DATA STRUCTURES AND ALGORITHMS

LECTURE 4

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

In Lecture 3...

- Iterators
- Containers
  - ADT Bag
  - ADT Sorted Bag
  - ADT Set and ADT SortedSet

Today

- Containers

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 $+ (i * M + j) * \textit{size of an element}$
- 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.
- 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

	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

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

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

- 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

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

- 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

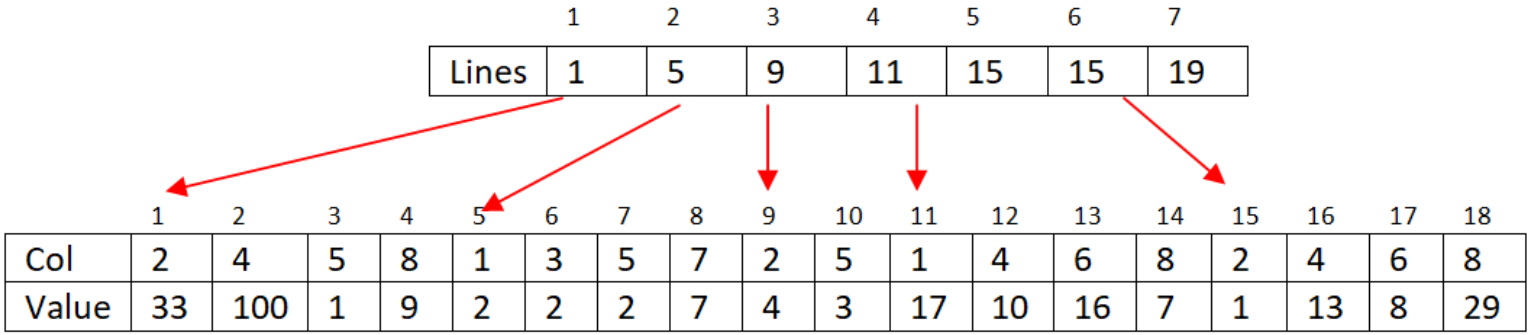
- Modify the value from position (3, 3) to 19

	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

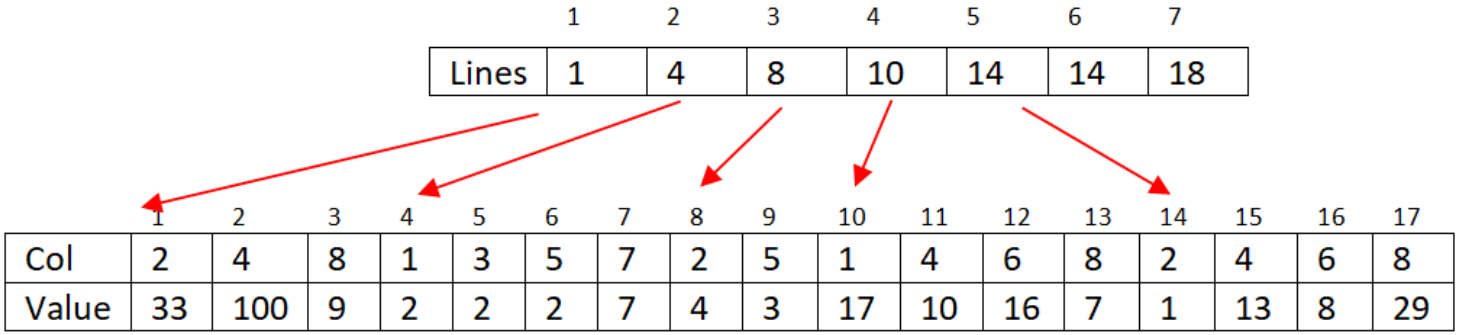
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Col	2	4	5	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	1	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

Sparse Matrix - R2 example

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

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

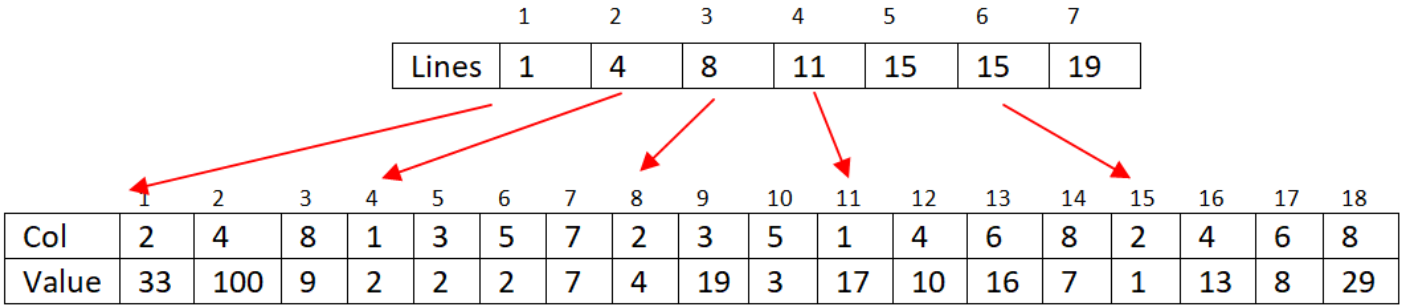
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Col	2	4	8	1	3	5	7	2	5	1	4	6	8	2	4	6	8
Value	33	100	9	2	2	2	7	4	3	17	10	16	7	1	13	8	29

