DATA STRUCTURES AND ALGORITHMS

LECTURE 3

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2018 - 2019

In Lecture 2...

- Algorithm Analysis
- Dynamic Array

Today

1. Dynamic Array
2. Iterators

Dynamic Array

- There are arrays whose size can grow or shrink, depending on the number of elements that need to be stored in the array: they are called *dynamic arrays* (or *dynamic vectors*).
- Dynamic arrays are still arrays, the elements are still kept at contiguous memory locations and we still have the advantage of being able to compute the address of every element in $\Theta(1)$ time.

Dynamic Array - Representation

- In general, for a Dynamic Array we need the following fields:
  - *cap* - denotes the number of slots allocated for the array (its capacity)
  - *len* - denotes the actual number of elements stored in the array
  - *elems* - denotes the actual array with *capacity* slots for TElems allocated

DynamicArray:
cap: Integer
len: Integer
elems: TElem[]

Dynamic Array - DS vs. ADT

- Dynamic Array is a data structure:
  - It describes how data is actually stored in the computer (in a single contiguous memory block) and how it can be accessed and processed
  - It can be used as representation to implement different abstract data types
- However, Dynamic Array is so frequently used that in most programming languages it exists as a separate container as well.
  - The Dynamic Array is not really an ADT, since it has one single possible implementation, but we still can treat is as an ADT, and discuss its interface.

Dynamic Array - Interface I

- **Domain** of ADT DynamicArray

$$\mathcal{DA} = \{\mathbf{da} | da = (cap, len, e_1 e_2 e_3 ... e_{len}), cap \in N^*, len \in N, len \leq cap, e_i \text{ is of type TElem}\}$$

Dynamic Array - Interface II

- What operations should we have in the **interface** of the Dynamic Array ADT?

Dynamic Array - Interface III

- init(da, cp)
  - **description:** creates a new, empty DynamicArray with initial capacity *cp* (constructor)
  - **pre:** $cp \in N^*$
  - **post:** $da \in \mathcal{DA},\ da.cap = cp,\ da.n = 0$
  - **throws:** an exception if *cp* is negative or zero

Dynamic Array - Interface IV

- destroy(da)
  - **description:** destroys a DynamicArray (destructor)
  - **pre:** $da \in \mathcal{DA}$
  - **post:** *da* was destroyed (the memory occupied by the dynamic array was freed)

Dynamic Array - Interface V

- size(da)
  - **description:** returns the size (number of elements) of the DynamicArray
  - **pre:** $da \in \mathcal{DA}$
  - **post:** size $\leftarrow$ the size of da (the number of elements)

Dynamic Array - Interface VI

- getElement(da, i)
  - **description:** returns the element from a position from the DynamicArray
  - **pre:** $da \in \mathcal{DA},\ 1 \leq i \leq da.len$
  - **post:** $getElement \leftarrow e, e \in TElem,\ e = da.e_i$ (the element from position i)
  - **throws:** an exception if i is not a valid position

Dynamic Array - Interface VII

- setElement(da, i, e)
  - **description:** changes the element from a position to another value
  - **pre:** $da \in \mathcal{DA},\ 1 \leq i \leq da.len,\ e \in TElem$
  - **post:** $da' \in \mathcal{DA}, da'.e_i = e$ (the i^{th} element from *da'* becomes e), setElement $\leftarrow da.e_i$ (returns the old value from position i)
  - **throws:** an exception if i is not a valid position

Dynamic Array - Interface VIII

- addToEnd(da, e)
  - **description:** adds an element to the end of a DynamicArray. If the array is full, its capacity will be increased
  - **pre:** $da \in \mathcal{DA},\ e \in TElem$
  - **post:** $da' \in \mathcal{DA},\ da'.len = da.len + 1; da'.e_{da'.len+1} = e\ (da.cap = da.len \Rightarrow da'.cap \leftarrow da.cap * 2)$

