DATA STRUCTURES AND ALGORITHMS

LECTURE 3

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2023 - 2024

In Lecture 2...

- Dynamic array
- Iterators
- ADT Bag

Today

- Containers

Containers

- There are many different containers, based on different properties:
  - do the elements have to be unique?
  - do the elements have positions assigned?
  - can we access any element or just some specific ones?
  - do we have simple elements, or key-value pairs?

ADT Bag

- The ADT Bag is a container in which the elements are not unique and they do not have positions.
- Interface of the Bag was discussed at Seminar 1.

ADT Bag - representation I

- Remember, in Lecture 2 we have talked about 3 possible representations:
- R1 - a sequence of elements, with duplicates.

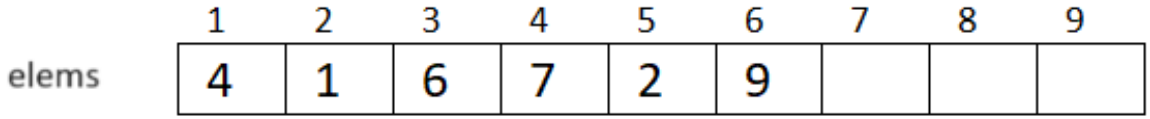
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
4	1	6	4	7	2	1	1	1	9						

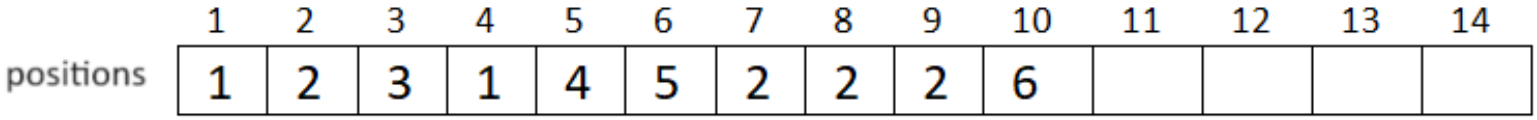
- R2 - a sequence of unique element - frequency pairs.

	1	2	3	4	5	6	7	8	9
elems	4	1	6	7	2	9			
freq	2	4	1	1	1	1			

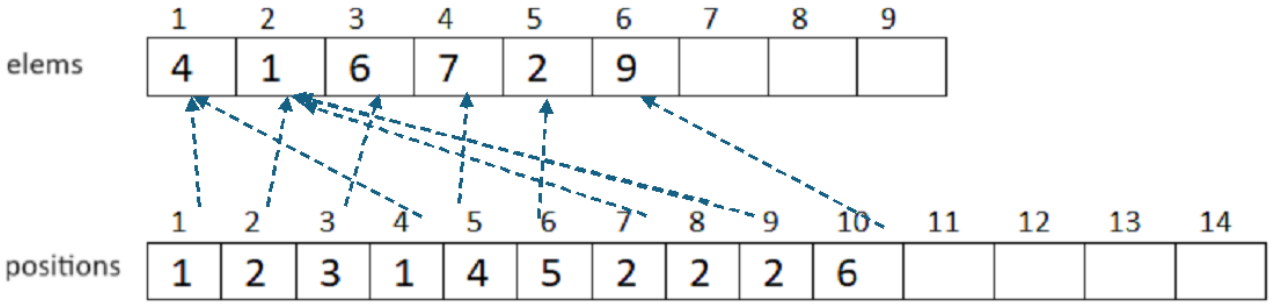
ADT Bag - representation II

- R3 (dynamic array specific) - an array of unique elements and an array of positions



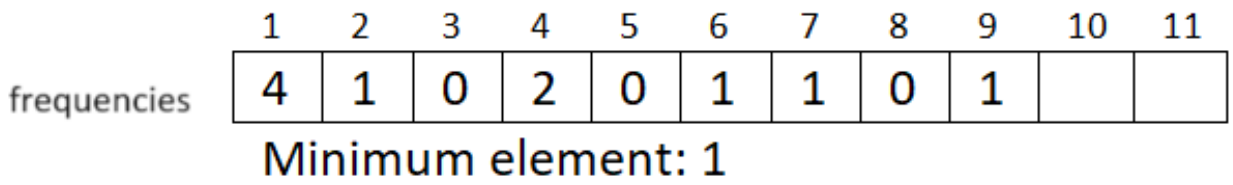


- You can see the actual elements that are in the bag, if you follow the arrows from the *positions* array.



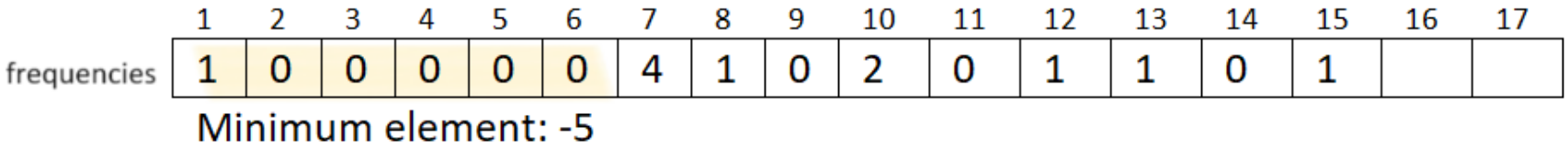
ADT Bag - R4

- If the elements of the Bag are integer numbers (and a dynamic array is used for storing them), another representation is possible, where the positions of the array represent the elements and the value from the position is the frequency of the element. Thus, the frequency of the minimum element is at position 1 (assume 1-based indexing).
- Assume the same elements as before: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R4 the Bag looks in the following way:



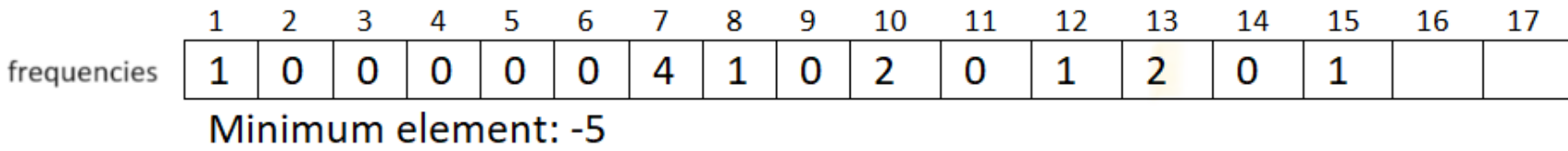
ADT Bag - R4 example

- Add element -5



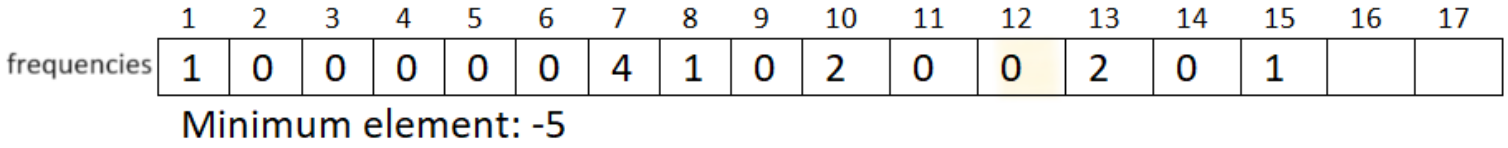
- When indexing starts from 1, the element in the dynamic array that is on position i, represents the actual value: $minimum + i - 1 \Rightarrow$ position of an element e is $e - minimum + 1$

