DATA STRUCTURES AND ALGORITHMS
LECTURE 2

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2023 - 2024

In Lecture 1...

- Course Organization
- Abstract Data Types and Data Structures
- Pseudocode
- Algorithm Analysis
  - O - notation
  - Ω - notation
  - Θ - notation
  - Best Case, Worst Case, Average Case
  - Computing the complexity of recursive functions

Today

- Dynamic Array
- Iterators
- ADT Bag

Records

- A *record* (or *struct*) is a static data structure.
- It represents the reunion of a fixed number of components (which can have different types) that form a logical unit together.
- We call the components of a record *fields*.
- For example, we can have a record to denote a *Person* formed of fields for *name*, *date of birth*, *address*, etc.

Person:
    name: String
    dob: String
    address: String
    etc.

Arrays

- An array is one of the simplest and most basic data structures.
- An array can hold a fixed number of elements of the same type and these elements occupy a contiguous memory block.
- Arrays are often used as a basis for other (more complex) data structures.

- When a new array is created we have to specify two things:
  - The type of the elements in the array
  - The maximum number of elements that can be stored in the array (*capacity* of the array)
- The memory occupied by the array will be the capacity times the size of one element.
- The array itself is memorized by the address of the first element.

- An array of *boolean* values (addresses of the elements are displayed in base 16 and base 10)

```
Size of boolean: 1
Address of array: 00EFF760
Address of element from position 0: 00EFF760 15726432
Address of element from position 1: 00EFF761 15726433
Address of element from position 2: 00EFF762 15726434
Address of element from position 3: 00EFF763 15726435
Address of element from position 4: 00EFF764 15726436
Address of element from position 5: 00EFF765 15726437
Address of element from position 6: 00EFF766 15726438
Address of element from position 7: 00EFF767 15726439
```

15726440

- Can you guess the address of the element from position 8?

- An array of *integer* values (integer values occupy 4 bytes)

```
Size of int: 4
Address of array: 00D9FE6C
Address of element from position 0: 00D9FE6C 14286444
Address of element from position 1: 00D9FE70 14286448
Address of element from position 2: 00D9FE74 14286452
Address of element from position 3: 00D9FE78 14286456
Address of element from position 4: 00D9FE7C 14286460
Address of element from position 5: 00D9FE80 14286464
Address of element from position 6: 00D9FE84 14286468
Address of element from position 7: 00D9FE88 14286472
```

14286476

- Can you guess the address of the element from position 8?

- An array of *fraction* record values (the fraction record is composed of two integers)

```
Size of fraction: 8
Address of array: 007BF97C
Address of element from position 0: 007BF97C 8124796
Address of element from position 1: 007BF984 8124804
Address of element from position 2: 007BF98C 8124812
Address of element from position 3: 007BF994 8124820
Address of element from position 4: 007BF99C 8124828
Address of element from position 5: 007BF9A4 8124836
Address of element from position 6: 007BF9AC 8124844
Address of element from position 7: 007BF9B4 8124852
```

- Can you guess the address of the element from position 8?

Arrays

- The main **advantage** of arrays is that any element of the array can be accessed in constant time ($\Theta(1)$), because the address of the element can simply be computed (considering that the first element is at position 0):

Address of i^{th} element $=$ address of array $+$ i * size of an element

- The above formula works even if we consider that the first element is at position 1, but then we need to use $i - 1$ instead of i.

- An array is a static structure: once the *capacity* of the array is specified, you cannot add or delete slots from it (you can modify the value of the elements from the slots, but the number of slots, the capacity, remains the same)

- This leads to an important **disadvantage**: we need to know/estimate from the beginning the number of elements:
  - if the capacity is too small: we cannot store every element we want to
  - if the capacity is too big: we waste memory

- 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)
  - *nrElem* - denotes the actual number of elements stored in the array
  - *elems* - denotes the actual array with *capacity* slots for TElems allocated

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

- When the value of *nrElem* equals the value of *capacity*, we say that the array is full. If more elements need to be added, the *capacity* of the array is increased (usually doubled) and the array is *resized*.