Dynamic Array - Interface IX

- addToPosition(da, i, e)
  - **description:** adds an element to a given position in the DynamicArray. If the array is full, its capacity will be increased
  - **pre:** $da \in \mathcal{DA},\ 1 \leq i \leq da.len + 1,\ e \in TElem$
  - **post:** $da' \in \mathcal{DA},\ da'.len = da.len + 1,\ da'.e_i = da.e_{i-1} \forall j = da'.len, da'.len - 1, ..., i + 1, da'.e_i = e\ (da.cap = da.len \Rightarrow da'.cap \leftarrow da.cap * 2)$
  - **throws:** an exception if i is not a valid position (da.len+1 is a valid position when adding a new element)

Dynamic Array - Interface X

- deleteFromPosition(da, i)
  - **description:** deletes an element from a given position from the DynamicArray. Returns the deleted element
  - **pre:** $da \in \mathcal{DA},\ 1 \leq i \leq da.len$
  - **post:** $deleteFromPosition \leftarrow e, e \in TElem,\ e = da.e_i,\ da' \in \mathcal{DA},\ da'.len = da.len - 1,\ da'.e_j = da.e_{j+1} \forall i \leq j \leq da'.len$
  - **throws:** an exception if i is not a valid position

Dynamic Array - Interface XI

- iterator(da, it)
  - **description:** returns an iterator for the DynamicArray
  - **pre:** $da \in \mathcal{DA}$
  - **post:** $it \in \mathcal{I}$, it is an iterator over da

Dynamic Array - Interface XII

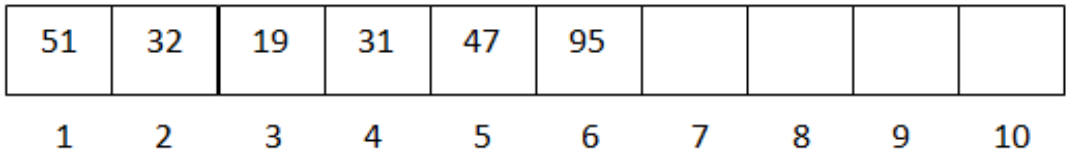
- Other possible operations:
  - Delete all elements from the Dynamic Array (make it empty)
  - Verify if the Dynamic Array is empty
  - Delete an element (given as element, not as position)
  - Check if an element appears in the Dynamic Array
  - etc.

Dynamic Array - Implementation

- Most operations from the interface of the Dynamic Array are very simple to implement.
- In the following we will discuss the implementation of three operations: *addToEnd*, *addToPosition* and *deleteFromPosition*
- For the implementation we are going to use the representation discussed earlier:

DynamicArray:
cap: Integer
len: Integer
elems: TElem[]

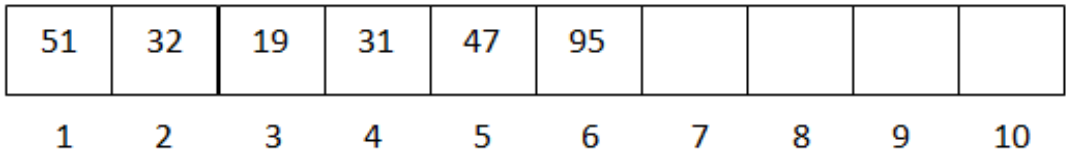
Dynamic Array - addToEnd - Case 1



- capacity (cap): 10
- length (len): 6

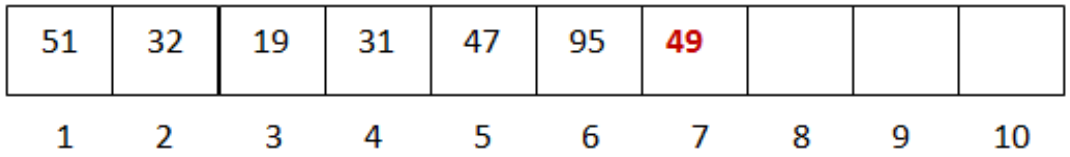
- Add the element 49 to the end of the dynamic array