- Add element 7

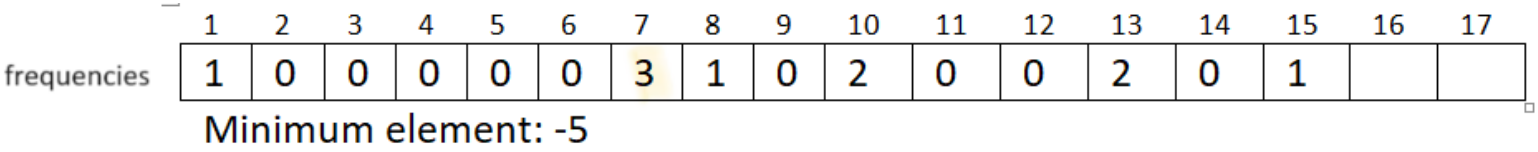


ADT Bag - R4 example

- Remove element 6



- Remove element 1



ADT Bag - R4 example I

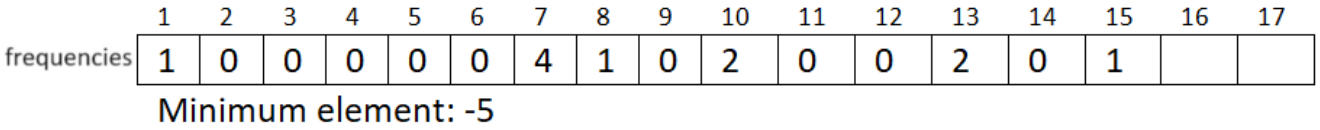
- If the elements are far from each other, R4 can be a representation which uses a lot of memory.
- Can you think of an advantage that R4 has (over R1, R2 and R3)?

ADT Bag - R4 example II

- Resize is an interesting operation in case of R4: you must resize when you add a new maximum or a new minimum (you have no place where to put them), however, when you remove the last occurrence of the minimum or maximum element, in theory you could choose to not resize (just leave the 0 frequency). However, this is going to possibly waste even more memory, so either you need a more complex mechanism for checking when to resize (for example, when the first/last X% of the array is empty) and during resize to need to make sure to not copy the empty ends of the array or you can simply decide to resize whenever the minimum or the maximum is removed.

ADT Bag - R4 Iterator

- How would you define the iterator in case of this representation?



- Even if the array used for representing the Bag contains *gaps* (positions with zero frequency, representing elements which are not there), the iterator is only allowed to return existing elements.
- This means that when we go to the next element, we need to skip the positions with 0 frequency $\Rightarrow$ complexity of operation *next* will not be $\Theta(1)$.
- Also, remember, since it is a frequency-based representation, iterator needs to consider a *current frequency* as well (discussed in Seminar 1 for R2, but it applies to every frequency-based representation).

ADT Bag - R4 Iterator

- So, what would be the complexities of the iterator operations in case of R4?
- Do these complexities depend on your resize policy for the Bag (discussed 3 slides ago)?
- Could you make them NOT depend on the resize policy?

ADT Sorted Bag

- There are no positions in a Bag, but sometimes we need the elements to be sorted $\Rightarrow$ ADT SortedBag.
- Remember, these were the operations in the interface of the ADT Bag:
  - init(b)
  - add(b, e)
  - remove(b, e)
  - search(b, e)
  - nrOfOccurrences(b, e)
  - size(b)
  - iterator(b, it)
  - destroy
- What should be different (new operations, removed operations, modified operations) in case of a *SortedBag*?

ADT Sorted Bag

- The only modification in the interface is that the init operation receives a *relation* as parameter
- Domain of Sorted Bag:
  - $\mathcal{SB} =$ $\{\textbf{sb} | sb \textit{ is a sorted bag that uses a relation to order the elements}\}$
- init (sb, rel)
  - **descr:** creates a new, empty sorted bag, where the elements will be ordered based on a relation
  - **pre:** $rel \in Relation$
  - **post:** $sb \in \mathcal{SB}$, sb is an empty sorted bag which uses the relation rel

The relation

- Usually there are two approaches, when we want to order elements:
  - Assume that they have a *natural ordering*, and use this ordering (for ex: alphabetical ordering for strings, ascending ordering for numbers, etc.).
  - Sometimes, we want to order the elements in a different way than the natural ordering (or there is no natural ordering) $\Rightarrow$ we use a relation
  - A relation will be considered as a function with two parameters (the two elements that are compared) which returns *true* if they are in the correct order, or *false* if they should be reversed.

ADT Sorted Bag

- The other operations from the interface are the same for a Bag and a SortedBag.
- A SortedBag does not have positions; how the elements are stored internally, is hidden. Then how can I see the difference between a Bag and a SortedBag?
  - the iterator for a SortedBag has to return the elements in the order given by the relation.
  - Since the iterator operations should have a $\Theta(1)$ complexity, this means that internally the elements have to be stored based on the relation.

ADT SortedBag - representation

- A SortedBag can be represented using several data structure, one of them being the dynamic array (others will be discussed later):
- Independently of the chosen data structure, there are two options for storing the elements:
  - Store separately every element that was added (in the order given by the relation)
  - Store each element only once (in the order given by the relation) and keep a frequency count for it
  - Assuming that the elements are integer numbers and we use a Dynamic array as data structure, is R4 from the Bag a possible representation for a SortedBag?

- Consider the following problem: *in order to avoid electoral fraud (especially the situation when someone votes multiple times in different locations) we want to build a software system which stores the personal numerical code (CNP) of everyone who votes.* What would be the characteristics of the container used to store these personal numerical codes?
  - The elements have to be unique
  - The order of the elements is not important
- The container in which the elements have to be unique and the order of the elements is not important (there are no positions) is the **ADT Set**.

ADT Set - Domain

- Domain of the ADT Set:
  $\mathcal{S} = \{s | \text{s is a set with elements of the type TElem}\}$

ADT Set - Interface I

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

ADT Set - Interface II

- add(s, e)
  - **descr:** 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. If s contains the element e already, no change is made).
    $add \leftarrow$ true if e was added to the set, *false* otherwise.

ADT Set - Interface III

- remove(s, e)
  - **descr:** 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* $\leftarrow$ true, if e was removed, *false* otherwise

ADT Set - Interface IV

- search(s, e)
  - **descr:** verifies if an element is in the set.
  - **pre:** $s \in \mathcal{S}$, $e \in TElem$
  - **post:**

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

ADT Set - Interface V

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

ADT Set - Interface VI

- isEmpty(s)
  - **descr:** verifies if the set is empty
  - **pre:** $s \in \mathcal{S}$
  - **post:**

$$isEmpty \leftarrow \begin{cases} True, & \text{if } s \text{ has no elements} \\ False, & \text{otherwise} \end{cases}$$

ADT Set - Interface VII

- iterator(s, it)
  - **descr:** returns an iterator for a set
  - **pre:** $s \in \mathcal{S}$
  - **post:** $it \in \mathcal{I}$, it is an iterator over the set s

ADT Set - Interface VIII

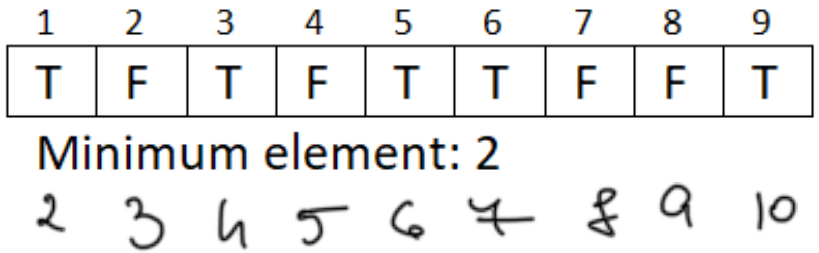
- destroy (s)
  - **descr:** destroys a set
  - **pre:** $s \in S$
  - **post:**the set s was destroyed.