Sparse Matrix - R2 example

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

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



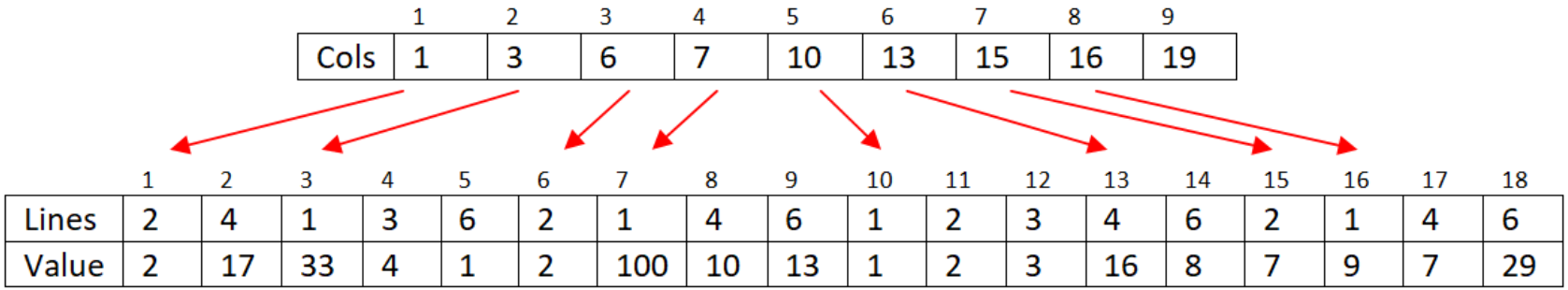
	1	2	3	4	5	6	7
Lines	1	4	8	11	15	15	19

	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

Source: https://clipart.wpblink.com/wallpaper-1911442

- Consider the above figure: if you had to add a new plate to the pile, where would you put it?
- If you had to remove a plate, which one would you take?

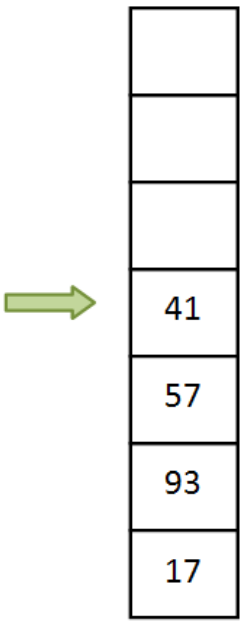
ADT Stack

- 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

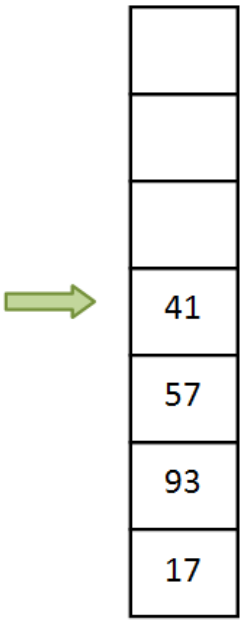
- Suppose that we have the following stack (green arrow shows the top of the stack):



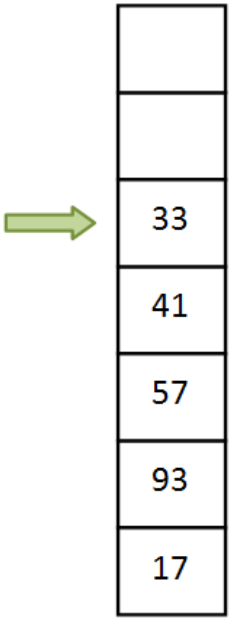
- We *push* the number 33:

ADT Stack Example

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



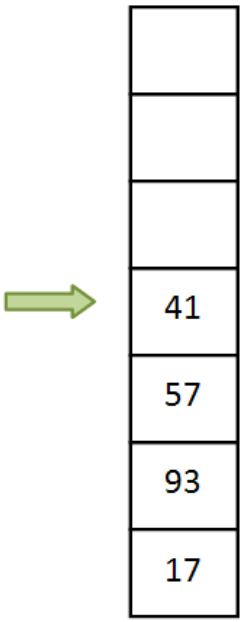
- We *push* the number 33:



