DATA STRUCTURES AND ALGORITHMS

LECTURE 2

Lect. PhD. Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017 - 2018

In Lecture 1...

- Course Organization
- Abstract Data Types and Data Structures
- Pseudocode
- Algorithm Analysis

Today

Algorithm Analysis for Recursive Functions

- How can we compute the time complexity of a recursive algorithm?

Recursive Binary Search

```
function BinarySearchR (array, elem, start, end) is:
//array - an ordered array of integer numbers
//elem - the element we are searching for
//start - the beginning of the interval in which we search (inclusive)
//end - the end of the interval in which we search (inclusive)
  if start > end then
    BinarySearchR ← False
  end-if
  middle ← (start + end) / 2
  if array[middle] = elem then
    BinarySeachR ← True
  else if elem < array[middle] then
    BinarySearchR ← BinarySearchR(array, elem, start, middle-1)
  else
    BinarySearchR ← BinarySearchR(array, elem, middle+1, end)
  end-if
end-function
```

Recursive Binary Search

- First call to the *BinarySearchR* algorithms for an ordered array of *nr* elements:

```
BinarySearchR(array, elem, 1, nr)
```

- How do we compute the complexity of the BinarySearchR algorithm?

Recursive Binary Search

- We will denote the length of the sequence that we are checking at every iteration by n (so $n = end - start$)
- We need to write the recursive formula of the solution

$$T(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ T(\frac{n}{2}) + 1, & \text{otherwise} \end{cases}$$

Master method

- The *master method* can be used to compute the time complexity of algorithms having the following general recursive formula:

$$T(n) = a * T(\frac{n}{b}) + f(n)$$

- where $a \geq 1$, $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

Master method

- Advantage of the master method: we can determine the time complexity of a recursive algorithm without further computations.

- Disadvantage of the master method: we need to memorize the three cases of the method and there are some situations when none of these cases can be applied.

Computing the time complexity without the master method

- If we do not want to memorize the cases for the master method we can compute the time complexity in the following way:
- Recall, the recursive formula for BinarySearchR was:

$$T(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ T(\frac{n}{2}) + 1, & \text{otherwise} \end{cases}$$

Time complexity for BinarySearchR

- We suppose that $n = 2^k$ and rewrite the second branch of the recursive formula:

$$T(2^k) = T(2^{k-1}) + 1$$

- Now, we write what the value of $T(2^{k-1})$ is (based on the recursive formula)

$$T(2^{k-1}) = T(2^{k-2}) + 1$$

- Next, we add what the value of $T(2^{k-2})$ is (based on the recursive formula)

$$T(2^{k-2}) = T(2^{k-3}) + 1$$

Time complexity for BinarySearchR

- The last value that can be written is the value of $T(2^1)$

$$T(2^1) = T(2^0) + 1$$

Time complexity for BinarySearchR

- Now, we write all these equations together and add them (and we will see that many terms can be simplified, because they appear on the left hand side of an equation and the right hand side of another equation):

$$
\begin{array}{l}
T(2^k) = T(2^{k-1}) + 1 \\
T(2^{k-1}) = T(2^{k-2}) + 1 \\
T(2^{k-2}) = T(2^{k-3}) + 1 \\
\ldots \\
T(2^1) = T(2^0) + 1 \\
\hline
T(2^k) = T(2^0) + 1 + 1 + 1 + \ldots + 1 = 1 + k
\end{array}
\quad +
$$

Time complexity for BinarySearchR

- We started from the notation $n = 2^k$.
- We want to go back to the notation that uses n. If $n = 2^k \Rightarrow k = log_2 n$

$$T(2^k) = 1 + k$$
$$T(n) = 1 + log_2 n \in \Theta(log_2 n)$$

Another example

- Let's consider the following pseudocode and compute the time complexity of the algorithm:

```
subalgorithm operation(n, i) is:
//n and i are integer numbers, n is positive
   if n > 1 then
      i ← 2 * i
      m ← n/2
      operation(m, i-2)
      operation(m, i-1)
      operation(m, i+2)
      operation(m, i+1)
   else
      write i
   end-if
end-subalgorithm
```

- The first step is to write the recursive formula:

$$T(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ 4 * T(\frac{n}{2}) + 1, & \text{otherwise} \end{cases}$$