ADT Set - Interface IX

- Other possible operations (characteristic for sets from mathematics):
  - reunion of two sets
  - intersection of two sets
  - difference of two sets (elements that are present in the first set, but not in the second one)

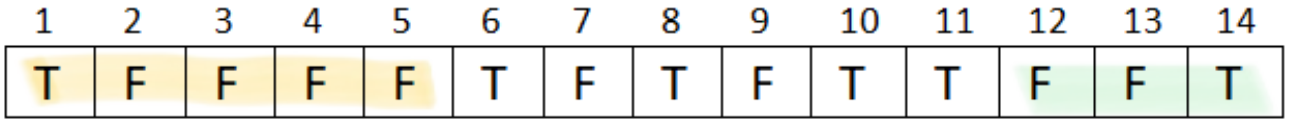
ADT Set - representation

- If a Dynamic Array is used as data structure and the elements of the set are numbers, we can choose a representation in which the elements are represented by the positions in the dynamic array and a boolean value from that position shows if the element is in the set or not.
- Assume a Set with the following numbers: 4, 2, 10, 7, 6.
- This Set would be represented in the following way (the formulae discussed at Bag can be applied here as well):



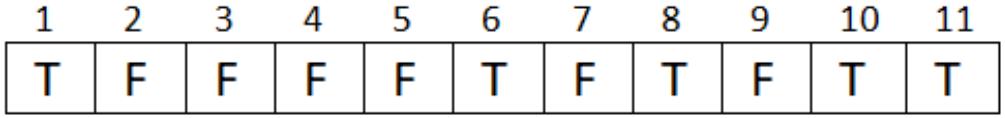
ADT Set - representation

- Add element -3



Minimum element: -3

- Remove element 10

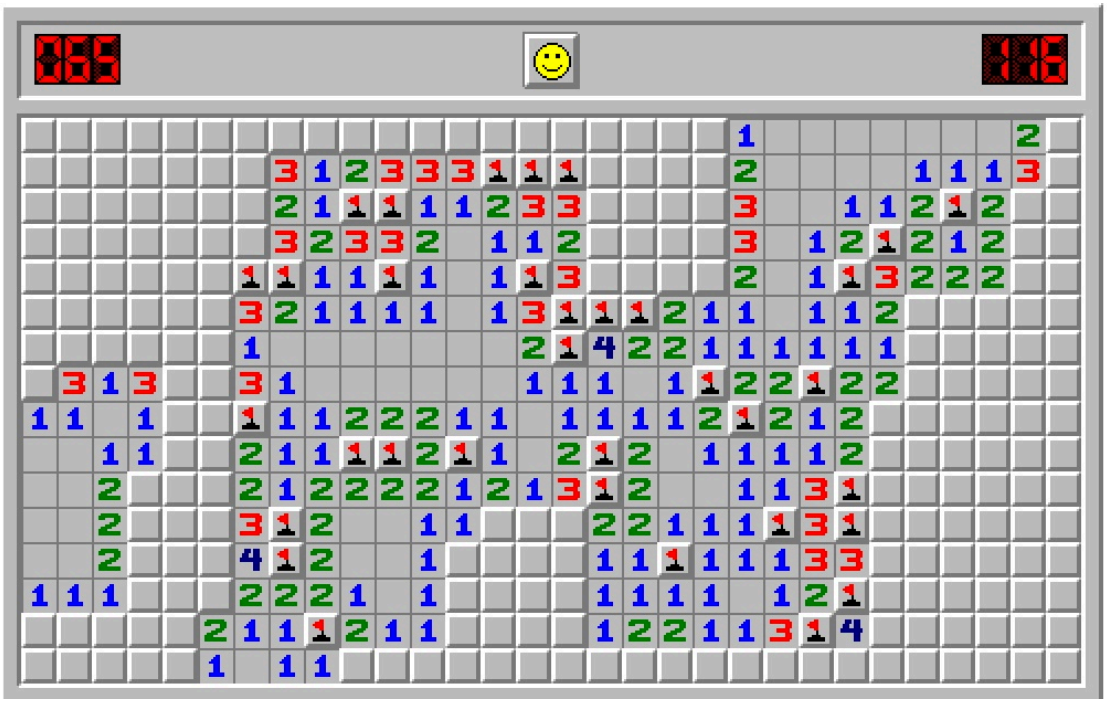


Minimum element: -3

ADT Sorted Set

- We can have a Set where the elements are ordered based on a *relation* $\Rightarrow$ *SortedSet*.
- The only change in the interface is for the *init* operation that will receive the *relation* as parameter.
- For a sorted set, the iterator has to iterate through the elements in the order given by the *relation*, so we need to keep them ordered in the representation.

- Imagine that you wanted to implement this game:



Source: http:minesweeperonline.com#

- What would be the specifics of the container needed to store the game board (location of the mines)?

ADT Matrix

- 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.
- The domain of the ADT Matrix: $\mathcal{MAT} = \{mat | \text{mat is a matrix with elements of the type TElem}\}$
- What operations should we have for a Matrix?

ADT Matrix - Interface I

- 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:** $mat \in \mathcal{MAT}$, mat is a matrix with nrL lines and nrC columns
  - **throws:** an exception if nrL or nrC is negative or zero

ADT Matrix - Interface II

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

ADT Matrix - Interface III

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

ADT Matrix - Interface IV

- element(mat, i, j)
  - **descr:** returns the element from a given position from the matrix (assume 1-based indexing)
  - **pre:** $mat \in \mathcal{MAT}$, $1 \leq i \leq nrLines$, $1 \leq j \leq nrColumns$
  - **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)

ADT Matrix - Interface V

- modify(mat, i, j, val)
  - **descr:** sets the element from a given position to a given value (assume 1-based indexing)
  - **pre:** $mat \in \mathcal{MAT}$, $1 \leq i \leq nrLines$, $1 \leq j \leq nrColumns$, $val \in TElem$
  - **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)

ADT Matrix - Operations

- Other possible operations:
  - get the (first) position of a given element
  - create an iterator that goes through the elements by columns
  - create an iterator the goes through the elements by lines
  - etc.

ADT Matrix - representation

- Usually a sequential representation is used for a Matrix (we memorize all the lines one after the other in a consecutive memory block).
- If this sequential representation is used, for a matrix with N lines and M columns, the element from position (i, j) can be found at the memory address:
  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.

ADT Matrix - representation

```
Size of int: 4
Address of matrix (5 rows, 8 cols): 6224024
Address of element 0, 0: 6224024
Address of element 2, 4: 6224104
Address of element 2, 5: 6224108
Address of element 2, 6: 6224112
Address of element 2, 7: 6224116
Address of element 3, 0: 6224120
Address of element 3, 4: 6224136
Address of element 4, 7: 6224180
```

ADT Matrix - representation

- In the Minesweeper game example above we have a matrix with 480 elements (16 * 30) but only 99 bombs.
- If the Matrix contains many values of 0 (or 0_{TElem}), we have a *sparse matrix*, where it is more (space) efficient to memorize only the elements that are different from 0.

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 lines: 6
- Number of columns: 8

Sparse Matrix - R1

- We can memorize (line, column, value) triples, where value is different from 0 (or 0_{TElem}). For efficiency, we memorize the elements sorted by the (line, column) pairs (if the lines are different we order by line, if they are equal we order by column) - R1.
- If a (line, column) pair is not in the representation, we know that its value is 0 (or 0_{TElem}):
- Triples can be stored in a dynamic array or other data structures (will be discussed later):

Sparse Matrix - R1 example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
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

- In an ADT Matrix, there is no operation to add an element or to remove an element. In the interface we only have the *modify* operation which changes a value from a position. If we represent the matrix as a sparse matrix, the *modify* operation might add or remove an element to/from the underlying data structure. But the operation from the interface is still called *modify*.

