

DATA STRUCTURES AND ALGORITHMS

LECTURE 3

Lect. PhD. Marian Zsuzsanna

Babeş - Bolyai University
Computer Science and Mathematics Faculty

2017 - 2018

In Lecture 2...

- Algorithm Analysis
- Dynamic Array
- Iterator

Today

1 Iterators

2 Binary Heap

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

$\mathcal{I} = \{\mathbf{it} \mid \text{it is an iterator over a container with elements of type TElem} \}$

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

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 \leftarrow *da*

it.current \leftarrow 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 \leftarrow *da*

it.current \leftarrow 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?

```
subalgorithm getCurrent(it, e) is:  
    e  $\leftarrow$  it.da.elems[it.current]  
end-subalgorithm
```

- Complexity:

Iterator for a Dynamic Array - *getCurrent*

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

```
subalgorithm getCurrent(it, e) is:  
    e  $\leftarrow$  it.da.elems[it.current]  
end-subalgorithm
```

- 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*:

it.current \leftarrow 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*:

it.current \leftarrow 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

- 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 printDA(da) **is:**

//pre: da is a Dynamic Array

//post: the elements of da were printed

for $i \leftarrow 1, \text{size}(da)$ **execute**

 getElement(da, i, elem)

print elem

end-for

end-subalgorithm

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 (for example for the Bag ADT)
 - the time complexity of iterating through all the elements is smaller

Binary Heap

- A binary heap is a data structure that can be used as an efficient representation for Priority Queues (will be discussed later).
- A binary heap is a kind of hybrid between a dynamic array and a binary tree.
- The elements of the heap are actually stored in the dynamic array, but the array is visualized as a binary tree.
- **Note:** There are several heap data structures (Binary Heap, Binominal Heap, Fibonacci Heap, etc.) in the following we are going to use the term *heap* to represent a binary heap.

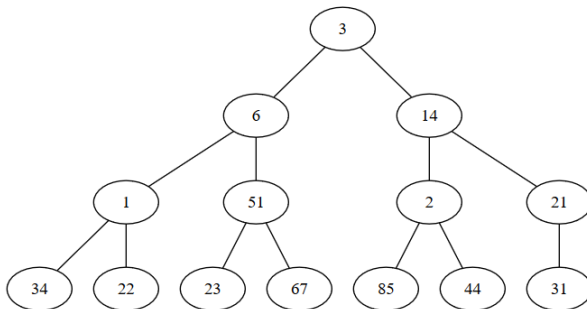
Heap

- Assume that we have the following array (upper row contains positions, lower row contains elements):

1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	6	14	1	51	2	21	34	22	23	67	85	44	31

Heap

- We can visualize this array as a binary tree, in which each node has exactly 2 children, except for the last two rows, but there the children of the nodes are completed from left to right.



Heap

- If the elements of the array are: $a_1, a_2, a_3, \dots, a_n$, we know that:
 - a_1 is the root of the heap
 - for an element from position i , its children are on positions $2 * i$ and $2 * i + 1$ (if $2 * i$ and $2 * i + 1$ is less than or equal to n)
 - for an element from position i ($i > 1$), the parent of the element is on position $[i/2]$ (integer part of $i/2$)

Heap

- A *binary heap* is an array that can be visualized as a binary tree having a *heap structure* and a *heap property*.
 - *Heap structure*: in the binary tree every node has exactly 2 children, except for the last two levels, where children are completed from left to right.
 - *Heap property*: $a_i \geq a_{2*i}$ (if $2 * i \leq n$) and $a_i \geq a_{2*i+1}$ (if $2 * i + 1 \leq n$)
 - The \geq relation between a node and both its descendants can be generalized (other relations can be used as well).

Heap - Notes

- If we use the \geq relation, we will have a *MAX-HEAP*.
- If we use the \leq relation, we will have a *MIN-HEAP*.
- The height of a heap with n elements is $\log_2 n$, so the operations performed on the heap have $O(\log_2 n)$ complexity.

Heap - operations

- A heap can be used as representation for a Priority Queue and it has two specific operations:
 - add a new element in the heap (in such a way that we keep both the heap structure and the heap property).
 - remove (we always remove the root of the heap - no other element can be removed).