- During the *resize* operation a new, bigger array is allocated and the existing elements are copied from the old array to the new one.

- Optionally, *resize* can be performed after delete operations as well: if the dynamic array becomes "too empty", a resize operation can be performed to shrink its size (to avoid occupying unused memory).

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 it as an ADT, and discuss its interface.

Dynamic Array - Interface I

- **Domain** of ADT DynamicArray

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

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

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)

- 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)

- getElement(da, i)
  - **description:** returns the element from a position from the DynamicArray
  - **pre:** $da \in \mathcal{DA}$, $1 \leq i \leq da.nrElem$
  - **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:** $da \in \mathcal{DA}$, $1 \leq i \leq da.nrElem$, $e \in TElem$
  - **post:** $da' \in \mathcal{DA}, da'.e_i = e$ (the i^{th} element from *da'* becomes e), $da'.e_j = da.e_j \; \forall \, 1 \leq j \leq n, j \neq i$.
    setElement $\leftarrow e_{old}$, $e_{old} \in TElem$, $e_{old} \leftarrow da.e_i$ (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:** $da' \in \mathcal{DA}, \ da'.nrElem = da.nrElem + 1; da'.e_{da'.nrElem} = e \ (da.cap = da.nrElem \Rightarrow da'.cap \leftarrow da.cap * 2)$

- 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:** $da \in \mathcal{DA},\ 1 \leq i \leq da.nrElem + 1,\ e \in TElem$
  - **post:** $da' \in \mathcal{DA},\ da'.nrElem = da.nrElem + 1,\ da'.e_j = da.e_{j-1} \forall j = da'.nrElem, da'.nrElem - 1, ..., i+1, da'.e_i = e\ , da'.e_j = da.e_j\ \forall j = i-1, ..., 1\ (da.cap = da.nrElem \Rightarrow da'.cap \leftarrow da.cap * 2)$
  - **throws:** an exception if i is not a valid position (da.nrElem+1 is a valid position when adding a new element)

- deleteFromPosition(da, i)
  - **description:** deletes an element from a given position from the DynamicArray. Returns the deleted element
  - **pre:** $da \in \mathcal{DA}$, $1 \leq i \leq da.nrElem$
  - **post:** $da' \in \mathcal{DA}$, $da'.nrElem = da.nrElem - 1$, $da'.e_j = da.e_{j+1} \forall i \leq j \leq da'.nrElem$, $da'.e_j = da.e_j\ \forall 1 \leq j < i$ deleteFromPosition $\leftarrow$ e, $e \in TElem$, $e = da.e_i$
  - **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*, the current element from *it* refers to the first element from *da*, or, if *da* is empty, *it* is invalid

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
  - Remove the element from the end of the Dynamic Array
  - etc.