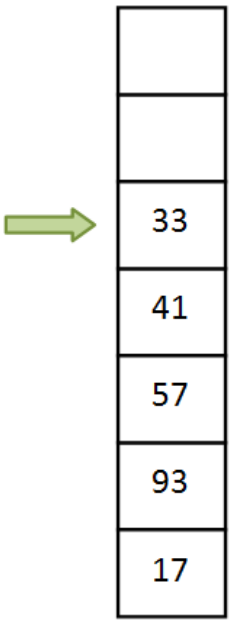
- We *pop* an element:

ADT Stack Example

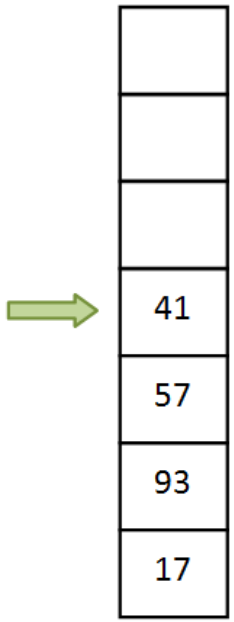
- Suppose that we have the following stack (green arrow shows the top of the stack):



- We *push* the number 33:

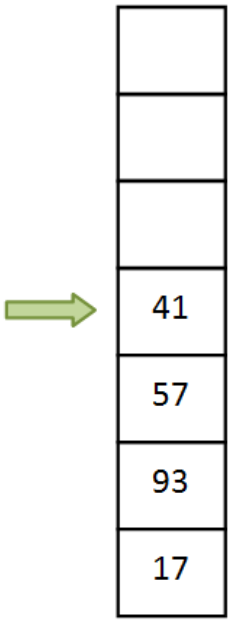


- We *pop* an element:



ADT Stack Example

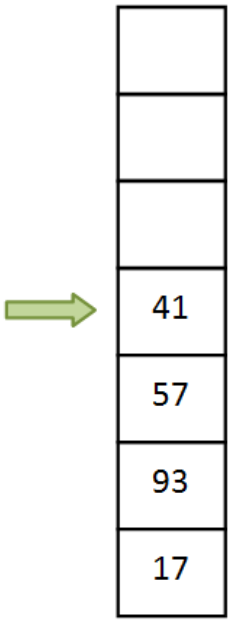
- This is our stack:



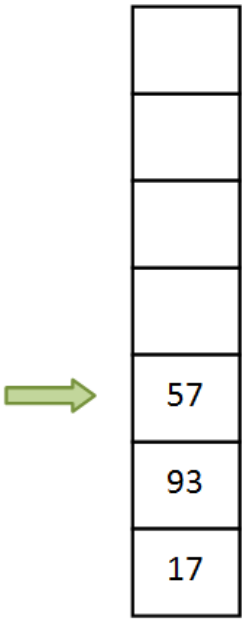
- We *pop* another element:

ADT Stack Example

- This is our stack:



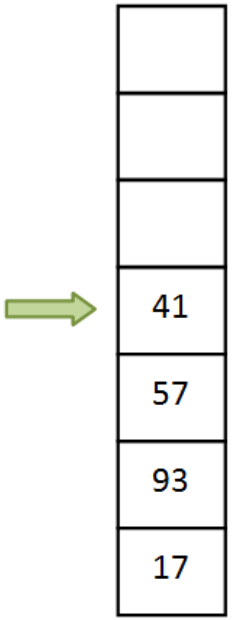
- We *pop* another element:



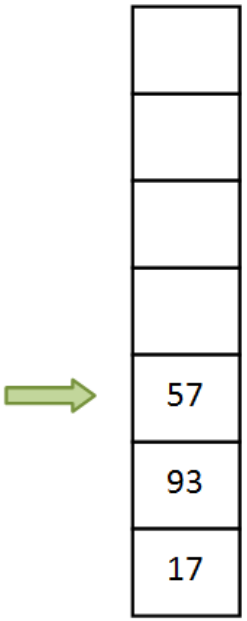
- We *push* the number 72:

ADT Stack Example

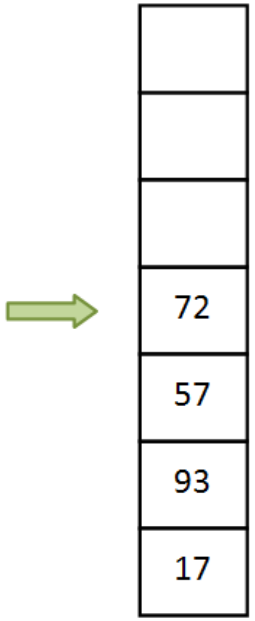
- This is our stack:



- 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

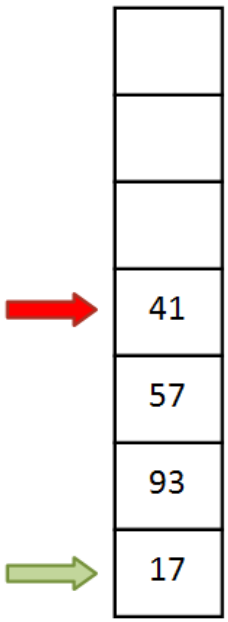
- Look at the queue above.
- If a new person arrives, where should he/she stand?
- When the blue person finishes, who is going to be the next at the desk?