Heap - representation

Heap:

cap: Integer

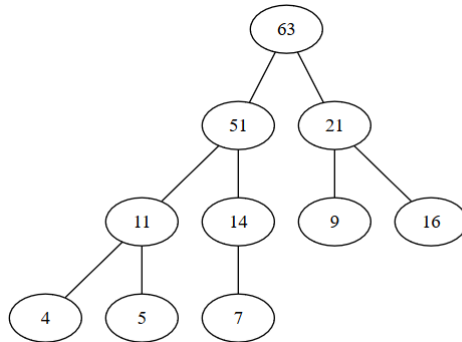
len: Integer

elems: TElem[]

- For the implementation we will assume that we have a MAX-HEAP.

Heap - Add - example

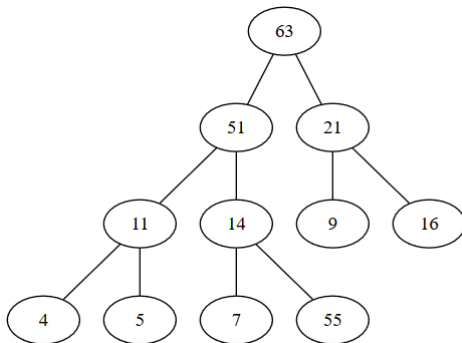
- Consider the following (MAX) heap:



- Let's add the number 55 to the heap.

Heap - Add - example

- In order to keep the *heap structure*, we will add the new node as the right child of the node 14 (and as the last element of the array in which the elements are kept).

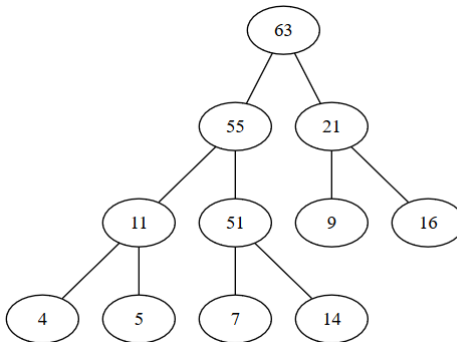


Heap - Add - example

- *Heap property* is not kept: 14 has as child node 55 (since it is a MAX-heap, each node has to be greater than or equal to its descendants).
- In order to restore the heap property, we will start a *bubble-up* process: we will keep swapping the value of the new node with the value of its parent node, until it gets to its final place. No other node from the heap is changed.

Heap - Add - example

- When *bubble-up* ends:



Heap - add

subalgorithm add(heap, e) **is:**

//heap - a heap

//e - the element to be added

if heap.len = heap.cap **then**

 @ resize

end-if

heap.elms[heap.len+1] \leftarrow e

heap.len \leftarrow heap.len + 1

bubble-up(heap, heap.len)

end-subalgorithm

Heap - add

subalgorithm bubble-up (heap, p) **is:**

//heap - a heap

//p - position from which we bubble the new node up

poz \leftarrow p

elem \leftarrow heap.elems[p]

parent \leftarrow p / 2

while poz > 1 **and** elem > heap.elems[parent] **execute**

//move parent down

heap.elems[poz] \leftarrow heap.elems[parent]

poz \leftarrow parent

parent \leftarrow poz / 2

end-while

heap.elems[poz] \leftarrow elem

end-subalgorithm

- Complexity:

Heap - add

subalgorithm bubble-up (heap, p) **is:**

//heap - a heap

//p - position from which we bubble the new node up

poz \leftarrow p

elem \leftarrow heap.elems[p]

parent \leftarrow p / 2

while poz > 1 **and** elem > heap.elems[parent] **execute**

//move parent down

heap.elems[poz] \leftarrow heap.elems[parent]

poz \leftarrow parent

parent \leftarrow poz / 2

end-while

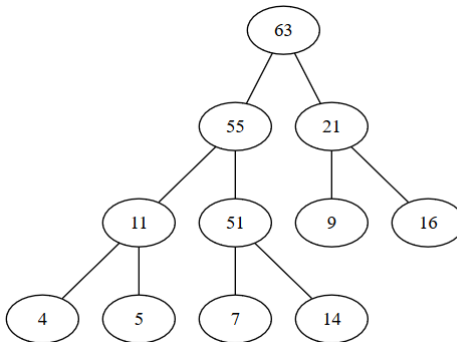
heap.elems[poz] \leftarrow elem

end-subalgorithm

- Complexity: $O(\log_2 n)$
- Can you give an example when the complexity of the algorithm is less than $\log_2 n$ (best case scenario)?