- We suppose that $n = 2^k$.

$$T(2^k) = 4 * T(2^{k-1}) + 1$$

- This time we need the value of $4 * T(2^{k-1})$

$$T(2^{k-1}) = 4 * T(2^{k-2}) + 1 \Rightarrow$$
$$4 * T(2^{k-1}) = 4^2 * T(2^{k-2}) + 4$$

- And the value of $4^2 * T(2^{k-2})$

$$4^2 * T(2^{k-2}) = 4^3 * T(2^{k-3}) + 4^2$$

- The last value we can compute is $4^{k-1} * T(2^1)$

$$4^{k-1} * T(2^1) = 4^k * T(2^0) + 4^{k-1}$$

- We write all the equations and add them:

$$
\begin{array}{l}
T(2^k) = 4 * T(2^{k-1}) + 1 \\
4 * T(2^{k-1}) = 4^2 * T(2^{k-2}) + 4 \\
4^2 * T(2^{k-2}) = 4^3 * T(2^{k-3}) + 4^2 \\
\dots \\
4^{k-1} * T(2^1) = 4^k * T(2^0) + 4^{k-1} \\
\hline
T(2^k) = 4^k * T(1) + 4^0 + 4^1 + 4^2 + \dots + 4^{k-1}
\end{array}
\quad +
$$

- $T(1)$ is 1 (first case from recursive formula)

$$T(2^k) = 4^0 + 4^1 + 4^2 + \dots + 4^{k-1} + 4^k$$

$$\sum_{i=0}^{n} p^i = \frac{p^{n+1} - 1}{p - 1}$$

$$T(2^k) = \frac{4^{k+1} - 1}{4 - 1} = \frac{4^k * 4 - 1}{3} = \frac{(2^k)^2 * 4 - 1}{3}$$

- We started from $n = 2^k$. Let's change back to n

$$T(n) = \frac{4n^2 - 1}{3} \in \Theta(n^2)$$

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 representation for other (more complex) data structures.

Arrays

- 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.

Arrays - Example 1

- An array of *boolean* values (boolean values occupy one byte)

```
Size of a boolean: 1
Address of array: 006FFB48
Address of element at position 0: 006FFB48
Address of element at position 1: 006FFB49
Address of element at position 2: 006FFB4A
Address of element at position 3: 006FFB4B
Address of element at position 4: 006FFB4C
Address of element at position 5: 006FFB4D
Address of element at position 6: 006FFB4E
Address of element at position 7: 006FFB4F
Address of element at position 8: 006FFB50
Press any key to continue . . .
```

- Can you guess the address of the element from position 9?

Arrays - Example 2

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

```
Size of an int: 4
Address of array: 008FF87C
Address of element at position 0: 008FF87C
Address of element at position 1: 008FF880
Address of element at position 2: 008FF884
Address of element at position 3: 008FF888
Address of element at position 4: 008FF88C
Address of element at position 5: 008FF890
Address of element at position 6: 008FF894
Address of element at position 7: 008FF898
Address of element at position 8: 008FF89C
Press any key to continue . . .
```

- Can you guess the address of the element from position 9?

Arrays - Example 3

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

```
Size of a fraction: 8
Address of array: 008FFD04
Address of element at position 0: 008FFD04
Address of element at position 1: 008FFD0C
Address of element at position 2: 008FFD14
Address of element at position 3: 008FFD1C
Address of element at position 4: 008FFD24
Address of element at position 5: 008FFD2C
Address of element at position 6: 008FFD34
Address of element at position 7: 008FFD3C
Address of element at position 8: 008FFD44
Press any key to continue . . .
```

- Can you guess the address of the element from position 9?

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):

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

Arrays

- An array is a static structure: once the *capacity* of the array is specified, you cannot add or delete slots from it (you can add and delete 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

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 - Resize

- When the value of *len* 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 - Interface I

- Although a Dynamic Array can be implemented in a single way (using an array that occupies a contiguous memory block, but which can grow and shrink), we can present it in an abstract way, as the **ADT DynamicArray**.