Dynamic Array - addToEnd - Case 1



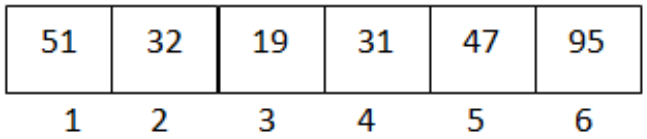
- capacity (cap): 10
- length (len): 6

- Add the element 49 to the end of the dynamic array



- capacity (cap): 10
- length (len): 7

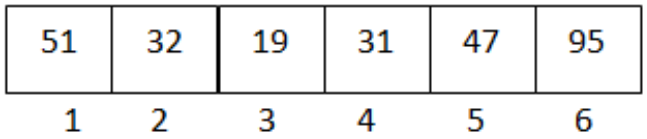
Dynamic Array - addToEnd - Case 2



- capacity (cap): 6
- length (len): 6

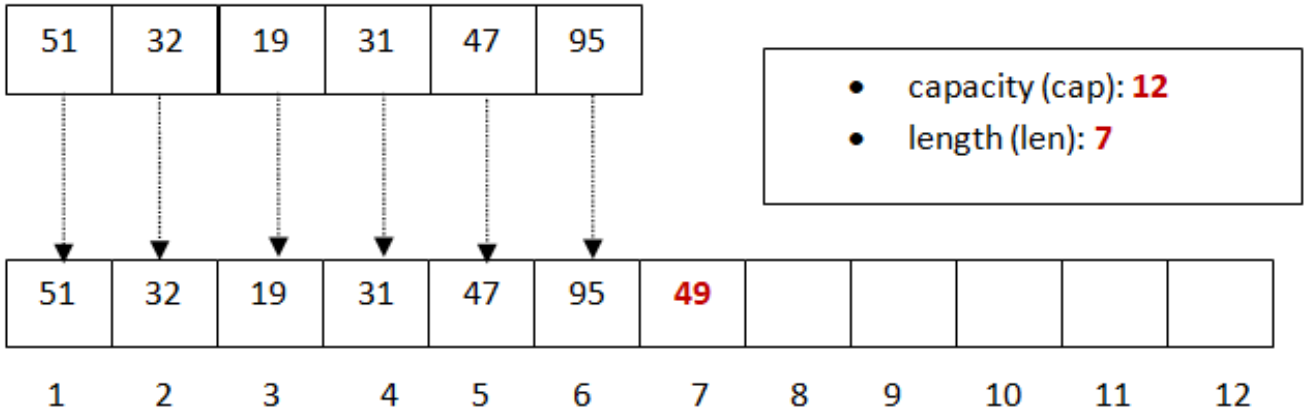
- **Add the element 49 to the end of the dynamic array**

Dynamic Array - addToEnd - Case 2



- capacity (cap): 6
- length (len): 6

- **Add the element 49 to the end of the dynamic array**

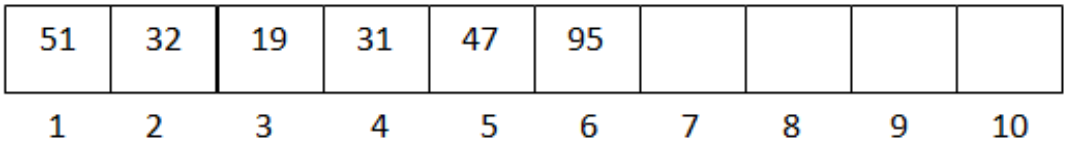


- capacity (cap): **12**
- length (len): **7**

Dynamic Array - addToEnd

```
subalgorithm addToEnd (da, e) is:
    if da.len = da.cap then
    //the dynamic array is full. We need to resize it
        da.cap ← da.cap * 2
        newElems ← @ an array with da.cap empty slots
        //we need to copy existing elements into newElems
        for index ← 1, da.len execute
            newElems[index] ← da.elems[index]
        end-for
        //we need to replace the old element array with the new one
        //depending on the prog. lang., we may need to free the old elems array
        da.elems ← newElems
    end-if
    //now we certainly have space for the element e
    da.len ← da.len + 1
    da.elems[da.len] ← e
end-subalgorithm
```