Modify

- When we have a Sparse Matrix (i.e., we keep only the values different from 0), for the modify operation we have four different cases, based on the value of the element currently at the given position (let's call it *current_value*) and the new value that we want to put on that position (let's call it *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

Sparse Matrix - R1 example

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

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
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

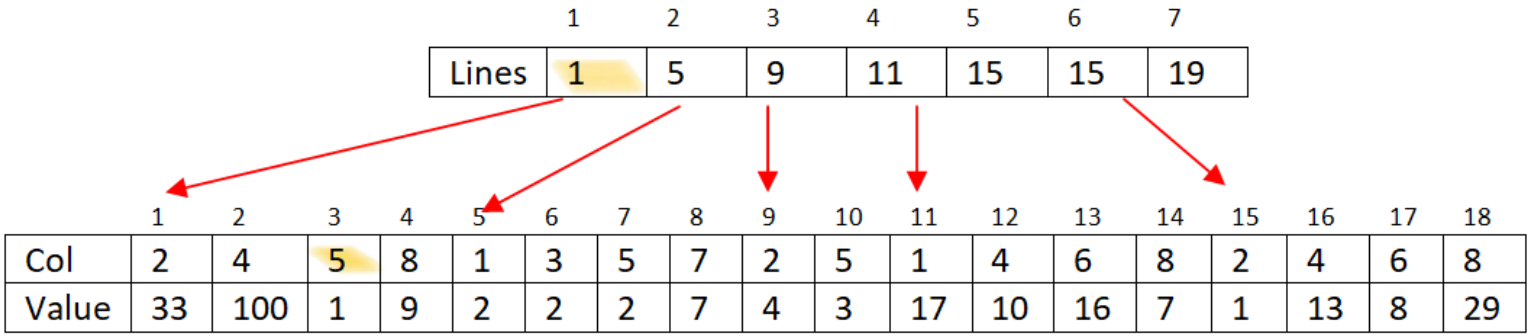
- Modify the value from position (3, 3) to 19

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
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

Sparse Matrix - R2

- We can see that in the previous representation there are many consecutive elements which have the same value in the line array. The array containing this information could be compressed, in the following way:
  - Keep the *Col* and *Value* arrays as in the previous representation.
  - For the lines, have an array of number of lines + 1 element, in which at position i we have the position from the *Col* array where the sequence of elements from line *i* begins.
  - Thus, elements from line *i* are in the *Col* and *Value* arrays between the positions [*Line*[i], *Line*[i+1]).
- This is called **compressed sparse line representation**.
- **Obs:** In order for this representation to work, in the *Col* and *Value* arrays the elements have to be stored by rows (first elements of the first row, then elements of second row, etc.)

Sparse Matrix - R2 example

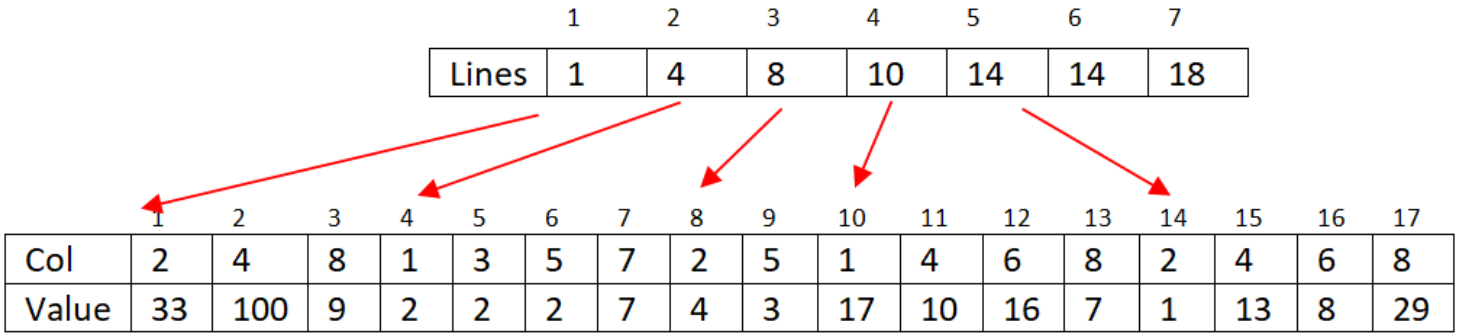


	1	2	3	4	5	6	7
Lines	1	5	9	11	15	15	19

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
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

Sparse Matrix - R2 example

- Modify the value from position (1, 5) to 0
  - First we look for element on position (1,5).
  - Elements from line 1 are between positions 1 and 4 (inclusive)
  - Since we have there an item with column 5, we found our element
  - Setting to 0, means removing from *Col* and *Value* array.
  - In *Lines* array just the values change, not the size of the array.

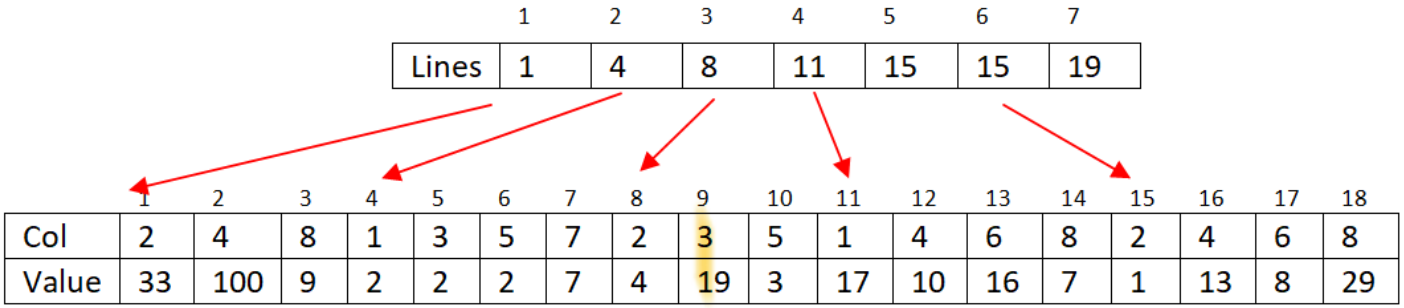


	1	2	3	4	5	6	7
Lines	1	4	8	10	14	14	18

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
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

Sparse Matrix - R2 example

- Modify the value from position (3, 3) to 19
  - First we look for element on position (3,3)
  - Elements from line 3 are between positions 8 and 9 (inclusive)
  - Since we have no column 3 there, at this position currently the value is 0. To set it to 19 we need to insert a new element in the *Col* and *Value* array.
  - In *Lines* array just the values change, not the size of the array



	1	2	3	4	5	6	7
Lines	1	4	8	11	15	15	19

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
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

Sparse Matrix - R3

- In a similar manner, we can define **compressed sparse column representation**:
  - We need two arrays *Lines* and *Values* for the non-zero elements, in which first the elements of the first column are stored, than elements from the second column, etc.
  - We need an array with nrColumns + 1 elements, in which at position i we have the position from the *Lines* array where the sequence of elements from column *i* begins.
  - Thus, elements from column *i* are in the *Lines* and *Value* arrays between the positions [*Col*[i], *Col*[i+1]).