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

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

Dynamic Array - addToEnd - Case 1

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

- capacity (cap): 10
- nrElem: 6

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

Dynamic Array - addToEnd - Case 1

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

- capacity (cap): 10
- nrElem: 6

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

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

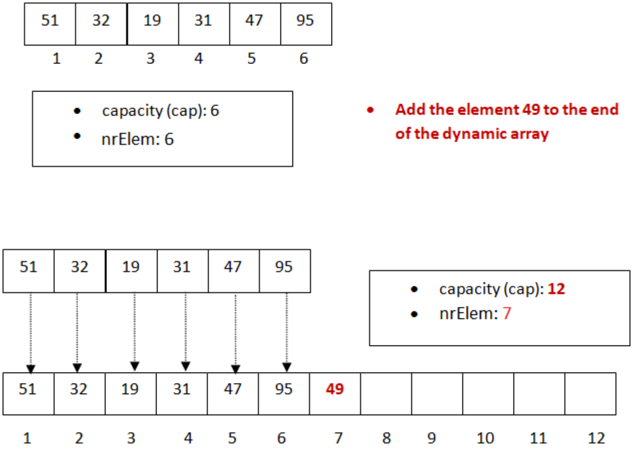
- capacity (cap): 10
- nrElem: 7

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

- capacity (cap): 6
- nrElem: 6

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

Dynamic Array - addToEnd - Case 2



Dynamic Array - addToEnd

```
subalgorithm addToEnd (da, e) is:
    if da.nrElem = 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.nrElem 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.nrElem ← da.nrElem + 1
    da.elems[da.nrElem] ← e
end-subalgorithm
```

- What is the complexity of *addToEnd*?

Dynamic Array - addToPosition

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

- capacity (cap): 10
- nrElem: 6

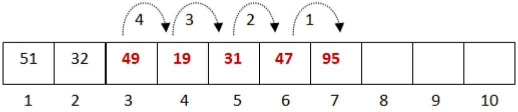
- **Add the element 49 to position 3**

Dynamic Array - addToPosition

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

- capacity (cap): 10
- nrElem: 6

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



- capacity (cap): 10
- nrElem: 7

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

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

- What is the complexity of *addToPosition*?

- **Obs.**: 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).

- The value used for increasing the capacity is often called the *growth factor*.

- How do dynamic arrays in other programming languages grow at resize? (I tried to look at capacities for 100 adds)
  - Microsoft Visual C++ <vector> - multiply by 1.5 (initially 0, then 1, 2, 3, 4, 6, 9, 13, 19, 28, 42, 63, 94, etc.)
    - But it also has a *resize* operation which lets you resize it *manually* and the new size can be any value (less or greater than current size).
  - Java - has two containers based on Dynamic Array
    - *Vector* - multiply by 2 (initially 10, 20, 40, 80, 160)
    - *ArrayList* - not specified in the documentation: *The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.* Many implementations work with a factor of 1.5 (but it cannot be checked, unless you look at the actual implementation).

- Both Vector and ArrayList have an *ensureCapacity* method, where you can make sure that the capacity is at least the specified value. This is useful, if a larger number of add operations are going to follow and we want to avoid intermediary resizes.

- Python *list* - this is also implementation dependent and hidden. However, by doing some tricks and accessing the total size of memory occupied by a list you can infer the capacity. In the Python implementation from Visual Studio, capacities are: 4, 8, 16, 24, 32, 40, 52, 64, 76, 92, 108 $=>$ the growth factor is about 1.25.

- C# - *List* in C# has a visible attribute *Capacity*, where you can see the current capacity. It is multiplied by 2 (initially 0, 4, 8, 16, 32, 64, 128).
    - This *Capacity* can be used to set the capacity as well, but you cannot make it less than the current number of elements (otherwise an Exception will be thrown).

