

# DATA STRUCTURES AND ALGORITHMS

## LECTURE 7

Lect. PhD. Marian Zsuzsanna

Babeş - Bolyai University  
Computer Science and Mathematics Faculty

2017 - 2018

## In Lecture 6...

- Sorted Linked List
- Linked Lists on Arrays
  - Singly Linked List on Array
    - init
    - search
    - insertFirst
    - insertPosition

# Today

- 1 Linked Lists on Arrays
- 2 ADT Set
- 3 ADT Map
- 4 Iterator

# Linked Lists on Arrays

elems		78	11	6	59	19		44		
next	7	6	5	-1	8	4	9	2	10	-1

head = 3

```
firstEmpty = 1
```

## Linked Lists on Arrays

- In a more formal way, we can simulate a singly linked list on an array with the following:
  - an array in which we will store the elements.
  - an array in which we will store the links (indexes to the next elements).
  - the capacity of the arrays (the two arrays have the same capacity, so we need only one value).
  - an index to tell where the *head* of the list is.
  - an index to tell where the first empty position in the array is.

## SLL on Array - Representation

- The representation of a singly linked list on an array is the following:

### SLLA:

```
elems: TElem[]  
next: Integer[]  
cap: Integer  
head: Integer  
firstEmpty: Integer
```

## SLLA - DeleteElement

**subalgorithm** deleteElement(slla, elem) is:

*//pre: slla is a SLLA; elem is a TElem*

*//post: the element elem is deleted from SLLA*

# SLLA - DeleteElement

**subalgorithm** deleteElement(slla, elem) **is:**

*//pre: slla is a SLLA; elem is a TElem*

*//post: the element elem is deleted from SLLA*

nodC  $\leftarrow$  slla.head

prevNode  $\leftarrow$  -1

**while** nodC  $\neq$  -1 **and** slla.elms[nodC]  $\neq$  elem **execute**

prevNode  $\leftarrow$  nodC

nodC  $\leftarrow$  slla.next[nodC]

**end-while**

**if** nodC  $\neq$  -1 **then**

**if** nodC = slla.head **then**

slla.head  $\leftarrow$  slla.next[slla.head]

**else**

slla.next[prevNode]  $\leftarrow$  slla.next[nodC]

**end-if**

*//continued on the next slide...*



## SLLA - DeleteElement

```
//add the nodC position to the list of empty spaces  
slla.next[nodC] ← slla.firstEmpty  
slla.firstEmpty ← nodC  
else  
  @the element does not exist  
end-if  
end-subalgorithm
```

- Complexity:  $O(n)$

## SLLA - Iterator

- Iterator for a SLLA is a combination of an iterator for an array and of an iterator for a singly linked list:
- Since the elements are stored in an array, the *currentElement* will be an index from the array.
- But since we have a linked list, going to the next element will not be done by incrementing the *currentElement* by one; we have to follow the *next* links.

# DLLA

- Obviously, we can define a doubly linked list as well without pointers, using arrays.
- For the DLLA we will see another way of representing a linked list on arrays:
  - The main idea is the same, we will use array indexes as links between elements
  - We are using the same information, but we are going to structure it differently
  - However, we can make it look more similar to linked lists with dynamic allocation

## DLLA - Node

- Linked Lists with dynamic allocation are made of nodes. We can define a structure to represent a node, even if we are working with arrays.
- A node (for a doubly linked list) contains the information and links towards the previous and the next nodes:

DLLANode:

info: TElem

next: Integer

prev: Integer

# DLLA

- Having defined the *DLLANode* structure, we only need one array, which will contain *DLLANodes*.
- Since it is a doubly linked list, we keep both the head and the tail of the list.

## DLLA:

```
nodes: DLLANode[]  
cap: Integer  
head: Integer  
tail: Integer  
firstEmpty: Integer
```

## DLLA - Allocate and free

- To make the representation and implementation even more similar to a dynamically allocated linked list, we can define the *allocate* and *free* functions as well.