Sparse Matrix - R3 example

	1	2	3	4	5	6	7	8	9
Cols	1	3	6	7	10	13	15	16	19

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Lines	2	4	1	3	6	2	1	4	6	1	2	3	4	6	2	1	4	6
Value	2	17	33	4	1	2	100	10	13	1	2	3	16	8	7	9	7	29

- Consider the following problem: *we have a text and want to find the word that appears most frequently in this text.* How would you do it, and what would be the characteristics of the container used for this problem?
  - We need key (word) - value (number of occurrence) pairs
  - Keys should be unique
  - Order of the keys is not important
- The container in which we store key - value pairs, and where the keys are unique and they are in no particular order is the **ADT Map** (or Dictionary)

ADT Map

- Domain of the ADT Map:

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

ADT Map - Interface I

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

ADT Map - Interface II

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

ADT Map - Interface III

- add(m, k, v)
  - **descr:** add a new key-value pair to the map (the operation can be called *put* as well). If the key is already in the map, the corresponding value will be replaced with the new one. The operation returns the old value, or 0_{TValue} if the key was not in the map yet.
  - **pre:** $m \in \mathcal{M}, k \in TKey, v \in TValue$
  - **post:** $m' \in \mathcal{M}, m' = m \cup < k, v >$, $add \leftarrow v'$, $v' \in TValue$ where

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

ADT Map - Interface IV

- remove(m, k)
  - **descr:** 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:** $remove \leftarrow v$, $v \in TValue$, where

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

ADT Map - Interface V

- search(m, k)
  - **descr:** searches for the value associated with a given key in the map
  - **pre:** $m \in \mathcal{M}, k \in TKey$
  - **post:** $search \leftarrow v$, $v \in TValue$, where

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

ADT Map - Interface VI

- iterator(m, it)
  - **descr:** 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 the map is similar to the iterator for other ADTs, but the *getCurrent* operation returns a <key, value> pair.

ADT Map - Interface VII

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

ADT Map - Interface VIII

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

Other possible operations I

- Other possible operations
- keys(m, s)
  - **descr:** 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

Other possible operations II

- values(m, b)
  - **descr:** 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

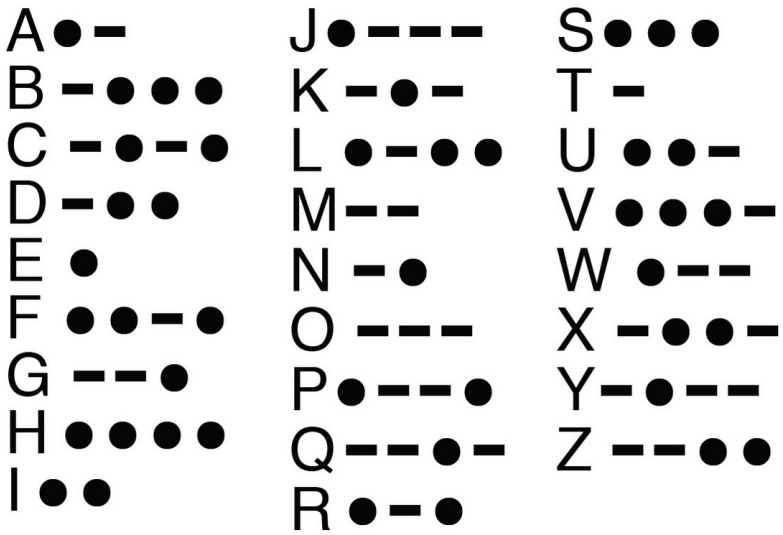
Other possible operations III

- pairs(m, s)
  - **descr:** 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

ADT Sorted Map

- We can have a Map where we can define an order (a relation) on the set of possible keys
- The only change in the interface is for the *init* operation that will receive the *relation* as parameter.
- For a sorted map, the iterator has to iterate through the pairs in the order given by the *relation*, and the operations *keys* and *pairs* return SortedSets.

- Morse Code, is a code which assigns to every letter a sequence of dots and dashes.



https://medium.com/@timboucher/learning-morse-code-35e1f4d285f6

- Given a list of words, find the largest subset of the words, for which the Morse representation is the same.

- For example, if the words are *cat*, *ca*, *nna*, *abc* and *nnet*, their Morse code representation is:
  - *cat* -.-..–
  - *ca* -.-..-
  - *nna* -.-..-
  - *abc* .–...-.-.
  - *nnet* -.-..-
- How would you solve the problem and What would be the characteristics of the container used for the solution?
  - We could solve the problem if we used the Morse representation of a word as a key and the corresponding word as a value
  - One key can have multiple values
  - Order of the elements is not important
- The container in which we store key - value pairs, and where a key can have multiple associated values, is called an **ADT MultiMap**.

ADT MultiMap

- Domain of ADT MultiMap:

$\mathcal{MM} = \{mm | mm$ is a Multimap with TKey, TValue, pairs$\}$

ADT MultiMap - Interface I

- init (mm)
  - **descr:** creates a new empty multimap
  - **pre:** true
  - **post:** $mm \in \mathcal{MM}$, mm is an empty multimap

ADT MultiMap - Interface II

- destroy(mm)
  - **descr:** destroys a multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** the multimap was destroyed

ADT MultiMap - Interface III

- add(mm, k, v)
  - **descr:** add a new pair to the multimap
  - **pre:** $mm \in \mathcal{MM}$, $k - TKey$, $v - TValue$
  - **post:** $mm' \in \mathcal{MM}$, $mm' = mm \cup < k, v >$

ADT MultiMap - Interface IV

- remove(mm, k, v)
  - **descr:** removes a key value pair from the multimap
  - **pre:** $mm \in \mathcal{MM}$, $k - TKey$, $v - TValue$
  - **post:** $remove \leftarrow$
    $$\begin{cases} true, & \text{if } < k, v > \in mm, mm' \in \mathcal{MM}, mm' = mm - < k, v > \\ false, & \text{otherwise} \end{cases}$$

ADT MultiMap - Interface V

- search(mm, k, l)
  - **descr:** returns a list with all the values associated to a key
  - **pre:** $mm \in \mathcal{MM}$, $k - TKey$
  - **post:** $l \in \mathcal{L}$, l is the list of values associated to the key k. If k is not in the multimap, l is the empty list.

ADT MultiMap - Interface VI

- iterator(mm, it)
  - **descr:** returns an iterator over the multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** $it \in \mathcal{I}$, *it* is an iterator over *mm*, the current element from *it* is the first pair from *mm*, or, *it* is invalid if *mm* is empty

  - **Obs:** the iterator for a MultiMap is similar to the iterator for other containers, but the *getCurrent* operation returns a <key, value> pair.

ADT MultiMap - Interface VII

- size(mm)
  - **descr:** returns the number of pairs from the multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** $size \leftarrow$ the number of pairs from mm

ADT MultiMap - Interface VIII

- Other possible operations:
- keys(mm, s)
  - **descr:** returns the set of all keys from the multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** $s \in \mathcal{S}$, s is the set of all keys from mm

ADT MultiMap - Interface IX

- values(mm, b)
  - **descr:** returns the bag of all values from the multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** $b \in \mathcal{B}$, m b is a bag with all the values from mm

ADT MultiMap - Interface X

- pairs(mm, b)
  - **descr:** returns the bag of all pairs from the multimap
  - **pre:** $mm \in \mathcal{MM}$
  - **post:** $b \in \mathcal{B}$, b is a bag with all the pairs from mm

ADT SortedMultiMap

- We can have a MultiMap where we can define an order (a relation) on the set of possible keys. However, if a key has multiple values, the values can be in any order (we order the keys only, not the values) $\Rightarrow$ **ADT SortedMultiMap**
- The only change in the interface is for the *init* operation that will receive the *relation* as parameter.
- For a sorted MultiMap, the iterator has to iterate through the pairs in the order given by the *relation*, and the operations *keys* and *pairs* return SortedSet and SortedBag.