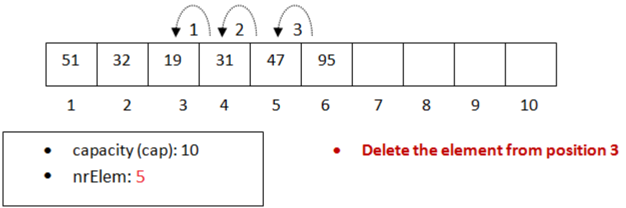
Dynamic Array

- 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
```

- 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*).



- 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 - $O(n)$
  - deleteGivenElement - $\Theta(n)$
  - addToEnd - $\Theta(1)$ *amortized*

- In *asymptotic* time complexity analysis we consider one 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.

- We can observe that if we consider a sequence of n operations, we rarely have to resize the array
- 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 *i*th operation we perform a resize if *i-1* is a power of 2. So, the cost of operation i, c_i, is:

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

- 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.

When do we have amortized complexity?

- The reason why in case of *addToEnd* we can talk about amortized complexity is that the worst case situation (the resize) happens rarely.
- Whenever you have an algorithm and you want to determine whether amortized complexity computation is applicable, ask the following questions:
  - Can I have worst case complexity for two calls in a row (one after the another)?
- If the answer is YES, than you do not have a situation of amortized complexity computation. (If the answer is NO, it is still not sure that you do have amortized complexity, but if it is YES, you definitely do not have amortized complexity.)

- 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.nrElem $\approx$ da.cap/2) and resize it to the half of its capacity
  - Wait until the table is only a quarter full (da.nrElem $\approx$ da.cap/4) and resize it to the half of its capacity

Iterator

- An *iterator* is an abstract data type 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 representation of the container (data structure used for the implementation of the container and possibly other representation details). If the representation/ implementation of the container changes, we need to change the representation/ implementation of the iterator as well (think about the Bag from Seminar 1).

- **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:

- 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

- 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 the iterator is not valid

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:** $it' \in \mathcal{I}$, the current element from *it'* points to the next element from the container or it' is invalid if no more elements are left
  - **throws:** an exception if the iterator is not valid

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

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

Iterator - Interface VII

- first(it)
    - **description:** sets the current element from the iterator to the first element of the container
    - **pre:** $it \in \mathcal{I}$
    - **post:** $it' \in \mathcal{I}$, the current element from it' points to the first element of the container if it is not empty, or it' is invalid

Types of iterators I

- The interface presented above describes the simplest iterator: *unidirectional* and *read-only*

- A *unidirectional* iterator can be used to iterate through a container in one direction only (usually *forward*, but we can define a *reverse* iterator as well).

- A *bidirectional* iterator can be used to iterate in both directions. Besides the *next* operation it has an operation called *previous* and it could also have a *last* operation (the pair of *first*).

Types of iterators II

- A *random access* iterator can be used to move multiple steps (not just one step forward or one step backward).

- A *read-only* iterator can be used to iterate through the container, but cannot be used to change it.

- A *read-write* iterator can be used to add/delete elements to/from the container.

Using the iterator

- Since the interface of an iterator is the same, independently of the exact container or its representation, the following subalgorithm can be used to print the elements of any container.

```
subalgorithm printContainer(c) is:
//pre: c is a container
//post: the elements of c were printed
//we create an iterator using the iterator method of the container
   iterator(c, it)
   while valid(it) execute
      //get the current element from the iterator
      getCurrent(it, elem)
      print elem
      //go to the next element
      next(it)
   end-while
end-subalgorithm
```

- How can we define an iterator for a Dynamic Array?
- How can we represent that *current element* from the iterator?
- In case of a Dynamic Array, the simplest way to represent a *current element* is to retain the position of the *current element*.

IteratorDA:
  da: DynamicArray
  current: Integer

- Let's see how the operations of the iterator can be implemented.

- What do we need to do in the *init* operation?

subalgorithm init(it, da) *is:*
//it is an IteratorDA, da is a Dynamic Array
  it.da ← da
  it.current ← 1
end-subalgorithm

- Complexity: $\Theta(1)$

- What do we need to do in the *getCurrent* operation?

function getCurrent(it) *is:*
  if it.current $>$ it.da.nrElem **then**
    @throw exception
  end-if
  getCurrent $\leftarrow$ it.da.elems[it.current]
end-function

- Complexity: $\Theta(1)$

Iterator for a Dynamic Array - next

- What do we need to do in the *next* operation?

subalgorithm next(it) *is:*
    if it.current $>$ it.da.nrElem **then**
        @throw exception
    end-if
    it.current $\leftarrow$ it.current $+ 1$
end-subalgorithm

- Complexity: $\Theta(1)$

- What do we need to do in the *valid* operation?

function valid(it) *is:*
  if it.current <= it.da.nrElem **then**
    valid ← True
  else
    valid ← False
  end-if
end-function

- Complexity: $\Theta(1)$

Iterator for a Dynamic Array - first

- What do we need to do in the *first* operation?

subalgorithm first(it) *is:*
  it.current $\leftarrow 1$
end-subalgorithm

- Complexity: $\Theta(1)$

- We can print the content of a Dynamic Array in two ways:
    - Using an iterator (as present above for a container)
    - Using the positions (indexes) of elements