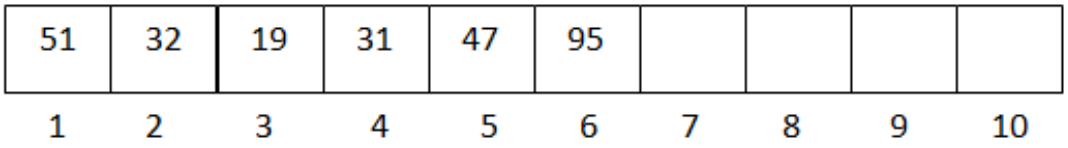
Dynamic Array - addToPosition



- capacity (cap): 10
- length (len): 6

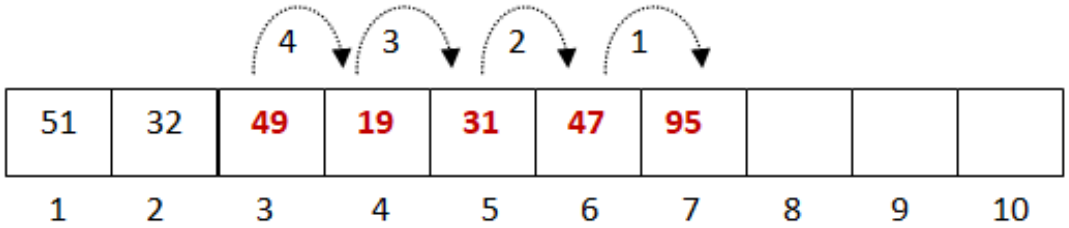
- **Add the element 49 to position 3**

Dynamic Array - addToPosition



- capacity (cap): 10
- length (len): 6

- **Add the element 49 to position 3**



- capacity (cap): 10
- length (len): **7**

- **Add the element 49 to position 3**

Dynamic Array - addToPosition

```
subalgorithm addToPosition (da, i, e) is:
    if i > 0 and i ≤ da.len+1 then
        if da.len = da.cap then //the dynamic array is full. We need to resize it
            da.cap ← da.cap * 2
            newElems ← @ an array with da.cap empty slots
            for index ← 1, da.len execute
                newElems[index] ← da.elems[index]
            end-for
            da.elems ← newElems
        end-if //now we certainly have space for the element e
        da.len ← da.len + 1
        for index ← da.len, i+1, -1 execute //move the elements to the right
            da.elems[index] ← da.elems[index-1]
        end-for
        da.elems[i] ← e
    else
        @throw exception
    end-if
end-subalgorithm
```

Dynamic Array

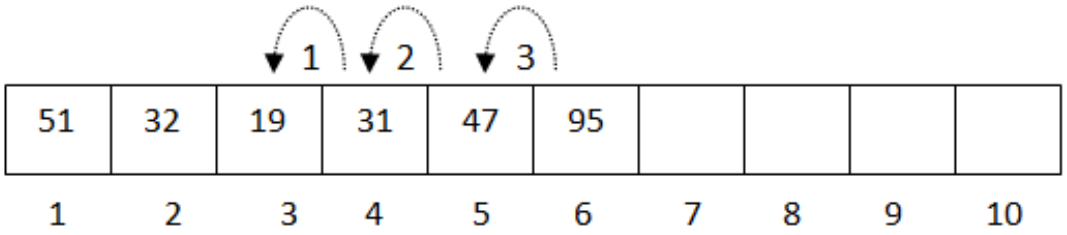
- Observations:
  - While it is not mandatory to double the capacity, it is important to define the new capacity as a product of the old one with a constant number greater than 1 (just adding one new slot, or a constant number of new slots is not OK - you will see later why).
  - After a resize operation the elements of the Dynamic Array will still occupy a contiguous memory zone, but it will be a different one than before.

