DATA STRUCTURES
LECTURE 3

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

In Lecture 2...

- Algorithm analysis for recursive algorithms

- Dynamic Array
    - Definition
    - Interface
    - Representation
    - addToEnd, addToPosition

Today

- Dynamic Array
- Iterator
- Containers

Dynamic Array - Representation

- Remember the representation of the Dynamic Array

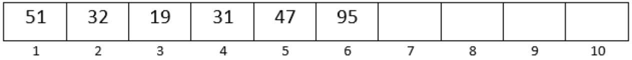
```
DynamicArray:
    cap: Integer
    len: Integer
    elems: TElem[]
```

- Last week we talked about how to add an element to the end of the Dynamic Array and how to add an element to a given position.

Dynamic Array - delete operation

- There are two operations to delete an element from a position of the Dynamic Array:
  - To delete an element from a given position *i*.
  - To delete a given element. In this case you first have to find the position of the element, and then delete the element from that position.

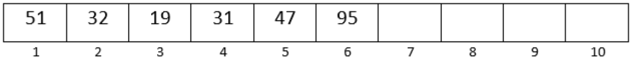
Dynamic Array - Delete From Position



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

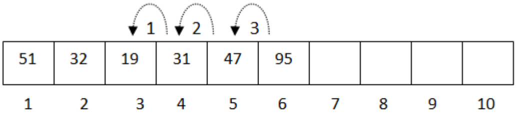
- Delete the element from position 3

Dynamic Array - Delete From Position



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

- Delete the element from position 3



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

- **Delete the element from position 3**

DynamicArray - deleteFromPosition

```
**subalgorithm** deleteFromPosition (da, i) **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
    deleteFromPosition $\leftarrow$ e
  **else**
    @throw exception
  **end-if**
**end-subalgorithm**
```

- What is the complexity of *deleteFromPosition*?

Dynamic Array - Complexity of operations

- size -

Dynamic Array - Complexity of operations

- size - $\Theta(1)$
- getElement -

Dynamic Array - Complexity of operations

- size - $\Theta(1)$
- getElement - $\Theta(1)$
- setElement -

Dynamic Array - Complexity of operations

- size - $\Theta(1)$
- getElement - $\Theta(1)$
- setElement - $\Theta(1)$
- iterator - $\Theta(1)$
- addToPosition -

Dynamic Array - Complexity of operations

- size - $\Theta(1)$
- getElement - $\Theta(1)$
- setElement - $\Theta(1)$
- iterator - $\Theta(1)$
- addToPosition - $O(n)$
- deleteFromEnd -

Dynamic Array - Complexity of 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

- 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

- 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 power of 2. So, the cost of operation *i*, c_i, is:

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

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.

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, you still might not have amortized complexity)

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 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 (compared to where *it* pointed to), or, if no more elements are left, it' is invalid
    - **throws:** an exception if *it* is not valid

Iterator - Interface VI

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

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

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*.

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
        elem ← getCurrent(it)
        print elem
        //go to the next element
        next(it)
    end-while
end-subalgorithm
```

Iterator for a Dynamic Array

- How can we define an iterator for a Dynamic Array?
- How can we represent that *current element* from the iterator?

Iterator for a Dynamic Array

- 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.

Iterator for a Dynamic Array - init

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

Iterator for a Dynamic Array - init

- 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:

Iterator for a Dynamic Array - init

- 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: $\Theta(1)$

Iterator for a Dynamic Array - getCurrent

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

Iterator for a Dynamic Array - getCurrent

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

```
function getCurrent(it) is:
    if not valid(it) then
        @throw an exception
    end-if
    getCurrent ← it.da.elems[it.current]
end-function
```

- Complexity:

Iterator for a Dynamic Array - getCurrent

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

```
**function** getCurrent(it) *is:*
    **if not** valid(it) **then**
        @throw an 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?

