	47	95	*49*			
1	2	3	4	5	6	7	8	9	10

Capacity: 10
Length: 7

ADT Set ▶ Representation on Dynamic Array ▶ *add* ▶ Example

51	32	19	31	47	95
1	2	3	4	5	6

Capacity: 6
Length: 6

- *Add the element 49 to the set*

ADT Set ▶ Representation on Dynamic Array ▶ *add* ▶ Example

51	32	19	31	47	95
1	2	3	4	5	6

Capacity: 6
Length: 6

▶ *Add the element 49 to the set*

51	32	19	31	47	95

51	32	19	31	47	95	49					
1	2	3	4	5	6	7	8	9	10	11	12

Capacity: 12
Length: 7

Adding an element into a set:

function add (s, e) **is:**

Adding an element into a set:

function add (s, e) **is:**

index $\leftarrow$ 1

found $\leftarrow$ false

Adding an element into a set:

function add (s, e) **is:**
  index $\leftarrow$ 1
  found $\leftarrow$ false
  //the search part
  while index $\leq$ s.len **and** found = false **execute:**
    if s.elems[index] = e **then**
      found $\leftarrow$ true
    else
      index $\leftarrow$ index + 1
    end-if
  end-while

Adding an element into a set:

function add (s, e) **is:**
    index ← 1
    found ← false
    //the search part
    while index ≤ s.len **and** found = false **execute:**
        if s.elems[index] = e **then**
            found ← true
        else
            index ← index + 1
        end-if
    end-while
    if found = true **then**
        add ← false *//it is already in the set, return false*
    else
    //continued on the next slide ...

Adding an element into a set:

```
if s.len = s.cap then //resize
    s.cap ← s.cap * 2
    newElems ← @a new array with s.cap slots
    for i ← 1, s.len execute
        newElems[i] ← s.elems[i]
    end-if
    free(s.elems)
    s.elems ← newElems
end-if
```

Adding an element into a set:

```
    if s.len = s.cap then //resize
        s.cap ← s.cap * 2
        newElems ← @a new array with s.cap slots
        for i ← 1, s.len execute
            newElems[i] ← s.elems[i]
        end-if
        free(s.elems)
        s.elems ← newElems
    end-if
    //actually add the element
    s.len ← s.len + 1
    s.elems[s.len] ← e
    add ← true
  end-if
end-function
```

What is the time complexity?

What is the time complexity?

Best-case complexity: $\Theta(1)$

What is the time complexity?

- Best-case complexity: $\Theta(1)$
- Worst-case complexity: $\Theta(n)$

What is the time complexity?

- Best-case complexity: $\Theta(1)$
- Worst-case complexity: $\Theta(n)$
- Overall complexity: $O(n)$

How can we represent an Iterator for a Set represented on a Dynamic Array? What would be the type of the Iterator's *cursor*?

How can we represent an Iterator for a Set represented on a Dynamic Array? What would be the type of the Iterator's *cursor*?

Representation of an Iterator over a Set represented on a Dynamic Array:

<u>SetIterator:</u>

s: Set

currentIndex: int

To represent a Set on a SLL (with dynamic allocation) we need two structures: one for the node and one for the Set itself.

SLL's node representation:

SLLNode:
  info: TElem *//the actual information; an element of the set*
  next: ↑ SLLNode *//pointer to the next node*

ADT Set's representation on a SLL:

Set:
  head: ↑ SLLNode *//pointer to the first node*

Adding an element into a set:

function add (s, e) **is:**

Adding an element into a set:

function add (s, e) **is:**
    current ← s.head
    found ← false

Adding an element into a set:

function add (s, e) **is:**
    current $\leftarrow$ s.head
    found $\leftarrow$ false
    while current $\neq$ NIL **and** found = false **execute:**
        if [current].info $=$ e **then**
            found $\leftarrow$ true
        else
            current $\leftarrow$ [current].next
        end-if
    end-while

Adding an element into a set:

function add (s, e) **is:**
    current $\leftarrow$ s.head
    found $\leftarrow$ false
    while current $\neq$ NIL **and** found = false **execute:**
        if [current].info $=$ e **then**
            found $\leftarrow$ true
        else
            current $\leftarrow$ [current].next
        end-if
    end-while
    if found = true **then**
        add $\leftarrow$ false *//it is already in the set, return false*
    //continued on the next slide ...

Adding an element into a set:

```
  else
    newNode ← allocate() //allocate a new SLLNode
    [newNode].info ← e
    [newNode].next ← s.head
    s.head ← newNode
    add← True
end-function
```

What is the time complexity?

What is the time complexity?

