DATA STRUCTURES

LECTURE 2

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

In Lecture 1...

- Course Organization
- Abstract Data Types and Data Structures
- Pseudocode
- Algorithm Analysis
  - The Θ, O and Ω notation
  - Best case, worst case, average case
  - Extra reading - empirical algorithm analysis

Today

- Algorithm Analysis
  - Algorithm analysis for recursive algorithms
- Dynamic Array
- Iterator

Algorithm Analysis for Recursive Algorithms

- 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 $\leftarrow$ False
  end-if
  middle $\leftarrow$ (start + end) / 2
  if array[middle] $=$ elem **then**
    BinarySeachR $\leftarrow$ True
  else if elem $<$ array[middle] **then**
    BinarySearchR $\leftarrow$ BinarySearchR(array, elem, start, middle-1)
  else
    BinarySearchR $\leftarrow$ BinarySearchR(array, elem, middle+1, end)
  end-if
end-function

- Initial call to the *BinarySearchR* algorithm 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

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

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

- 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)

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

- 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):

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

- **Obs:** For non-recursive functions adding a $+1$ or not, does not influence the result. In case of recursive functions it is important to have another term besides the recursive one.

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

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

Time complexity for BinarySearchR

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

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

- Actually, if we look at the code from BinarySearchR, we can observe that it has a best case (element can be found at the first iteration), so final complexity is $O(log_2 n)$

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:

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

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

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

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

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

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

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

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

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

- We write all the equations and add them:

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

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

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

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

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

- 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.

- 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.

Arrays - C++ Example 1

- An array of *boolean* values (boolean values occupy one byte)
- Obs: Address of elements is displayed in base 16 and base 10.

- An array of *boolean* values (boolean values occupy one byte)
- Obs: Address of elements is 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
```

- 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
```

- 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?

- 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

- **Obs:** If array indexing starts from 1, the above formula is still valid, but with $i - 1$ instead of i

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

- 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 - 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.

- **Domain** of ADT DynamicArray

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

- 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)

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

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

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

- 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
  - Return the position of a given element
  - etc.

- 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

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				
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

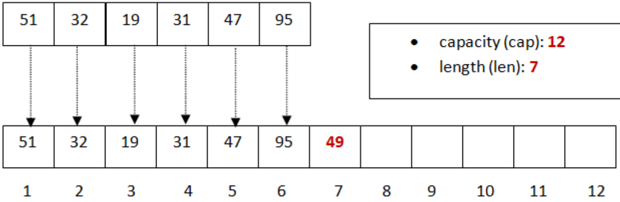
- **Add the element 49 to the end of the dynamic array**

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**



- 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
```

- 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
- length (len): 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
- length (len): 6

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

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
```

- What is the complexity of *addToPosition*?

- 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.

```
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
```

- How do dynamic arrays in other programming languages grow at resize?
  - C++ - multiply by 1.5 (initially 1, then 2, 3, 4, 6, 9, 13, etc.)
  - Java - multiply by 1.5 (initially 10, then 15, 22, 33, etc.)
  - Python - multiply by $\approx$ 1.125 (0, 4, 8, 16, 25, 35, 46, 58, etc.)
  - C# - multiply by 2 (initially 0, 4, 8, 16, 32, etc.)