- **Domain** of ADT DynamicArray

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

Dynamic Array - Interface II

- **Interface** of the ADT Dynamic Array (interface of an ADT contains the set of operations that should exist for the ADT, together with the specifications, pre- and postconditions, for each operation)

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

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, e)
  - **description:** returns the element from a position from the DynamicArray
  - **pre:** $da \in \mathcal{DA},\ 1 \leq i \leq da.len$
  - **post:** $e \in TElem,\ e = da.e_i$ (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)
  - **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:** $da' \in \mathcal{DA},\ da'.len = da.len + 1; da'.e_{da'.len} = e\ (da.cap = da.len \Rightarrow da'.cap \leftarrow da.cap * 2)$

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:** $da \in \mathcal{DA},\ 1 \leq i \leq da.len,\ e \in TElem$
    - **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

- deleteFromEnd(da, e)
  - **description:** deletes an element from the end of the DynamicArray. Returns the deleted element
  - **pre:** $da \in \mathcal{DA},\ da.len > 0$
  - **post:** $e \in TElem,\ e = da.e_{da.len},\ da' \in \mathcal{DA},\ da'.len = da.len - 1$
  - **throws:** an exception if *da* is empty

Dynamic Array - Interface XI

- deleteFromPosition(da, i, e)
  - **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:** $e \in TElem\ e = da.e_i,\ da' \in \mathcal{DA},\ da'.len = da.len - 1,\ da'.e_j = da.e_{j+1} \forall i \leq j \leq da'.len$
  - **throws:** an exception if i is not a valid position

Dynamic Array - Interface XII

- 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 XIII

- 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

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

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

Dynamic Array - addToEnd - Case 2

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

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

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

Dynamic Array - addToEnd - Case 2

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

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

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

51	32	19	31	47	95

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

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

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

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

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

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

4 3 2 1

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

- 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

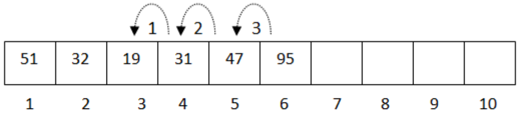
- Observations:
  - If the capacity of a Dynamic Array can be 0, then $da.cap \leftarrow da.cap * 2 + 1$ can be used at resize.
  - While it is not mandatory to double the capacity, it is important to define the new capacity as a product of the old one with a constant number greater than 1 (just adding 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 Dynamic Array Structure: 0081F098
 Length is: 3 Capacity is: 3
 Address of array from da: 0081EFF0
   Address of element from position 0: 0081EFF0
   Address of element from position 1: 0081EFF4
   Address of element from position 2: 0081EFF8
Address of Dynamic Array Structure: 0081F098
 Length is: 6 Capacity is: 6
 Address of array from da: 00820A10
   Address of element from position 0: 00820A10
   Address of element from position 1: 00820A14
   Address of element from position 2: 00820A18
   Address of element from position 3: 00820A1C
   Address of element from position 4: 00820A20
   Address of element from position 5: 00820A24
Address of Dynamic Array Structure: 0081F098
 Length is: 7 Capacity is: 7
 Address of array from da: 00820740
   Address of element from position 0: 00820740
   Address of element from position 1: 00820744
   Address of element from position 2: 00820748
   Address of element from position 3: 0082074C
   Address of element from position 4: 00820750
   Address of element from position 5: 00820754
   Address of element from position 6: 00820758
Press any key to continue . . .
```

Dynamic Array - delete operations

- There are two operations to delete an element from a position of the Dynamic Array:
  - To delete the element from the end of the array.
  - To delete an element from a given position *i*. In this case 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

subalgorithm deleteFromPosition (da, i, e) **is:**
  if i $>$ 0 **and** i $\leq$ da.len **then**
    e $\leftarrow$ da.elems[i]
    for index $\leftarrow$ i, da.len-1 **execute**
      da.elems[i] $\leftarrow$ da.elems[i+1]
    end-for
    da.len $\leftarrow$ da.len - 1
  else
    @throw exception
  end-if
end-subalgorithm

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 *i*th 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, e)
  - **description:** returns the current element from the iterator
  - **pre:** $it \in \mathcal{I}$, it is valid
  - **post:** $e \in TElem$, e is the current element from it

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

Iterator - Interface VI

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

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

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*.

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