```
subalgorithm printDAWithIterator(da) is:
//pre: da is a DynamicArray
//we create an iterator using the iterator method of DA
    iterator(da, it)
    while valid(it) execute
        //get the current element from the iterator
        elem ← getCurrent(it)
        print elem
        //go to the next element
        next(it)
    end-while
end-subalgorithm
```

$n * (1+1+1)$

- What is the complexity of *printDAWithIterator*? $\Theta(n)$

Print with indexes

subalgorithm printDAWithIndexes(da) **is:**
//pre: da is a Dynamic Array
  for i ← 1, size(da) **execute** n x(1+1)
    elem ← getElement(da, i)
    print elem
  end-for
end-subalgorithm

- What is the complexity of *printDAWithIndexes*? $\Theta(n)$

Iterator for a Dynamic Array

- In case of a Dynamic Array both printing algorithms have $\Theta(n)$ complexity
- For other data structures/containers we need iterator because
  - there are no positions in the data structure/container
  - the time complexity of iterating through all the elements is smaller (in general we want all iterator operations to be $\Theta(1)$)

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.

- A Bag can be represented using several data structures, one of them being a 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. (R1)
    - Store each element only once and keep a frequency count for it. (R2)

- Assume a dynamic array as data structure for the representation (but the idea is applicable for other representations as well)
- Assume that we have a Bag with the following numbers: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R1 the Bag looks in the following way:

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						

- Add element -5

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	-5					

ADT Bag - R1 example

- Remove element 6

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

- **Obs.** The order of the elements in a Bag is not important, you do not need to keep the same order in which they were added. So, when an element needs to be removed, you do not need to do the array-specific *move each element one position to the left* operation, you can simply take the last element and put it in the place of the removed one, for a small improvement in the number of operations.
- Think about it: What is the complexity of the remove operation when:
  - we move elements to the left ? $\Theta(n)$
  - we take the last element and put it to replace the removed one? $\Theta(1)$

ADT Bag - R2 example

- Assume the same elements as before: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R2 the Bag looks in the following way:

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

- Add element -5

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

- Add element 7

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

ADT Bag - R2 example

- Remove element 6

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

- Remove element 1

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

- Besides the two representations presented above which can be used for other data structures as well, there are two other possible representations which are specific for dynamic arrays.

ADT Bag - R3

- Another representation would be to store the unique elements in a dynamic array and store separately the positions from this array for every element that appears in the Bag (R3).
- Assume the same elements as before: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R3 the Bag looks in the following way (assume 1-based indexing):

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

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

- Add element -5

elems

1	2	3	4	5	6	7	8	9
4	1	6	7	2	9	-5		

positions

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

- Add element 7

elems

1	2	3	4	5	6	7	8	9
4	1	6	7	2	9	-5		

positions

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

- Remove element 6

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

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

- Removing element 6 implies a few steps:
    - Finding the position of 6 in the *elems* array, let's call it *elempos*.
    - Finding *elempos* in the *positions* array, and removing it (move the last element in its place).
    - Checking if *elempos* still appears in the *positions* array. If not, it means we have removed the last occurrence of the element from the Bag. We need to remove element 6 from the *elems* array as well.
    - Removing 6 from the *elems* array, by moving the last element in its place.
    - Changing the value of the position for the last element (moved in place of 6) in the *positions* array.

ADT Bag - R3 example

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

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

- Remove element ~~1~~ -5

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

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

Summary

- Today we have talked about:
  - Dynamic Array - the most basic data structure
  - Iterator
  - ADT Bag - different representations
- Extra reading:
  - How does the iterator look like in Python? (next extra readings will be about C++, Java and C#)