Dynamic Array

```
Address of the Dynamic Array structure: 00D3FE00 13893120
  Length is: 3 si capacitate: 3
  Address of array from DA: 0039E568 3794280
     Address of element from position 0 0039E568 3794280
     Address of element from position 1 0039E56C 3794284
     Address of element from position 2 0039E570 3794288
------------------------------
Address of the Dynamic Array structure: 00D3FE00 13893120
  Length is: 6 si capacitate: 6
  Address of array from DA: 003A0100 3801344
     Address of element from position 0 003A0100 3801344
     Address of element from position 1 003A0104 3801348
     Address of element from position 2 003A0108 3801352
     Address of element from position 3 003A010C 3801356
     Address of element from position 4 003A0110 3801360
     Address of element from position 5 003A0114 3801364
------------------------------
Address of the Dynamic Array structure: 00D3FE00 13893120
  Length is: 8 si capacitate: 12
  Address of array from DA: 00396210 3760656
     Address of element from position 0 00396210 3760656
     Address of element from position 1 00396214 3760660
     Address of element from position 2 00396218 3760664
     Address of element from position 3 0039621C 3760668
     Address of element from position 4 00396220 3760672
     Address of element from position 5 00396224 3760676
     Address of element from position 6 00396228 3760680
     Address of element from position 7 0039622C 3760684
```

Dynamic Array - delete operation

- When we want to delete an element from a given position *i*, the elements after position *i* need to be moved one position to the left (element from position *j* is moved to position *j-1*).



- capacity (cap): 10
- length (len): **5**

- **Delete the element from position 3**

DynamicArray - deleteFromPosition

```
function deleteFromPosition (da, i is:
    if i > 0 and i ≤ da.len then
        oldElem ← da.elems[i]
        for index ← i, da.len-1 execute
            da.elems[i] ← da.elems[i+1]
        end-for
        da.len ← da.len - 1
    else
        @throw exception
    end-if
    deleteFromPosition ← oldElem
end-function
```

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size - $\Theta(1)$
  - getElement -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size - $\Theta(1)$
  - getElement - $\Theta(1)$
  - setElement -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size - $\Theta(1)$
  - getElement - $\Theta(1)$
  - setElement - $\Theta(1)$
  - iterator -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size - $\Theta(1)$
  - getElement - $\Theta(1)$
  - setElement - $\Theta(1)$
  - iterator - $\Theta(1)$
  - addToPosition -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size - $\Theta(1)$
  - getElement - $\Theta(1)$
  - setElement - $\Theta(1)$
  - iterator - $\Theta(1)$
  - addToPosition - $O(n)$
  - deleteFromEnd -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size - $\Theta(1)$
  - getElement - $\Theta(1)$
  - setElement - $\Theta(1)$
  - iterator - $\Theta(1)$
  - addToPosition - $O(n)$
  - deleteFromEnd - $\Theta(1)$
  - deleteFromPosition -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size - $\Theta(1)$
  - getElement - $\Theta(1)$
  - setElement - $\Theta(1)$
  - iterator - $\Theta(1)$
  - addToPosition - $O(n)$
  - deleteFromEnd - $\Theta(1)$
  - deleteFromPosition - $O(n)$
  - addToEnd -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
  - size - $\Theta(1)$
  - getElement - $\Theta(1)$
  - setElement - $\Theta(1)$
  - iterator - $\Theta(1)$
  - addToPosition - $O(n)$
  - deleteFromEnd - $\Theta(1)$
  - deleteFromPosition - $O(n)$
  - addToEnd - $\Theta(1)$ *amortized*

Amortized analysis

- In *asymptotic* time complexity analysis we consider a single run of an algorithm.
  - *addToEnd* should have complexity $O(n)$ - when we have to resize the array, we need to move every existing element, so the number of instructions is proportional to the length of the array
  - Consequently, a sequence of n calls to the *addToEnd* operation would have complexity $O(n^2)$
- In *amortized* time complexity analysis we consider a sequence of operations and compute the average time for these operations.
  - In amortized time complexity analysis we will consider the total complexity of n calls to the *addToEnd* operation and divide this by n, to get the *amortized* complexity of the algorithm.