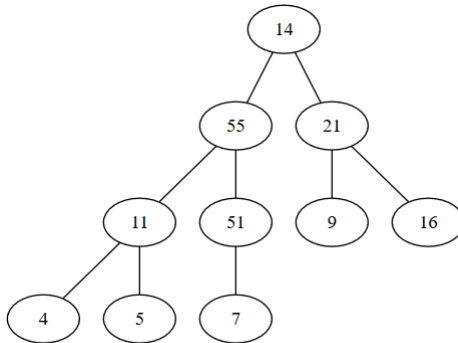
Heap - Remove - example

- From a heap we can only remove the root element.



Heap - Remove - example

- In order to keep the *heap structure*, when we remove the root, we are going to move the last element from the array to be the root.

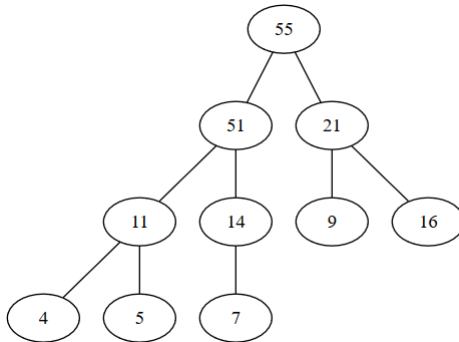


Heap - Remove - example

- *Heap property* is not kept: the root is no longer the maximum element.
- In order to restore the heap property, we will start a *bubble-down* process, where the new node will be swapped with its maximum child, until it becomes a leaf, or until it will be greater than both children.

Heap - Remove - example

- When the bubble-down process ends:



Heap - remove

function remove(heap, deletedElem) **is:**

//heap - is a heap

//deletedElem - is an output parameter, the removed element

if heap.len = 0 **then**

 @ error - empty heap

end-if

deletedElem \leftarrow heap.elems[1]

heap.elems[1] \leftarrow heap.elems[heap.len]

heap.len \leftarrow heap.len - 1

bubble-down(heap, 1)

remove \leftarrow deletedElem

end-function

Heap - remove

subalgorithm bubble-down(heap, p) **is:**

//heap - is a heap

//p - position from which we move down the element

poz \leftarrow p

elem \leftarrow heap.elems[p]

while poz \leq heap.len **execute**

maxChild \leftarrow -1

if poz * 2 \leq heap.len **then**

//it has a left child, assume it is the maximum

maxChild \leftarrow poz*2

end-if

if poz*2+1 \leq heap.len **and** heap.elems[2*poz+1] > heap.elems[2*poz] **th**

//it has two children and the right is greater

maxChild \leftarrow poz*2 + 1

end-if

//continued on the next slide...

Heap - remove

```
if maxChild  $\neq$  -1 and heap.elems[maxChild] > elem then  
    heap.elems[poz]  $\leftarrow$  heap.elems[maxChild]  
    poz  $\leftarrow$  maxChild  
else  
    heap.elems[poz]  $\leftarrow$  elem  
    poz  $\leftarrow$  heap.len + 1  
    //to stop the while loop  
end-if  
end-while  
end-subalgorithm
```

- Complexity:

Heap - remove

```
if maxChild  $\neq$  -1 and heap.elems[maxChild] > elem then  
    heap.elems[poz]  $\leftarrow$  heap.elems[maxChild]  
    poz  $\leftarrow$  maxChild  
else  
    heap.elems[poz]  $\leftarrow$  elem  
    poz  $\leftarrow$  heap.len + 1  
    //to stop the while loop  
end-if  
end-while  
end-subalgorithm
```

- Complexity: $O(\log_2 n)$
- Can you give an example when the complexity of the algorithm is less than $\log_2 n$ (best case scenario)?

Questions

- In a max-heap where can we find the:
 - maximum element of the array?

Questions

- In a max-heap where can we find the:
 - maximum element of the array?
 - minimum element of the array?

Questions

- In a max-heap where can we find the:
 - maximum element of the array?
 - minimum element of the array?
- Assume you have a MAX-HEAP and you need to add an operation that returns the minimum element of the heap. How would you implement this operation, using constant time and space? (Note: we only want to return the minimum, we do not want to be able to remove it).