

# DATA STRUCTURES

Heap. ADT Priority Queue.

---

*Lect. PhD. Diana-Lucia Miholca*

2022 - 2023



Babeş - Bolyai University  
Faculty of Mathematics and Computer Science

- Binary Search Trees

# Today

- Binary Heap
- ADT Priority Queue

# Binary Heap



A binary heap is a data structure whose elements are stored in a dynamic array, but that can be visualized as a binary tree.



A binary heap is a data structure that is particularly efficient for representing Priority Queues.

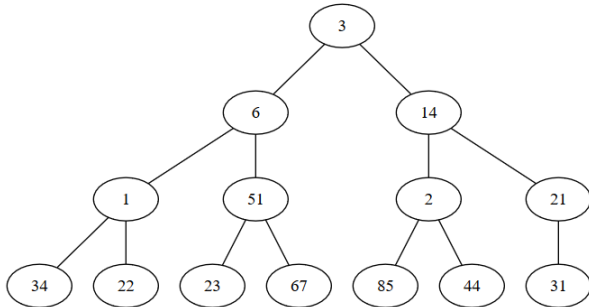
# Heap - Example



Assume that we have the following array:

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

- We can visualize this array as an **almost complete** binary tree whose root is the first element, its children are the next two elements and so on.





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 the element at index  $i$ , its children are on indexes  $2 * i$  and  $2 * i + 1$  (if  $\leq n$ )
- for the element at index  $i$  ( $i > 1$ ), its parent is at index  $[i/2]$



A **binary heap** is an array that can be visualized as a binary tree having a **heap structure** and a **heap property**.



**Heap structure:** the binary tree is *almost complete* (all the levels being completely filled, excepting the last one, which is filled 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 its children can be replaced by  $\leq$ . By doing so, we will have a *min-heap* instead of a *max-heap*.

# Heap - Examples



Are the following binary trees heaps?

- If yes, specify the relation between a node and its children.
- If not, specify if the problem is with the structure, the property or both.

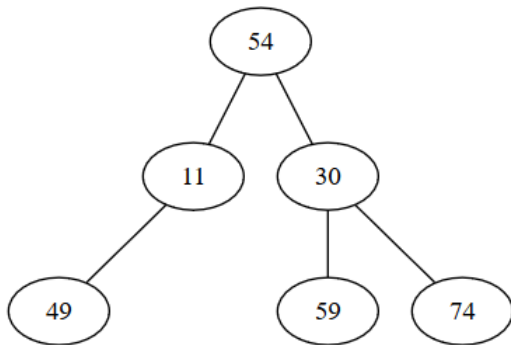


# Heap - Examples



Are the following binary trees heaps?

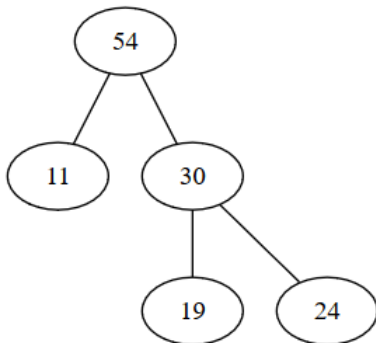
- If yes, specify the relation between a node and its children.
- If not, specify if the problem is with the structure, the property or both.



# Heap - Examples



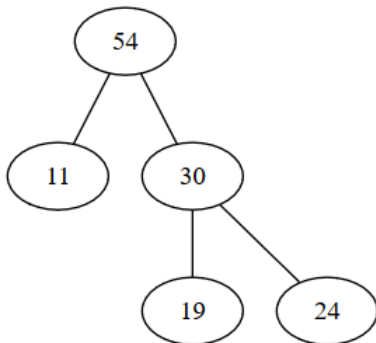
Is it a heap?



# Heap - Examples



Is it a heap?

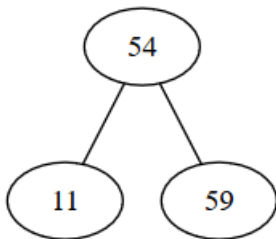


Correct answer: No

# Heap - Examples



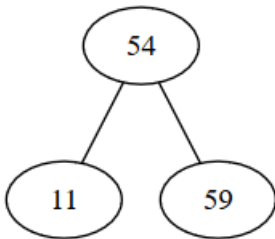
Is it a heap?



## Heap - Examples



Is it a heap?

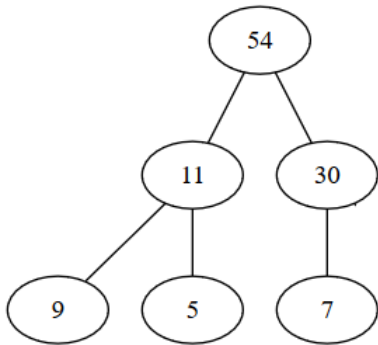


Correct answer: No

# Heap - Examples



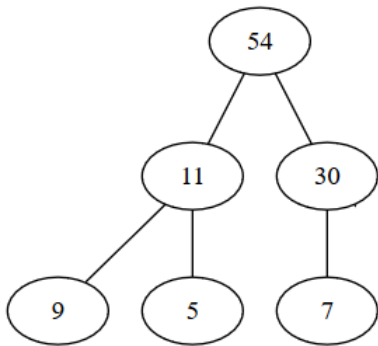
Is it a heap?