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

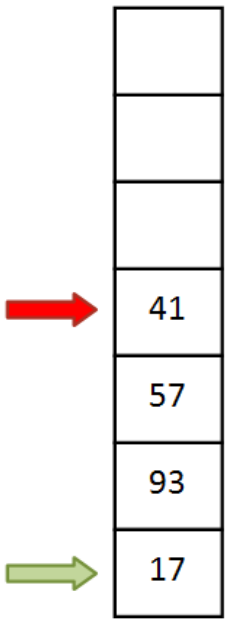
- Assume that we have the following queue (green arrow is the front, red arrow is the rear)



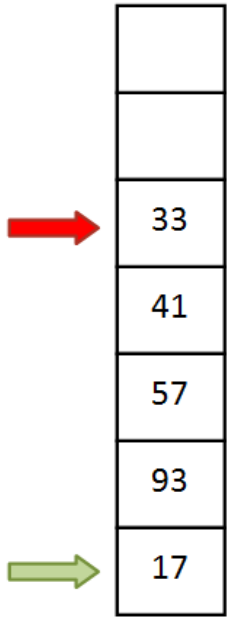
- Push number 33:

ADT Queue - Example

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



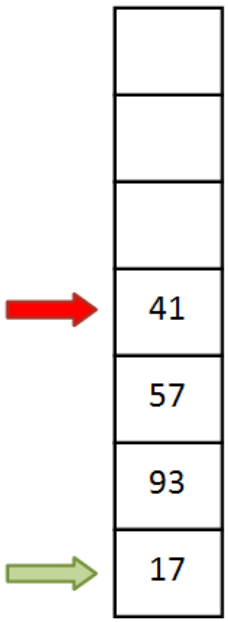
- Push number 33:



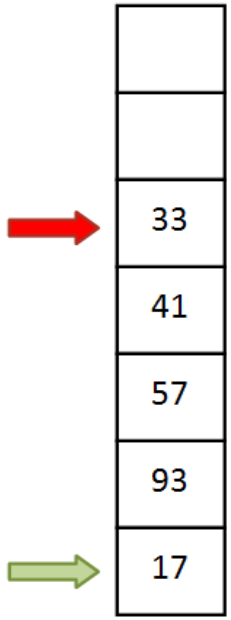
- Pop an element:

ADT Queue - Example

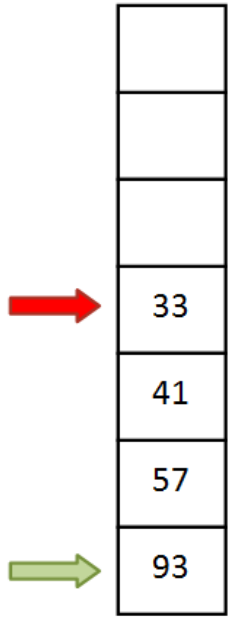
- Assume that we have the following queue (green arrow is the front, red arrow is the rear)



- Push number 33:

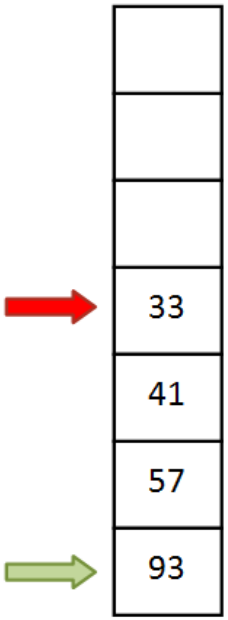


- Pop an element:



ADT Queue - Example

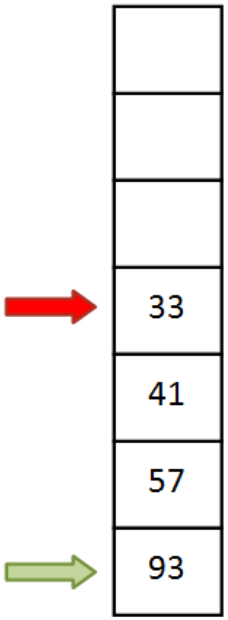
- This is our queue:



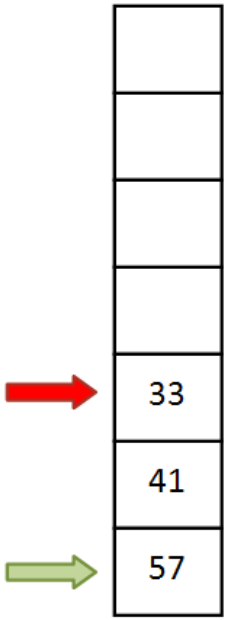
- Pop an element:

ADT Queue - Example

- This is our queue:



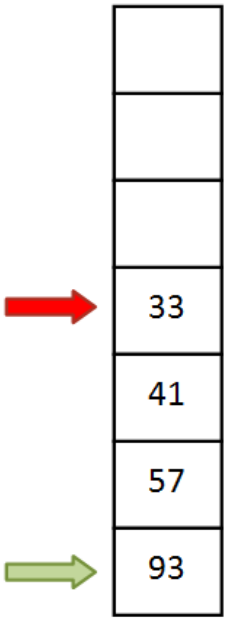
- Pop an element:



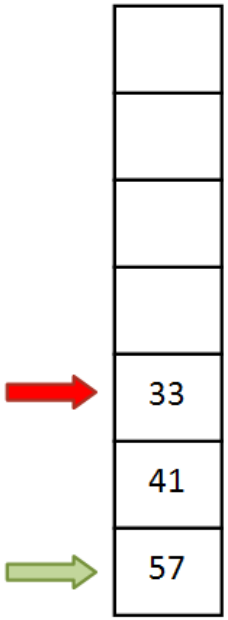
- Push number 72:

ADT Queue - Example

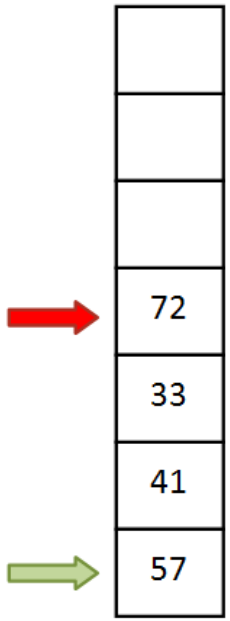
- This is our queue:



- 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?

ADT Queue - Array-based representation

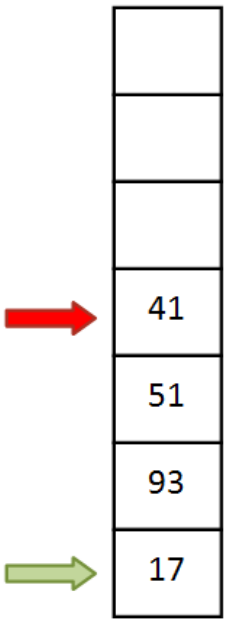
- If we want to implement a Queue using an array (static or dynamic), where should we place the *front* and the *rear* of the queue?
- In theory, we have two options:
  - Put *front* at the beginning of the array and *rear* at the end
  - Put *front* at the end of the array and *rear* at the beginning
- 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

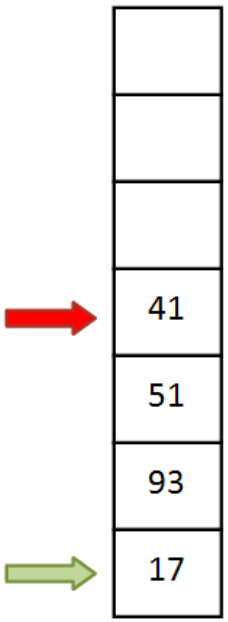
- This is our queue (green arrow is the front, red arrow is the rear)



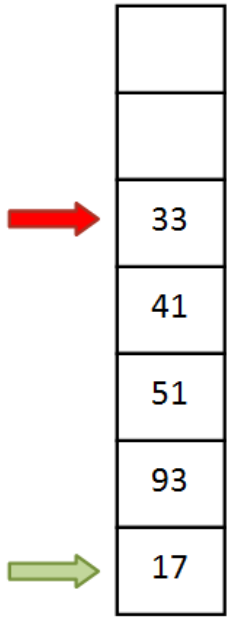
- Push number 33:

ADT Queue - Array-based representation

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



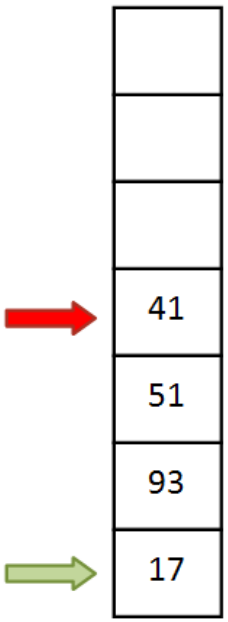
- Push number 33:



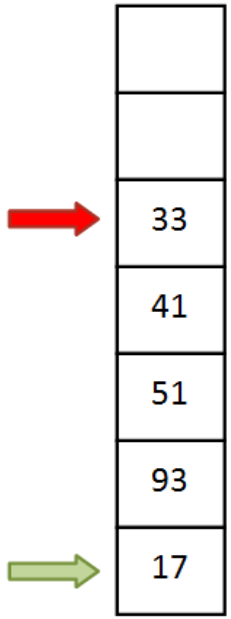
- Pop an element (and do not move the other elements):

ADT Queue - Array-based representation

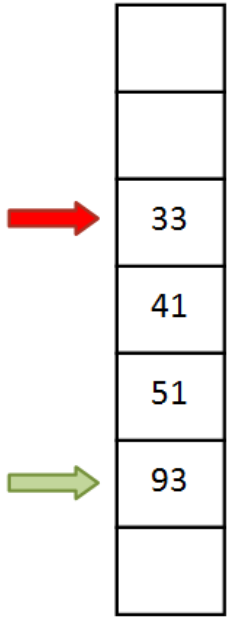
- This is our queue (green arrow is the front, red arrow is the rear)



- Push number 33:



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

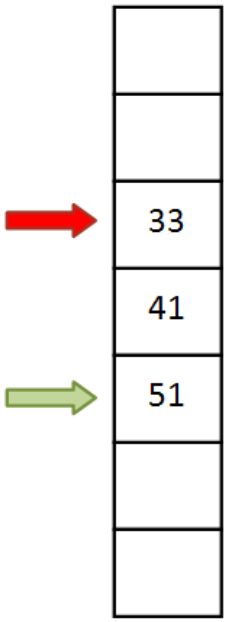


ADT Queue - Array-based representation

- Pop another element:

ADT Queue - Array-based representation

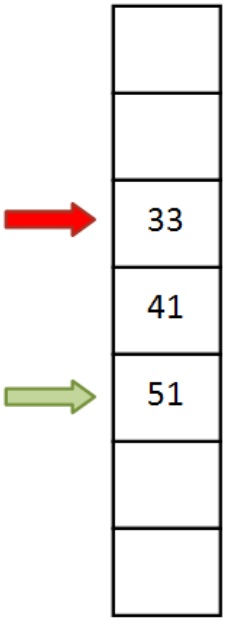
- Pop another element:



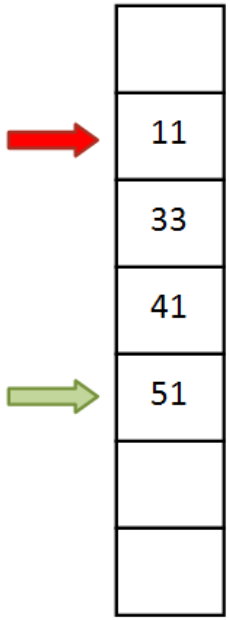
- Push number 11:

ADT Queue - Array-based representation

- Pop another element:



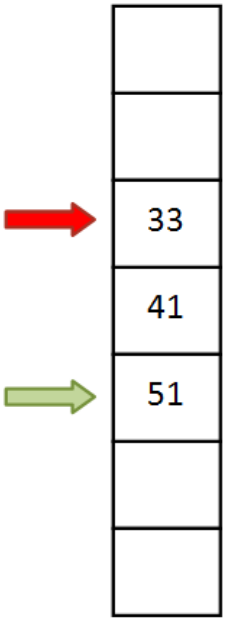
- Push number 11:



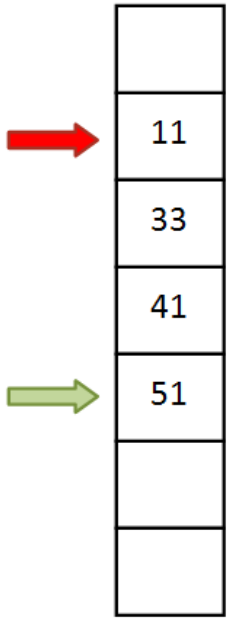
- Pop an element:

ADT Queue - Array-based representation

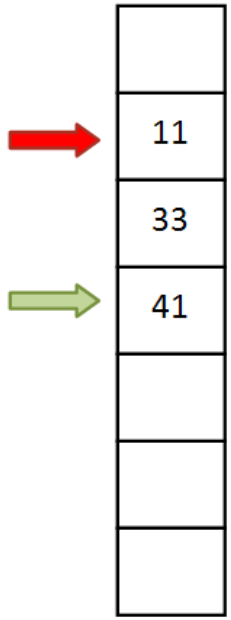
- Pop another element:



- Push number 11:



- Pop an element:

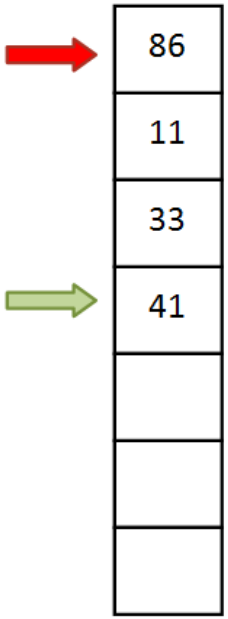


ADT Queue - Array-based representation

- Push number 86:

ADT Queue - Array-based representation

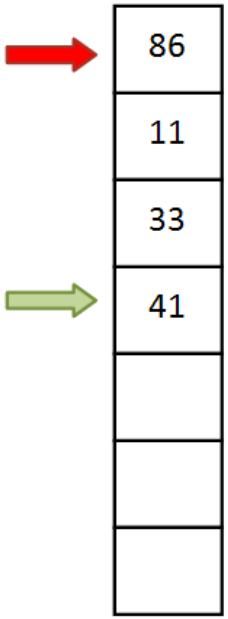
- Push number 86:



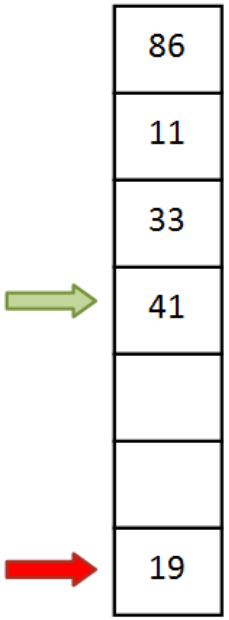
- Push number 19:

ADT Queue - Array-based representation

- Push number 86:



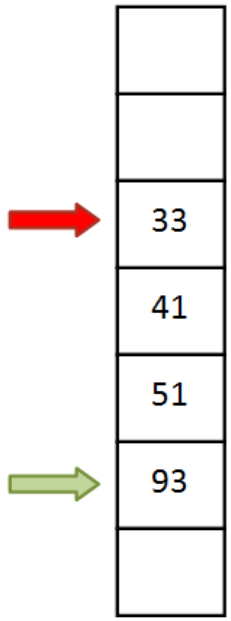
- Push number 19:

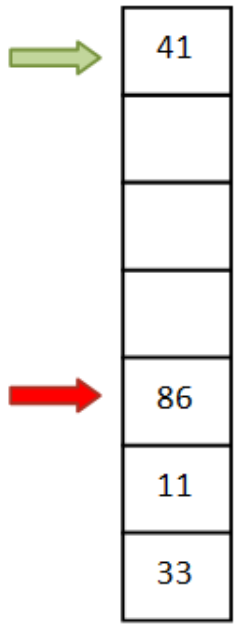


- This is called a **circular array**

ADT Queue - representation on a circular array - pop

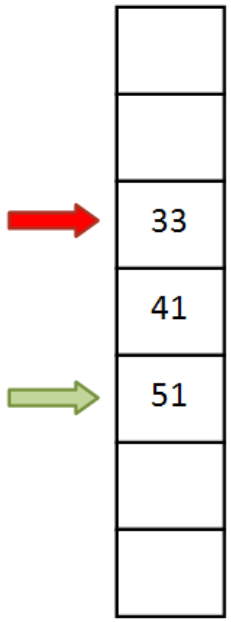
- There are two situations for our queue (green arrow is the front where we pop from):

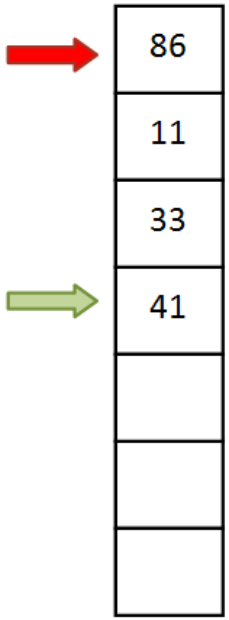




ADT Queue - representation on a circular array - push

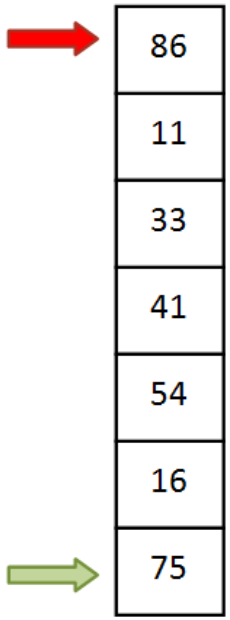
- There are two situations for our queue (red arrow is the end where we push):