What is the time complexity?

- Best-case complexity: $\Theta(1)$

What is the time complexity?

- Best-case complexity: $\Theta(1)$
- Worst-case complexity: $\Theta(n)$

What is the time complexity?

- Best-case complexity: $\Theta(1)$
- Worst-case complexity: $\Theta(n)$
- Overall complexity: $O(n)$

How can we represent an Iterator for a Set represented on a SLL with dynamic allocation? What would be the type of the Iterator's *cursor*?

How can we represent an Iterator for a Set represented on a SLL with dynamic allocation? What would be the type of the Iterator's *cursor*?

Representation of an Iterator over a Set represented on a SLL:

<u>SetIterator:</u>
    s: Set
    currentNode: $\uparrow$ SLLNode

Applications of ADT Set:

Websites

- For storing usernames

Compilers

- For storing the programming language's keywords

Playlists

- For storing songs

ADT Set ▶ Conclusions

ADT Set $=$ unordered container with **no** duplicates

$\mathcal{S} = \{s \mid \text{s is a set with elements of the type TElem}\}$

Interface:

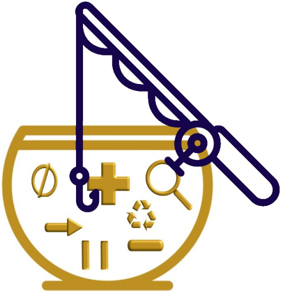
Possible representations:

- Dynamic Array
- Linked List

ADT Set = unordered container with **no** duplicates

$\mathcal{S} = \{s \mid \text{s is a set with elements of the type TElem}\}$

Interface:



Possible representations:

- Dynamic Array : $O(n)$
- Linked List : $O(n)$

Write the Pseudocode for the *remove* operation of the ADT Set when representing it using:

- a Dynamic Array
- a dynamically allocated Singly Linked List

Think about an efficient way of representing a Set with elements from $\{1, 2, 3, ..., N\}$, $N \in \mathbf{N}$ using an Array.

- What would be the time-complexity of the *search*, *add*, *remove* and *size* operations?

Problem: Most frequent words

P Consider the following problem:

We are given a text and we want to find the words that appear most frequently in this text.

? What would be the characteristics of a container used to store the words?

Problem: Most frequent words

P Consider the following problem:
We are given a text and we want to find the words that appear most frequently in this text.

? What would be the characteristics of a container used to store the words?

- We need key (word) - value (number of occurrences) pairs.
- Keys should be unique.
- The order of the keys is not important.

Problem: Most frequent words

P Consider the following problem:
We are given a text and we want to find the words that appear most frequently in this text.

? What would be the characteristics of a container used to store the words?

- We need key (word) - value (number of occurrences) pairs.
- Keys should be unique.
- The order of the keys is not important.

D The container in which we store key - value pairs, and where the keys are unique and in no particular order is the **ADT Map** (or Dictionary).

Domain of the ADT Map:

$\mathcal{M} = \{m \mid$ m is a map with elements $e =< k, v >,$ where $k \in TKey$ and $v \in TValue\}$

init(m)

- **desc:** creates a new empty map
- **pre:** true
- **post:** $m \in \mathcal{M}$, m is an empty map.

- destroy(m)
  - **desc:** destroys a map
  - **pre:** $m \in \mathcal{M}$
  - **post:** m has been destroyed

- add(m, k, v)

  desc: add a new key-value pair to the map. If the key is already in the map, its corresponding value will be replaced with the new one and the operation will return the old value. Otherwise, if the key is not in the map yet, then the new pair will be added to it and the operation will return 0_{TValue}.

  pre: $m \in \mathcal{M}, k \in TKey, v \in TValue$

  post: $m' \in \mathcal{M}, m' = m \cup < k, v >, add = v', v' \in TValue$ where

$$v' = \begin{cases} v'', & \text{if } \exists < k, v'' > \in m \\ 0_{TValue}, & \text{otherwise} \end{cases}$$

- remove(m, k)

desc: removes a pair with a given key from the map. Returns the value associated with the key or 0_{TValue} if the key is not in the map.

pre: $m \in \mathcal{M}, k \in TKey$

post: $m' \in \mathcal{M}, m' = m - < k, v' >$, if $\exists < k, v' > \in m$, $remove = v$, $v \in TValue$, where

$$v = \begin{cases} v', & \text{if } \exists < k, v' > \in m \\ 0_{TValue}, & \text{otherwise} \end{cases}$$

search(m, k)

desc: searches for the value associated with a given key in the map

pre: $m \in \mathcal{M}, k \in TKey$

post: $search = v$, $v \in TValue$, where

$$v = \begin{cases} v', & \text{if } \exists < k, v' > \in m \\ 0_{TValue}, & \text{otherwise} \end{cases}$$