ADT MultiMap - representations

- There are several data structures that can be used to implement an ADT MultiMap (or ADT SortedMultiMap), the dynamic array being one of them (others will be discussed later):
- Regardless of the data structure used, there are two options to represent a MultiMap (sorted or not):
  - Store individual $<$ key, value $>$ pairs. If a key has multiple values, there will be multiple pairs containing this key. (R1)
  - Store unique keys and for each key store a *list* of associated values. (R2)

ADT MultiMap - R1

- For the example with the Morse code, we would have:

-.-..-- cat	-.-..- ca	-.-..- nna	.--...-.-. abc	-.-..- nnet

- Key is written with red and the value with black.
- Every element is one key - value pair.

ADT MultiMap - R2

- For the example with the Morse code, we would have:

-.-..-- [cat]	-.-..- [ca, nna, nnet]	.--...-.-. [abc]

- Key is written with red and the value with black.
- Every element is one key together with all the values belonging to it. The *list of values* can be another dynamic array, or a linked list, or any other data structure.

Source: https://medium.com/analytics-vidhya/coding-interview-question-c8f2520faf72

- If you had to take a plate, which one would you take?
- If you had to add another plate in the pile, where would you put it?

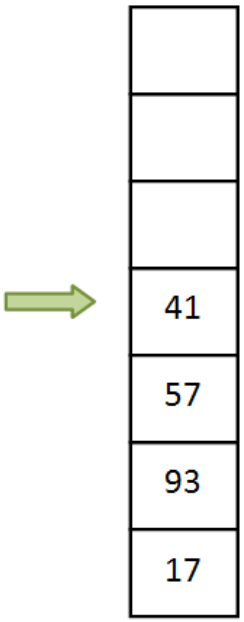
ADT Stack

- The ADT *Stack* represents a container in which access to the elements is restricted to one end of the container, called the *top* of the stack.
  - When a new element is added, it will automatically be added to the top.
  - When an element is removed the one from the top is automatically removed.
  - Only the element from the top can be accessed.

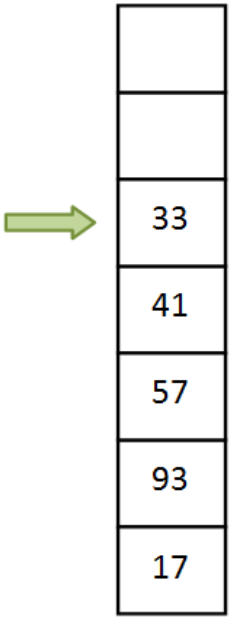
- Because of this restricted access, the stack is said to have a **LIFO** policy: **L**ast **I**n, **F**irst **O**ut (the last element that was added will be the first element that will be removed).

ADT Stack Example

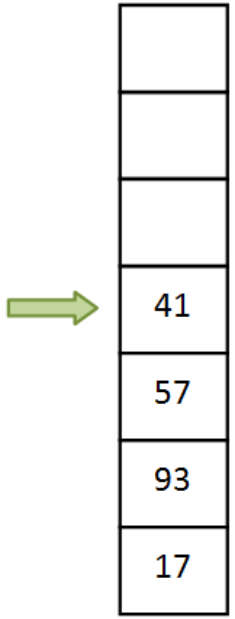
- Suppose that we have the following stack (green arrow shows the top of the stack):



- We *push* the number 33:

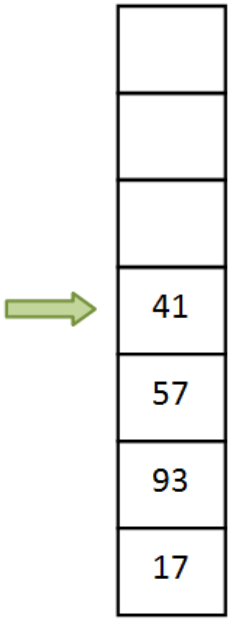


- We *pop* an element:

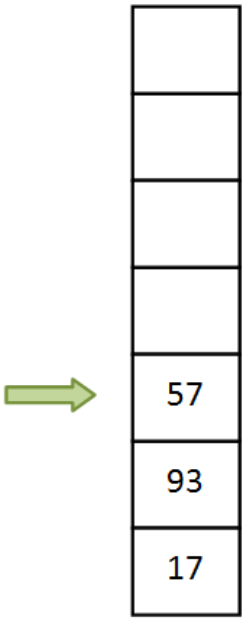


ADT Stack Example

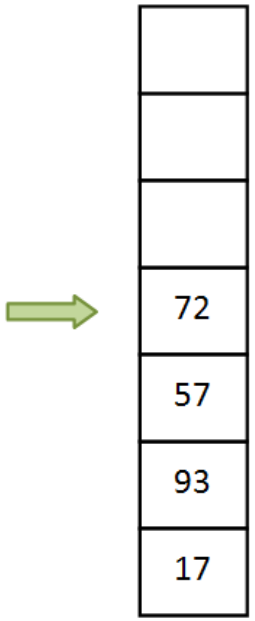
- This is our stack:



- We *pop* another element:



- We *push* the number 72:



ADT Stack - Interface I

- The domain of the ADT Stack:
  $\mathcal{S} = \{s | s \textit{ is a stack with elements of type } TElem\}$

- The interface of the ADT Stack contains the following operations:

ADT Stack - Interface II

- init(s)
  - **descr:** creates a new empty stack
  - **pre:** True
  - **post:** $s \in \mathcal{S}$, s is an empty stack

ADT Stack - Interface III

- destroy(s)
  - **descr:** destroys a stack
  - **pre:** $s \in \mathcal{S}$
  - **post:** s was destroyed

ADT Stack - Interface IV

- push(s, e)
  - **descr:** pushes (adds) a new element onto the stack
  - **pre:** $s \in \mathcal{S}$, e is a *TElem*
  - **post:** $s' \in \mathcal{S}$, $s' = s \oplus e$, e is the most recent element added to the stack

ADT Stack - Interface V

- pop(s)
  - **descr:** pops (removes) the most recent element from the stack
  - **pre:** $s \in \mathcal{S}$, s is not empty
  - **post:** $pop \leftarrow e$, e is a *TElem*, e is the most recent element from s, $s' \in \mathcal{S}$, $s' = s \ominus e$
  - **throws:** an *underflow* exception if the stack is empty

ADT Stack - Interface VI

- top(s)
  - **descr:** returns the most recent element from the stack (but it does not change the stack)
  - **pre:** $s \in \mathcal{S}$, s is not empty
  - **post:** $top \leftarrow e$, e is a *TElem*, e is the most recent element from s
  - **throws:** an *underflow* exception if the stack is empty

ADT Stack - Interface VII

- isEmpty(s)
  - **descr:** checks if the stack is empty (has no elements)
  - **pre:** $s \in \mathcal{S}$
  - **post:**

$$isEmpty \leftarrow \begin{cases} true, \ if \ s \ has \ no \ elements \\ false, \ otherwise \end{cases}$$

ADT Stack - Interface VIII

- **Note:** stacks cannot be iterated, so they don't have an *iterator* operation!

http:www.rgbstock.comphotomeZ8AhAQueue+Line

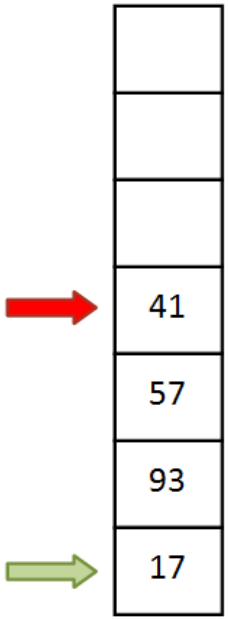
- Look at the queue above.
- If a new person arrives, where should he/she stand?
- When the blue person finishes, who is going to be the next at the desk?