**function** `allocate(dlla)` is:

## DLLA - Allocate and free

- To make the representation and implementation even more similar to a dynamically allocated linked list, we can define the *allocate* and *free* functions as well.

```
function allocate(dlla) is:
  //pre: dlla is a DLLA
  //post: a new element will be allocated and its position returned
  newElem ← dlla.firstEmpty
  if newElem ≠ -1 then
    dlla.firstEmpty ← dlla.nodes[dlla.firstEmpty].next
    dlla.nodes[dlla.firstEmpty].prev ← -1
    dlla.nodes[newElem].next ← -1
    dlla.nodes[newElem].prev ← -1
  end-if
  allocate ← newElem
end-function
```

## DLLA - Allocate and free

**subalgorithm** free (dlla, poz) **is:**



## DLLA - Allocate and free

**subalgorithm** free (dlla, poz) **is:**

*//pre: dlla is a DLLA, poz iss an integer number*

*//post: the position poz was freed*

$\text{dlla.nodes}[\text{poz}].\text{next} \leftarrow \text{dlla.firstEmpty}$

$\text{dlla.nodes}[\text{poz}].\text{prev} \leftarrow -1$

$\text{dlla.nodes}[\text{dlla.firstEmpty}].\text{prev} \leftarrow \text{poz}$

$\text{dlla.firstEmpty} \leftarrow \text{poz}$

**end-subalgorithm**

# DLLA - InsertPosition

**subalgorithm** insertPosition(dlla, elem, poz) **is:**

# DLLA - InsertPosition

**subalgorithm** insertPosition(dlla, elem, poz) **is:**

*//pre: dlla is a DLLA, elem is a TElem, poz is an integer number*

*//we assume that poz is a valid position*

*//post: the element elem is inserted in dlla at position poz*

newElem  $\leftarrow$  allocate(dlla)

**if** newElem = -1 **then**

    @resize

    newElem  $\leftarrow$  allocate(dlla)

**end-if**

dlla.nodes[newElem].info  $\leftarrow$  elem

**if** poz = 1 **then**

**if** dlla.head = -1 **then**

        dlla.head  $\leftarrow$  newElem

        dlla.tail  $\leftarrow$  newElem

**else**

*//continued on the next slide...*

# DLLA - InsertPosition

```

dlla.nodes[newElem].next  $\leftarrow$  dlla.head
dlla.nodes[dlla.head].prev  $\leftarrow$  newElem
dlla.head  $\leftarrow$  newElem

```

**end-if**

**else**

```

nodC  $\leftarrow$  dlla.head

```

```

pozC  $\leftarrow$  1

```

**while**  $\text{nodC} \neq -1$  **and**  $\text{pozC} < \text{poz} - 1$  **execute**

```

  nodC  $\leftarrow$  dlla.nodes[nodC].next

```

```

  pozC  $\leftarrow$  pozC + 1

```

**end-while**

**if**  $\text{nodC} \neq -1$  **then**

```

  nodNext  $\leftarrow$  dlla.nodes[nodC].next

```

```

  dlla.nodes[newElem].next  $\leftarrow$  nodNext

```

```

  dlla.nodes[newElem].prev  $\leftarrow$  nodC

```

```

  dlla.nodes[nodC].next  $\leftarrow$  newElem

```

*//continued on the next slide...*

## DLLA - InsertPosition

```
    if nodNext = -1 then
        dlla.tail ← newElem
    else
        dlla.nodes[nodNext].prev ← newElem
    end-if
end-if
end-subalgorithm
```

- Complexity:  $O(n)$

# DLLA - Iterator

## DLLA - Iterator

- The iterator for a DLLA contains as *current element* the index of the current node from the array.

DLLAIterator:

list: DLLA

currentElement: Integer

# DLLAlterator - init



## DLLAIterator - init

**subalgorithm** init(it, dlla) **is:**

*//pre: dlla is a DLLA*

*//post: it is a DLLAIterator for dlla*

it.list  $\leftarrow$  dlla

it.currentElement  $\leftarrow$  dlla.head

**end-subalgorithm**

- For a (dynamic) array, currentElement is set to 0 when an iterator is created. For a DLLA we need to set it to the head of the list (which might be position 0, but it might be a different position as well).

# DLLAlterator - getCurrent

## DLLAlterator - getCurrent

**subalgorithm** getCurrent(it, e) **is:**

*//pre: it is a DLLAlterator, it is valid*

*//post: e is a TElem, e is the current element from it*

$e \leftarrow \text{it.list.nodes}[\text{it.currentElement}].\text{info}$

**end-subalgorithm**

# DLLAlterator - next

## DLLAIterator - next

**subalgorithm** next (it) **is:**

*//pre: it is a DLLAIterator, it is valid*

*//post: the current elements from it is moved to the next element*

$\text{it.currentElement} \leftarrow \text{it.list.nodes}[\text{it.currentElement}].\text{next}$

**end-subalgorithm**

- In case a (dynamic) array, going to the next element means incrementing the *currentElement* by one. For a DLLA we need to follow the links.