- size(m)
  - **desc:** returns the number of pairs from the map
  - **pre:** $m \in \mathcal{M}$
  - **post:** size = the number of pairs from m

- isEmpty(m)
  - **desc:** verifies if the map is empty
  - **pre:** $m \in \mathcal{M}$
  - **post:** $\textit{isEmpty} = \begin{cases} \textit{true}, & \text{if m contains no pairs} \\ \textit{false}, & \text{otherwise} \end{cases}$

- iterator(m, it)
  - **desc:** returns an iterator for a map
  - **pre:** $m \in \mathcal{M}$
  - **post:** $it \in \mathcal{I}$, *it* is an iterator over *m*

Obs: The iterator for a Map returns, through its *getCurrent* operation, $<$key, value$>$ pairs.

Other possible operations: *keys*, *values*, *pairs*

- keys(m, s)
  - **desc:** returns the set of keys from the map
  - **pre:** $m \in \mathcal{M}$
  - **post:** $s \in \mathcal{S}$, s is the set of all keys from m

- values(m, b)
  - **desc:** returns a bag with all the values from the map
  - **pre:** $m \in \mathcal{M}$
  - **post:** $b \in \mathcal{B}$, b is the bag of all values from m

- pairs(m, s)
  - **desc:** returns the set of pairs from the map
  - **pre:** $m \in \mathcal{M}$
  - **post:** $s \in \mathcal{S}$, s is the set of all pairs from m

Real-word applications of ADT Map:

Compilers

- For mapping the variables names with memory locations

File system

- For mapping file names to the the file path and to the physical location of that file on the disk

Maps

- To diagrammatically represent areas (for instance, a game field)

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

The domain of the ADT Matrix:
$\mathcal{MAT} = \{m \mid$ m is a matrix with elements of the type TElem$\}$

init(mat, nrR, nrC)

desc: creates a new matrix with a given number of rows and columns

pre: $nrR \in N^*$ and $nrC \in N^*$

post: $mat \in \mathcal{MAT}$, *mat* is a matrix with *nrR* rows and *nrC* columns

throws: an exception if *nrR* or *nrC* is negative or zero

- nrRows(mat)
  - **desc:** returns the number of rows of the matrix
  - **pre:** $mat \in \mathcal{MAT}$
  - **post:** $nrRows$ = the number of rows from mat

- nrCols(mat)
  - **desc:** returns the number of columns of the matrix
  - **pre:** $mat \in \mathcal{MAT}$
  - **post:** *nrCols* = the number of columns from *mat*

- element(mat, i, j)

  desc: returns the element from a given position from the matrix

  pre: $mat \in \mathcal{MAT}$, $1 \leq i \leq nrRows$, $1 \leq j \leq nrColumns$

  post: $element =$ the element from row i and column j

  throws: an exception if the position (i, j) is not valid (less than 1 or greater than nrRows/nrColumns)

- modify(mat, i, j, val)

  desc: sets the element from a given position to a given value

  pre: $mat \in \mathcal{MAT}$, $1 \leq i \leq nrRows$, $1 \leq j \leq nrColumns$, $val \in TElem$

  post: the value from position (i, j) is set to *val*. $modify$ = the old value from position (i, j)

  throws: an exception if position (i, j) is not valid (less than 1 or greater than nrRows/nrColumns)

Other possible operations:

- get the (first) position of a given element
- create an iterator that iterates by rows
- create an iterator that iterates by columns
- etc.

Usually a sequential representation is used for a Matrix (the rows are memorized one after another in a consecutive memory blocks).

- For a matrix with N rows and M columns, the memory address of an element from position (i, j) can be computed as:
  address of element from position (i, j) = address of the matrix + (i * M + j) * size of an element

  The above formula works for 0-based indexing, but can be adapted to 1-based indexing as well.

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

ADT Matrix ▶ Sparse Matrix Example

0	33	0	100	1	0	0	9
2	0	2	0	2	0	7	0
0	4	0	0	3	0	0	0
17	0	0	10	0	16	0	7
0	0	0	0	0	0	0	0
0	1	0	13	0	8	0	29

- Number of rows (lines): 6
- Number of columns: 8

We can memorize (line, column, value) triples, where *value* is different from 0 (or 0_{TElem}). For efficiency, we can memorize the elements sorted by the (row, column) pairs.