ADT Queue

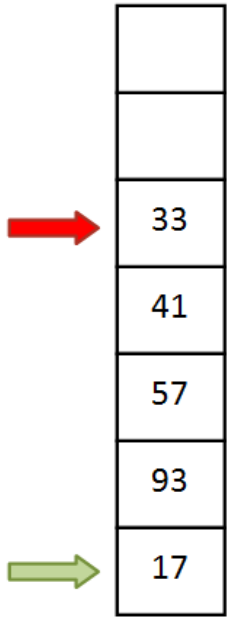
- The ADT Queue represents a container in which access to the elements is restricted to the two ends of the container, called *front* and *rear*.
  - When a new element is added (pushed), it has to be added to the *rear* of the queue.
  - When an element is removed (popped), it will be the one at the *front* of the queue.
- Because of this restricted access, the queue is said to have a **FIFO** policy: First In First Out.

ADT Queue - Example

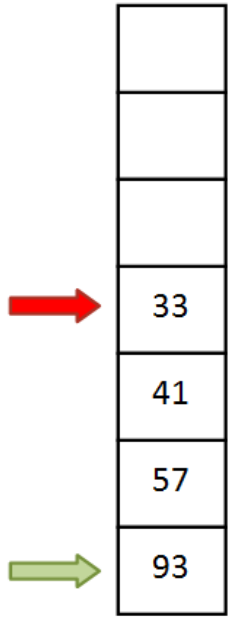
- Assume that we have the following queue (green arrow is the front, red arrow is the rear)



- Push number 33:

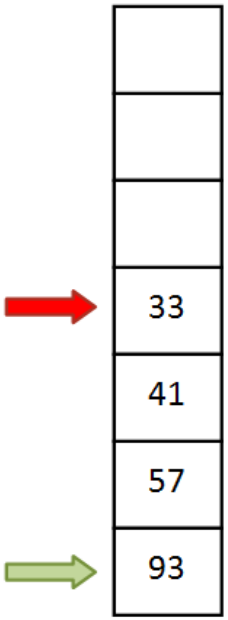


- Pop an element:

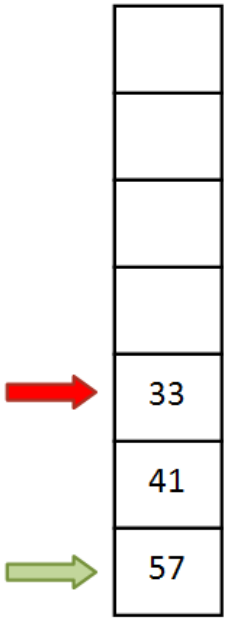


ADT Queue - Example

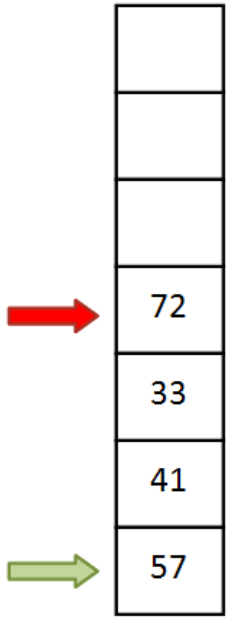
- This is our queue:



- Pop an element:



- Push number 72:



ADT Queue - Interface I

- The domain of the ADT Queue:
  $\mathcal{Q} = \{q | q \textit{ is a queue with elements of type } TElem\}$

- The interface of the ADT Queue contains the following operations:

ADT Queue - Interface II

- init(q)
  - **descr:** creates a new empty queue
  - **pre:** True
  - **post:** $q \in \mathcal{Q}$, q is an empty queue

ADT Queue - Interface III

- destroy(q)
  - **descr:** destroys a queue
  - **pre:** $q \in \mathcal{Q}$
  - **post:** q was destroyed

ADT Queue - Interface IV

- push(q, e)
  - **descr:** pushes (adds) a new element to the rear of the queue
  - **pre:** $q \in \mathcal{Q}$, e is a *TElem*
  - **post:** $q' \in \mathcal{Q}$, $q' = q \oplus e$, e is the element at the rear of the queue

ADT Queue - Interface V

- pop(q)
  - **descr:** pops (removes) the element from the front of the queue
  - **pre:** $q \in \mathcal{Q}$, q is not empty
  - **post:** $pop \leftarrow e$, e is a *TElem*, e is the element at the front of q, $q' \in \mathcal{Q}$, $q' = q \ominus e$
  - **throws:** an *underflow* exception if the queue is empty

ADT Queue - Interface VI

- top(q)
  - **descr:** returns the element from the front of the queue (but it does not change the queue)
  - **pre:** $q \in \mathcal{Q}$, q is not empty
  - **post:** $top \leftarrow e$, e is a *TElem*, e is the element from the front of q
  - **throws:** an *underflow* exception if the queue is empty

ADT Queue - Interface VII

- isEmpty(s)
  - **descr:** checks if the queue is empty (has no elements)
  - **pre:** $q \in \mathcal{Q}$
  - **post:**

$$isEmpty \leftarrow \begin{cases} true, \ if \ q \ has \ no \ elements \\ false, \ otherwise \end{cases}$$

ADT Queue - Interface VIII

- **Note:** queues cannot be iterated, so they do not have an *iterator* operation!

ADT Queue - Representation

- What data structures can be used to implement a Queue?
  - Static Array - for a fixed capacity Queue
    - In this case an *isFull* operation can be added, and *push* can also throw an exception if the Queue is full.
  - Dynamic Array
  - other data structures (will be discussed later)

ADT Queue - Array-based representation

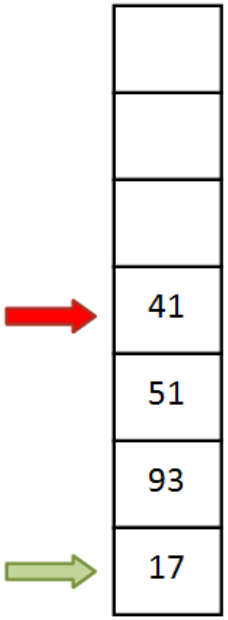
- If we want to implement a Queue using an array (static or dynamic), where should we place the *front* and the *rear* of the queue?
- In theory, we have two options:
  - Put *front* at the beginning of the array and *rear* at the end
  - Put *front* at the end of the array and *rear* at the beginning
- In either case we will have one operation (push or pop) that will have $\Theta(n)$ complexity.

ADT Queue - Array-based representation

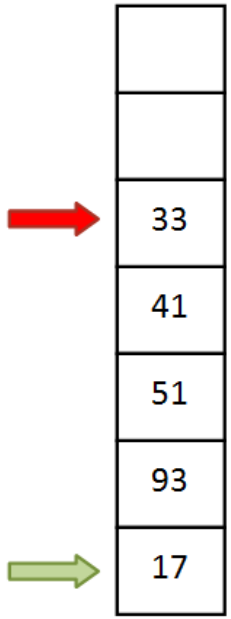
- We can improve the complexity of the operations, if we do not insist on having either *front* or *rear* at the beginning of the array (at position 1).