# DLLAlterator - valid

## DLLAlterator - valid

**function** valid (it) **is:**

*//pre: it is a DLLAlterator*

*//post: valid return true is the current element is valid, false otherwise*

**if** it.currentElement = -1 **then**

    valid  $\leftarrow$  False

**else**

    valid  $\leftarrow$  True

**end-if**

**end-function**

# ADT Set

- A *Set* is a container in which the elements are unique, and their order is not important (they do not have positions).
  - No operations based on positions.
  - We cannot make assumptions regarding the order in which elements are stored and will be iterated.
- Domain of the ADT Set:  
 $\mathcal{S} = \{s | s \text{ is a set with elements of the type TElem}\}$



# Set - Interface I

- **init** ( $s$ )
  - **descr:** creates a new empty set.
  - **pre:** true
  - **post:**  $s \in \mathcal{S}$ ,  $s$  is an empty set.

# Set - Interface II

- $\text{add}(s, e)$ 
  - **descr:** adds a new element into the set.
  - **pre:**  $s \in \mathcal{S}, e \in TElem$
  - **post:**  $s' \in \mathcal{S}, s' = s \cup \{e\}$  ( $e$  is added only if it is not in  $s$  yet. If  $s$  contains the element  $e$  already, no change is made).
  - What happens if  $e$  is already in  $s$ ?

# Set - Interface III

- `remove(s, e)`
  - **descr:** removes an element from the set.
  - **pre:**  $s \in \mathcal{S}, e \in TElem$
  - **post:**  $s \in \mathcal{S}, s' = s \setminus \{e\}$  (if  $e$  is not in  $s$ ,  $s$  is not changed).

# Set - Interface IV

- $\text{find}(s, e)$ 
  - **descr:** verifies if an element is in the set.
  - **pre:**  $s \in \mathcal{S}, e \in TElem$
  - **post:**

$$\text{find} \leftarrow \begin{cases} \text{True}, & \text{if } e \in s \\ \text{False}, & \text{otherwise} \end{cases}$$

# Set - Interface V

- **size(s)**
  - **descr:** returns the number of elements from a set
  - **pre:**  $s \in \mathcal{S}$
  - **post:**  $\text{size} \leftarrow$  the number of elements from  $s$

# Set - Interface VI

- `iterator(s, it)`
  - **descr:** returns an iterator for a set
  - **pre:**  $s \in \mathcal{S}$
  - **post:**  $it \in \mathcal{I}$ ,  $it$  is an iterator over the set  $s$

# Set - Interface VII

- **destroy** ( $s$ )
  - **descr:** destroys a set
  - **pre:**  $s \in S$
  - **post:** the set  $s$  was destroyed.

# Set - Interface VIII

- Other possible operations (characteristic for sets from mathematics):
  - reunion of two sets
  - intersection of two sets
  - difference of two sets (elements that are present in the first set, but not in the second one)



# Sorted Set

- We can have a Set where the elements are ordered based on a *relation*  $\rightarrow$  *SortedSet*.
- The only change in the interface is for the *init* operation that will receive the *relation* as parameter.
- For a sorted set, the iterator has to iterate through the elements in the order given by the *relation*.

# Set

- If we want to implement the ADT Set (or ADT SortedSet), we can use the following data structures as representation:
  - (dynamic) array
  - linked list
  - hash tables - to be discussed later
  - (balanced) binary trees - for sorted sets - to be discussed later

# ADT Map

- A *Map* is a container where the elements are  $\langle \text{key}, \text{value} \rangle$  pairs.
- Each *key* has one single associated *value*, and we can access the values only by using the key  $\rightarrow$  no positions in a *Map*.
- Keys have to be unique in a *Map*, and each *key* has one single associated value (if a key can have multiple values we have a *MultiMap*).
- When we implement a *Map*, we should use a data structure that makes finding the *keys* easy.

# Map

- Examples of using a map:
  - Bank account number (as key) and every information associated with the bank account (as value)
  - Student id (as key) and every information about the student (as value)
  - etc.

- Domain of the ADT Map:

$\mathcal{M} = \{m \mid m \text{ is a map with elements } e = (k, v), \text{ where } k \in T\text{Key} \text{ and } v \in T\text{Value}\}$

# Map - Interface I

- `init(m)`
  - **descr:** creates a new empty map
  - **pre:** true
  - **post:**  $m \in \mathcal{M}$ ,  $m$  is an empty map.