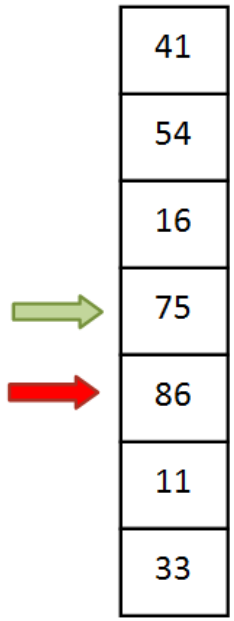




Queue - representation on a circular array - push

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

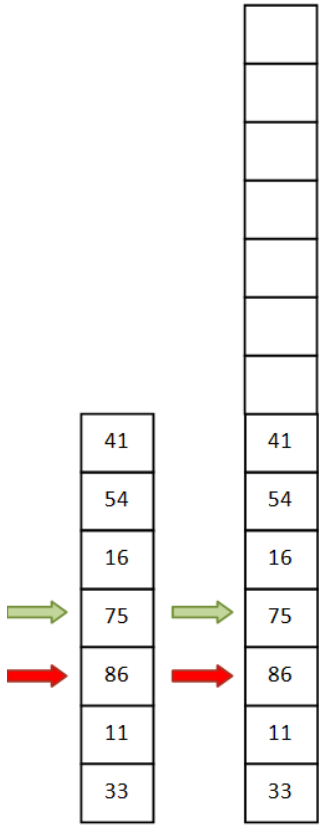




- For both example, the elements were added in the order: 75, 16, 54, 41, 33, 11, 86

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:



Source: https://www.vectorstock.com/royalty-free-vector/patients-in-doctors-waiting-room-at-the-hospital-vector-12041494

- Consider the following queue in front of the Emergency Room. Who should be the next person checked by the doctor?

ADT Priority Queue

- The ADT Priority Queue is a container in which each element has an associated *priority* (of type *TPriority*).
- In a Priority Queue access to the elements is restricted: we can access only the element with the highest priority.
- Because of this restricted access, we say that the Priority Queue works based on a **HPF - Highest Priority First** policy.

ADT Priority Queue

- In order to work in a more general manner, we can define a relation $\mathcal{R}$ on the set of priorities: $\mathcal{R} : \mathit{TPriority} \times \mathit{TPriority}$
- When we say *the element with the highest priority* we will mean that the highest priority is determined using this relation $\mathcal{R}$.
- If the relation $\mathcal{R} = "\geq"$, the element with the *highest priority* is the one for which the value of the priority is the largest (maximum).
- Similarly, if the relation $\mathcal{R} = "\leq"$, the element with the *highest priority* is the one for which the value of the priority is the lowest (minimum).

Priority Queue - Interface I

- The domain of the ADT Priority Queue:
  $\mathcal{PQ} = \{pq | pq$ is a priority queue with elements $(e, p), e \in TElem, p \in TPriority\}$

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

Priority Queue - Interface II

- init (pq, R)
  - **descr:** creates a new empty priority queue
  - **pre:** R is a relation over the priorities, $R : TPriority \times TPriority$
  - **post:** $pq \in \mathcal{PQ}$, pq is an empty priority queue

Priority Queue - Interface III

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

Priority Queue - Interface IV

- push(pq, e, p)
  - **descr:** pushes (adds) a new element to the priority queue
  - **pre:** $pq \in \mathcal{PQ}, e \in TElem, p \in TPriority$
  - **post:** $pq' \in \mathcal{PQ}, pq' = pq \oplus (e, p)$

Priority Queue - Interface V

- pop (pq)
  - **descr:** pops (removes) from the priority queue the element with the highest priority. It returns both the element and its priority
  - **pre:** $pq \in \mathcal{PQ}$, pq is not empty
  - **post:** $pop \leftarrow (e, p)$, $e \in TElem, p \in TPriority$, e is the element with the highest priority from pq, p is its priority. $pq' \in \mathcal{PQ}, pq' = pq \ominus (e, p)$
  - **throws:** an exception if the priority queue is empty.

Priority Queue - Interface VI

- top (pq)
  - **descr:** returns from the priority queue the element with the highest priority and its priority. It does not modify the priority queue.
  - **pre:** $pq \in \mathcal{PQ}$, pq is not empty
  - **post:** $top \leftarrow (e, p)$, $e \in TElem, p \in TPriority$, e is the element with the highest priority from pq, p is its priority.
  - **throws:** an exception if the priority queue is empty.

Priority Queue - Interface VII

- isEmpty(pq)
  - **Description:** checks if the priority queue is empty (it has no elements)
  - **Pre:** $pq \in \mathcal{PQ}$
  - **Post:**

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

Priority Queue - Interface VIII

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

- Consider the following problem: *we have a text and want to find the word that appears most frequently in this text.* What would be the characteristics of the container used for this problem?

- Consider the following problem: *we have a text and want to find the word that appears most frequently in this text.* 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.