ADT Queue - Array-based representation

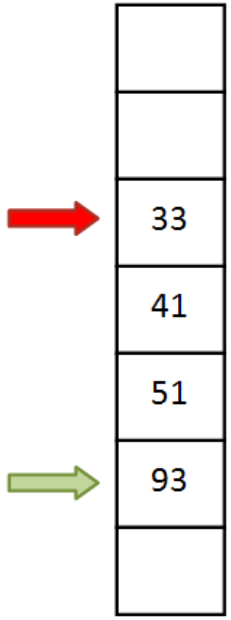
- This is our queue (green arrow is the front, red arrow is the rear)



- Push number 33:

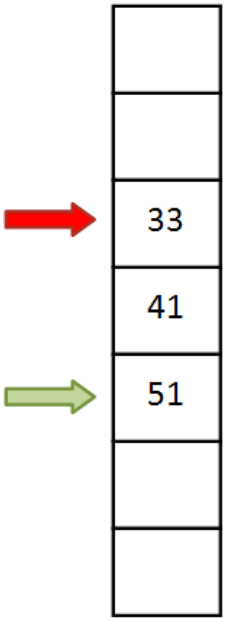


- Pop an element (and do not move the other elements):

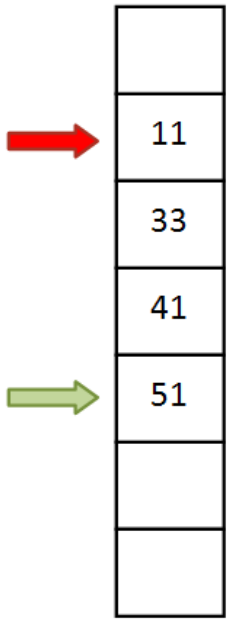


ADT Queue - Array-based representation

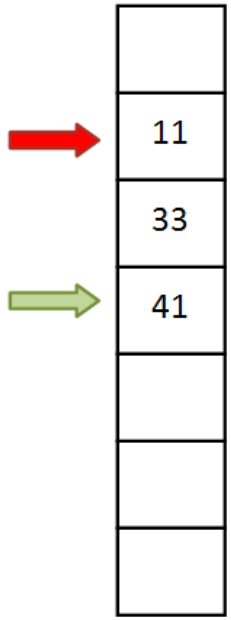
- Pop another element:



- Push number 11:

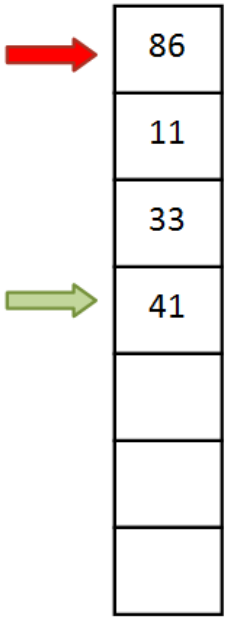


- Pop an element:

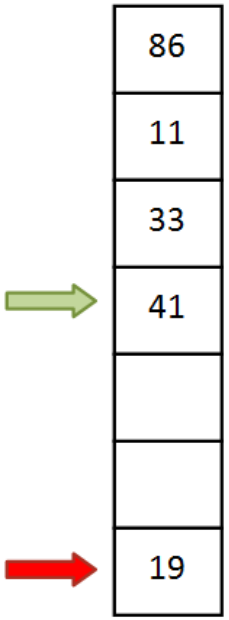


ADT Queue - Array-based representation

- Push number 86:



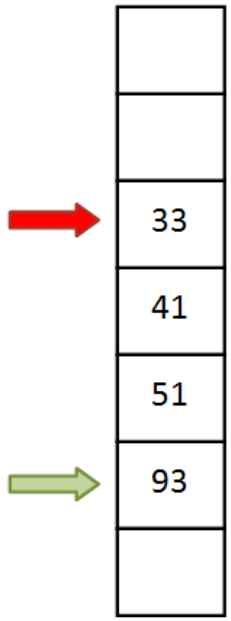
- Push number 19:

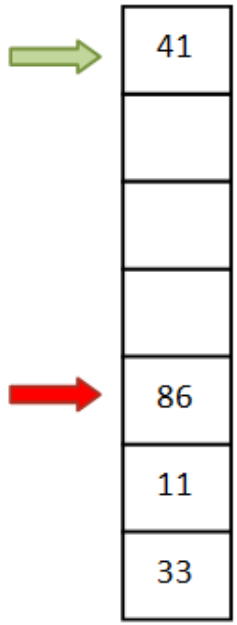


- This is called a **circular array**

ADT Queue - representation on a circular array - pop

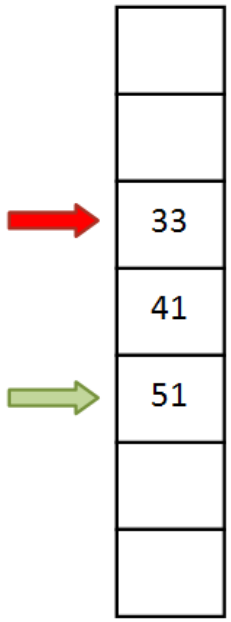
- There are two situations for our queue (green arrow is the front where we pop from):

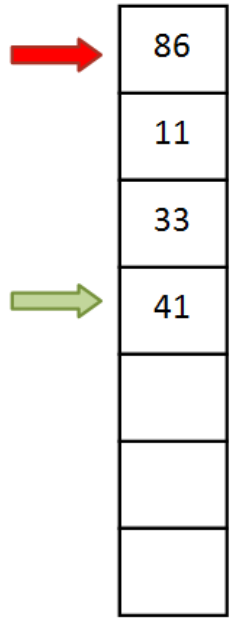




ADT Queue - representation on a circular array - push

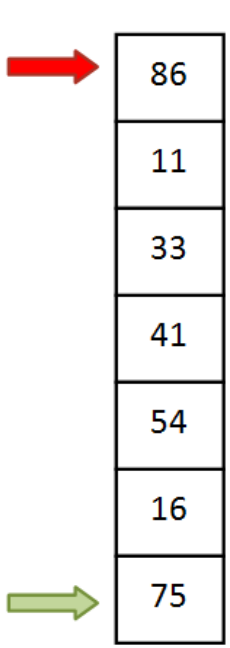
- There are two situations for our queue (red arrow is the end where we push):

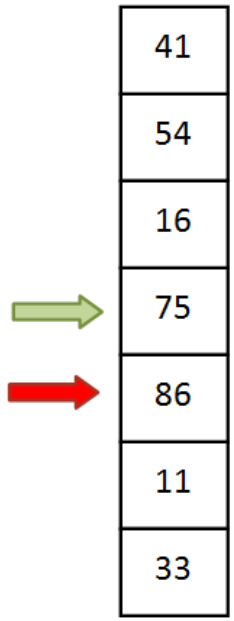




Queue - representation on a circular array - push

- When pushing a new element we have to check whether the queue is full

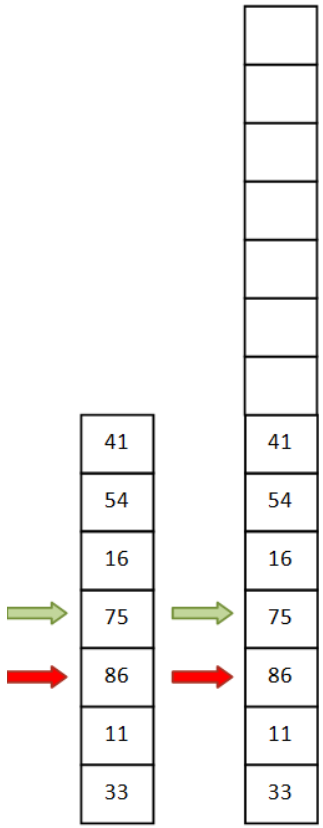




- For both example, the elements were added in the order: 75, 16, 54, 41, 33, 11, 86

ADT Queue - representation on a circular array - push

- If we have a dynamic array-based representation and the array is full, we have to allocate a larger array and copy the existing elements (as we always do with dynamic arrays)
- But we have to be careful how we copy the elements in order to avoid having something like:



Summary

- Today we have talked about:
  - ADT Bag - Representation R4
  - ADT Matrix - and possible representations on a dynamic array
  - ADT Set
  - ADT Map and ADT MultiMap
  - ADT SortedBag, ADT SortedSet, ADT SortedMap and ADT SortedMultiMap
  - ADT Stack, ADT Queue
- Extra Reading - did not have time to make one :(