Triples can be stored in:

- (dynamic) arrays
- linked lists
- other data structures

ADT Matrix ▶ Sparse Matrix ▶ Example

0	33	0	100	1	0	0	9
2	0	2	0	2	0	7	0
0	4	0	0	3	0	0	0
17	0	0	10	0	16	0	7
0	0	0	0	0	0	0	0
0	1	0	13	0	8	0	29

Line	1	1	1	1	2	2	2	2	3	3	4	4	4	4	6	6	6	6
Col	2	4	5	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	1	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

ADT Matrix ▶ Sparse Matrix ▶ modify

In the interface of the ADT Matrix we only have a *modify* operation which changes a value from a given position, but no *add* or *remove* operations. If we represent a matrix as a sparse matrix, the *modify* operation might add/remove an element to/from the underlying data structure.

We distinguish four different cases for the *modify*, depending on the current value at the given position (*current_value*) and the new value we want to put there (*new_value*).

- $current\_value = 0$ and $new\_value = 0 \Rightarrow$ do nothing
- $current\_value = 0$ and $new\_value \neq 0 \Rightarrow$ insert in the data structure
- $current\_value \neq 0$ and $new\_value = 0 \Rightarrow$ remove from the data structure
- $current\_value \neq 0$ and $new\_value \neq 0 \Rightarrow$ just change the value in the data structure

Initial Sparse Matrix:

Line	1	1	1	1	2	2	2	2	3	3	4	4	4	4	6	6	6	6
Col	2	4	5	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	1	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

- Modify the value from position (1, 5) to 0

Line	1	1	1	2	2	2	2	3	3	4	4	4	4	6	6	6	6
Col	2	4	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

- Modify the value from position (3, 3) to 19

Line	1	1	1	2	2	2	2	3	3	3	4	4	4	4	6	6	6	6
Col	2	4	8	1	3	5	7	2	3	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	19	3	17	10	16	7	1	13	8	29

Representation of a node:

Node:

info: TElem

row: Integer

column: Integer

next: ↑ Node

Representation of a SparseMatrix stored using a SLL:

SparseMatrix:

head: ↑ Node

nrRows: Integer

nrColumns: Integer

How can we implement the *element* operation? What cases should we consider?

How can we implement the *element* operation? What cases should we consider?

- First, we should check if the position is valid. If not, we will throw an exception.
- If the position is valid, we should check if we have a node with the given row and column. If not, we return NULL_TElem.
- If we find such a node, we return the *info* from it.

Since the nodes are ordered by row and column, we can stop searching when we reach a position subsequent to the given position.

Returning the element at a given position in a SparseMatrix:

function element(mat, i, j) **is**
    if $i < 1$ **OR** $j < 1$ **OR** $i > mat.nrRows$ **OR** $j > mat.nrColumns$ **then**
        @throw exception
    end-if
    current $\leftarrow$ mat.head
    //look for the element. Stop if we passed the position where it should be
    while current $\neq$ NULL **AND** ([current].row $<$ i **OR** ([current].row = i **AND** [current].column $<$ j)) **execute**
        current $\leftarrow$ [current].next
    end-while
    //check if we found the element
    if current $\neq$ NULL **AND** [current].row = i **AND** [current].column = j **then**
        element $\leftarrow$ [current].info
    else
        element $\leftarrow$ NULL_TElem
    end-if
end-function

What is the time complexity?

Returning the element at a given position in a SparseMatrix:

function element(mat, i, j) **is**
    if $i < 1$ **OR** $j < 1$ **OR** $i > mat.nrRows$ **OR** $j > mat.nrColumns$ **then**
        @throw exception
    end-if
    current $\leftarrow$ mat.head
    //look for the element. Stop if we passed the position where it should be
    while current $\neq$ NULL **AND** ([current].row $<$ i **OR** ([current].row = i **AND** [current].column $<$ j)) **execute**
        current $\leftarrow$ [current].next
    end-while
    //check if we found the element
    if current $\neq$ NULL **AND** [current].row = i **AND** [current].column = j **then**
        element $\leftarrow$ [current].info
    else
        element $\leftarrow$ NULL_TElem
    end-if
end-function

What is the time complexity? ✓ O(number of non-zero elements)

Applications of ADT Matrix:

Machine Learning

- Representing the training data

Optics

- For representing refraction and reflection

Image processing

- An image is a matrix of pixels

Bibliography

- David M. Mount, *Lecture notes for the course Data Structures* (CMSC 420), at the Dept. of Computer Science, University of Maryland, College Park, 2001
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Third Edition, The MIT Press, 2009
- Narasimha Karumanchi, *Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles*, Fifth Edition, 2016
- Clifford A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*, Third Edition, 2010

Thank you