Amortized analysis

- We can observe that we rarely have to resize the array (if we consider a sequence of n operations).
- Consider c_i the cost ($\approx$ number of instructions) for the i^{th} call to *addToEnd*
- Considering that we double the capacity at each resize operation, at the ith operation we perform a resize if *i-1* is a pover of 2. So, the cost of operation i, c_i, is:

$$c_i = \begin{cases} i, & \text{if i-1 is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

Amortized analysis

- Cost of n operations is:

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{[log_2 n]} 2^j < n + 2n = 3n$$

- The sum contains at most n values of 1 (this is where the n term comes from) and at most (integer part of) $log_2 n$ terms of the form 2^j.
- Since the total cost of n operations is $3n$, we can say that the cost of one operation is 3, which is constant.

Amortized analysis

- While the worst case time complexity of *addToEnd* is still $O(n)$, the amortized complexity is $\Theta(1)$.
- The amortized complexity is no longer valid, if the resize operation just adds a constant number of new slots.
- In case of the *addToPosition* operation, both the worst case and the amortized complexity of the operation is $O(n)$ - even if resize is performed rarely, we need to move elements to empty the position where we put the new element.

Amortized analysis

- In order to avoid having a Dynamic Array with too many empty slots, we can resize the array after deletion as well, if the array becomes "too empty".
- How empty should the array become before resize? Which of the following two strategies do you think is better? Why?
  - Wait until the table is only half full (da.len $\approx$ da.cap/2) and resize it to the half of its capacity
  - Wait until the table is only a quarter full (da.len $\approx$ da.cap/4) and resize it to the half of its capacity

Iterator

- An *iterator* is a structure that is used to iterate through the elements of a container.
- Containers can be represented in different ways, using different data structures. Iterators are used to offer a common and generic way of moving through all the elements of a container, independently of the representation of the container.
- Every container that can be iterated, has to contain in the interface an operation called *iterator* that will create and return an iterator over the container.

Iterator

- An iterator usually contains:
  - a reference to the container it iterates over
  - a reference to a *current element* from the container
- Iterating through the elements of the container means actually moving this *current element* from one element to another until the iterator becomes *invalid*
- The exact way of representing the *current element* from the iterator depends on the data structure used for the implementation of the container. If the representation/ implementation of the container changes, we need to change the representation/ implementation of the iterator as well.

Iterator - Interface I

- **Domain** of an Iterator

$$\mathcal{I} = \{\textbf{it} | \text{it is an iterator over a container with elements of type TElem } \}$$

Iterator - Interface II

- **Interface** of an Iterator:

Iterator - Interface III

- init(it, c)
  - **description:** creates a new iterator for a container
  - **pre:** c is a container
  - **post:** $it \in \mathcal{I}$ and it points to the first element in c if c is not empty or it is not valid

Iterator - Interface IV

- getCurrent(it)
  - **description:** returns the current element from the iterator
  - **pre:** $it \in \mathcal{I}$, it is valid
  - **post:** $getCurrent \leftarrow e, e \in TElem$, e is the current element from it
  - **throws:** an exception if it is invalid

Iterator - Interface V

- next(it)
  - **description:** moves the current element from the container to the next element or makes the iterator invalid if no elements are left
  - **pre:** $it \in \mathcal{I}$, it is valid
  - **post:** the current element from it points to the next element from the container
  - **throws:** an exception if it is invalid

Iterator - Interface VI

- valid(it)
  - **description:** verifies if the iterator is valid
  - **pre:** $it \in \mathcal{I}$
  - **post:**

$$valid \leftarrow \begin{cases} True, & \text{if it points to a valid element from the container} \\ False & \text{otherwise} \end{cases}$$