# Map - Interface II

- `destroy(m)`
  - **descr:** destroys a map
  - **pre:**  $m \in \mathcal{M}$
  - **post:**  $m$  was destroyed

## Map - Interface III

- $\text{add}(m, k, v)$ 
  - **descr:** add a new key-value pair to the map (the operation can be called *put* as well)
  - **pre:**  $m \in \mathcal{M}, k \in T\text{Key}, v \in T\text{Value}$
  - **post:**  $m' \in \mathcal{M}, m' = m \cup \langle k, v \rangle$
- What happens if there is already a pair with  $k$  as key?

## Map - Interface IV

- `remove(m, k, v)`
  - **descr:** removes a pair with a given key from the map
  - **pre:**  $m \in \mathcal{M}, k \in T\text{Key}$
  - **post:**  $v \in T\text{Value}$ , where

$$v \leftarrow \begin{cases} v', & \text{if } \exists \langle k, v' \rangle \in m \text{ and } m' \in \mathcal{M}, \\ & m' = m \setminus \langle k, v' \rangle \\ 0_{T\text{Value}}, & \text{otherwise} \end{cases}$$



# Map - Interface V

- **search**( $m, k, v$ )
  - **descr:** searches for the value associated with a given key in the map
  - **pre:**  $m \in \mathcal{M}, k \in TKey$
  - **post:**  $v \in TValue$ , where

$$v \leftarrow \begin{cases} v', & \text{if } \exists \langle k, v' \rangle \in m \\ 0_{TValue}, & \text{otherwise} \end{cases}$$

# Map - Interface VI

- `iterator(m, it)`
  - **descr:** returns an iterator for a map
  - **pre:**  $m \in \mathcal{M}$
  - **post:**  $it \in \mathcal{I}$ ,  $it$  is an iterator over  $m$ .

## Map - Interface VII

- **size(m)**
  - **descr:** returns the number of pairs from the map
  - **pre:**  $m \in \mathcal{M}$
  - **post:**  $\text{size} \leftarrow$  the number of pairs from  $m$

## Map - Interface VIII

- `keys(m, s)`
  - **descr:** returns the set of keys from the map
  - **pre:**  $m \in \mathcal{M}$
  - **post:**  $s \in \mathcal{S}$ ,  $s$  is the set of all keys from  $m$

# Map - Interface IX

- `values(m, b)`
  - **descr:** returns a bag with all the values from the map
  - **pre:**  $m \in \mathcal{M}$
  - **post:**  $b \in \mathcal{B}$ ,  $b$  is the bag of all values from  $m$

# Map - Interface X

- **pairs**( $m, s$ )
  - **descr**: returns the set of pairs from the map
  - **pre**:  $m \in \mathcal{M}$
  - **post**:  $s \in \mathcal{S}$ ,  $s$  is the set of all pairs from  $m$

## Sorted Map

- We can have a Map where we can define an order (a relation) on the set of possible keys: instead of *TKey* we will have *TComp*.
- The only change in the interface is for the *init* operation that will receive the *relation* as parameter.
- For a sorted map, the iterator has to iterate through the pairs in the order given by the *relation*, and the operations *keys* and *pairs* return SortedSets.

# Map

- If we want to implement the ADT Map (or ADT SortedMap), we can use the following data structures as representation:
  - (dynamic) array
  - linked list
  - hash tables - to be discussed later
  - (balanced) binary trees - for sorted maps - to be discussed later



# Iterator - why do we need it? I

- Most containers have iterators and for every data structure we will discuss how we can implement an iterator for a container defined on that data structure.
- Why are iterators so important?

# Iterator - why do we need it? II

- They offer a uniform way of iterating through 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**

# Iterator - why do we need it? III

- For most containers the iterator is the only thing we have to see the content of the container.
  - List (will be discussed later) is the only container that has positions, for other containers we can use only the iterator.

## Iterator - why do we need it? IV

- Giving up positions, we can gain performance.
  - Containers that do not have positions can be represented on data structures where some operations have good complexities, but where the notion of a position does not naturally exist and where enforcing positions is really complicated (ex. hash tables).

# Iterator - why do we need it? V

- Even if we have positions, using an iterator might be faster.
  - Going through the elements of a linked list with an iterator is faster than going through every position one-by-one.