Iterator for a Dynamic Array - next

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

```
subalgorithm next(it) is:
    if not valid(it) then
        @throw exception
    end-if
    it.current ← it.current + 1
end-subalgorithm
```

- Complexity:

Iterator for a Dynamic Array - next

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

```
**subalgorithm** next(it) *is:*
   **if not** valid(it) **then**
      @throw exception
   **end-if**
   it.current ← it.current + 1
**end-subalgorithm**
```

- Complexity: $\Theta(1)$

Iterator for a Dynamic Array - valid

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

Iterator for a Dynamic Array - valid

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

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

- Complexity:

Iterator for a Dynamic Array - valid

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

```
function valid(it) is:
    if it.current <= it.da.len 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?

Iterator for a Dynamic Array - first

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

```
**subalgorithm** first(it) *is:*
    it.current ← 1
**end-subalgorithm**
```

- Complexity: $\Theta(1)$

Iterator for a Dynamic Array

- 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

Print with Iterator

```
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 $\leftarrow$ getCurrent(it)
      print elem
      //go to the next element
      next(it)
   end-while
end-subalgorithm
```

- What is the complexity of *printDAWithIterator*?

Print with indexes

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

- What is the complexity of *printDAWithIndexes*?

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

Containers

- There are many different containers, based on different properties:
  - do the elements have to be unique?
  - do the elements have positions assigned?
  - can we access any element or just some specific ones?
  - do we have simple elements, or key-value pairs?

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.

ADT Bag - representation

- 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)

ADT Bag - R1 example

- 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

ADT Bag - R1 example

- 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					

- Remove element 6

ADT Bag - R1 example

- 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					

- 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						

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

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

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

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

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			

ADT Bag - Dynamic Array specific representations

- 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
4	1	6	7	2	9			

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

ADT Bag - R3 example

- Add element -5

ADT Bag - R3 example

- Add element -5

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

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

ADT Bag - R3 example

- Add element -5

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

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

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

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		

ADT Bag - R3 example

- Remove element 6

ADT Bag - R3 example

- Remove element 6

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

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

- Remove element 1

ADT Bag - R3 example

- Remove element 6

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

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

- Remove element 1

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

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

ADT Bag - R4

- If the elements of the Bag are integer numbers (and a dynamic array is used for storing them), another representation is possible, where the positions of the array represent the elements and the value from the position is the frequency of the element. Thus, the frequency of the minimum element is at position 1 (assume 1-based indexing).
- Assume the same elements as before: 4, 1, 6, 4, 7, 2, 1, 1, 1, 9
- In R4 the Bag looks in the following way:

1	2	3	4	5	6	7	8	9	10	11
4	1	0	2	0	1	1	0	1		

Minimum element: 1

ADT Bag - R4 example

- Add element -5

ADT Bag - R4 example

- Add element -5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	1	1	0	1		

Minimum element: -5

- When indexing starts from 1, the element in the dynamic array that is on position i, represents the actual value: $minimum + i - 1 \Rightarrow$ position of an element e is $e - minimum + 1$

- Add element 7

ADT Bag - R4 example

- Add element -5

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	1	1	0	1		

Minimum element: -5

- When indexing starts from 1, the element in the dynamic array that is on position i, represents the actual value: $minimum + i - 1 \Rightarrow$ position of an element e is $e - minimum + 1$

- Add element 7

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	1	2	0	1		

Minimum element: -5

ADT Bag - R4 example

- Remove element 6

ADT Bag - R4 example

- Remove element 6

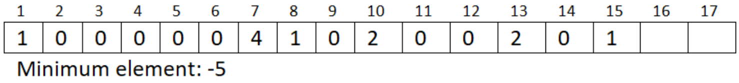
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	0	2	0	1		

Minimum element: -5

- Remove element 1

ADT Bag - R4 example

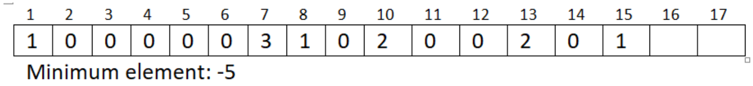
- Remove element 6



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	4	1	0	2	0	0	2	0	1		

Minimum element: -5

- Remove element 1



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	0	0	0	0	3	1	0	2	0	0	2	0	1		

Minimum element: -5