## Heap - Examples



Is it a heap?

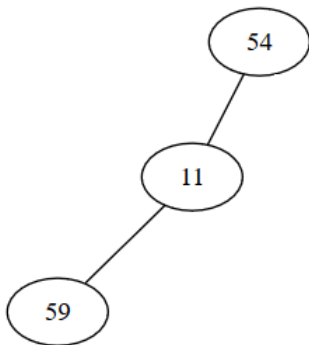


Correct answer: Yes

## Heap - Examples



Is it a heap?

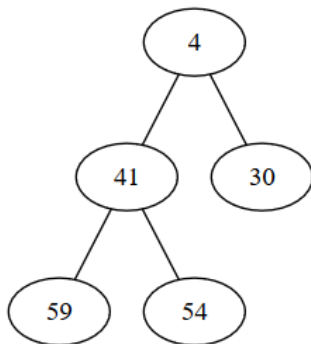




# Heap - Examples



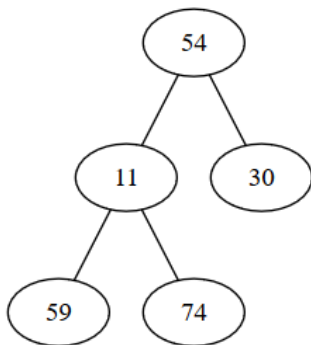
Is it a heap?



## Heap - Examples



Is it a heap?



# Heap - Examples



Is the following array a valid heap? If not, transform it into a valid heap by swapping two elements.

1. [70, 10, 50, 7, 1, 33, 3, 8]

## Heap - height



What is the height of a heap with  $n$  elements?



What is the height of a heap with  $n$  elements?



The height of a heap with  $n$  elements is  $\lceil \log_2 n \rceil$

# Heap - operations

- Specific operations:
  - **add** a new element to the heap (while keeping both the heap structure and the heap property);
  - **remove** **the root** of the heap (no other element can be removed).

# Heap - representation



## Heap representation:

Heap:

capacity: Integer

length: Integer

elems: TElem[]

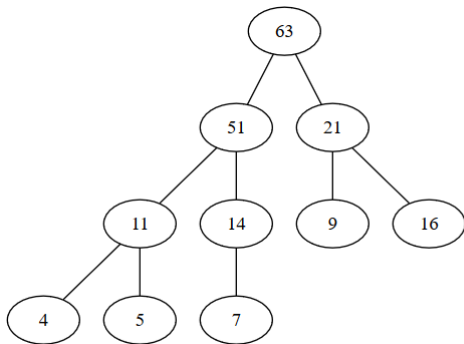


For the implementation, we assume that we have a  
MAX-HEAP.

## Heap - Add - example



Consider the following (MAX) heap:

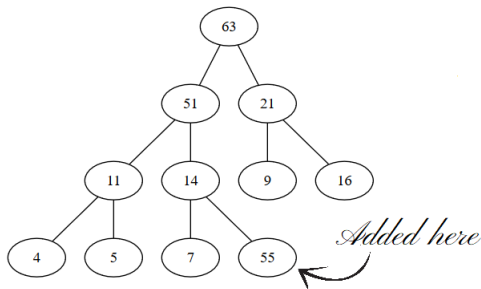


- Let's add the number 55 to the heap.

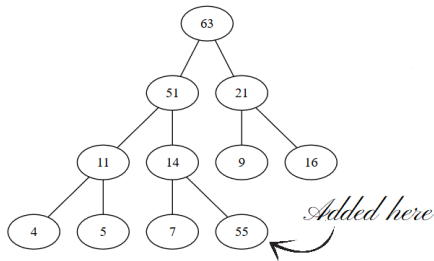


## Heap - Add - example

- 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



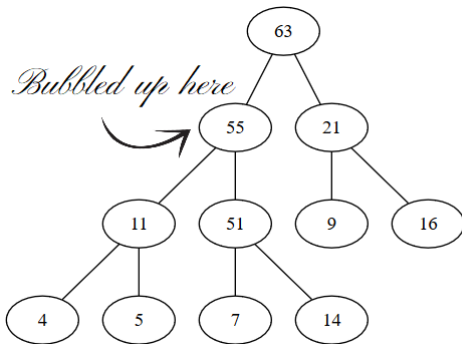
*Heap property* is not kept: 14 has as right child 55 and  $14 < 55$ .



We will perform a *bubble-up* process: we will keep swapping the value of the new node with the value of its parent node, until the heap property is met.

## Heap - Add - example

- When *bubble-up* ends:





## Adding a new element to the heap:

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

*//heap - a heap*

*//e - the element to be added*



## Adding a new element to the heap:

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

*//heap - a heap*

*//e - the element to be added*

**if** heap.length = heap.capacity **then**

    @ resize

**end-if**

heap.length  $\leftarrow$  heap.length + 1

heap.elms[heap.length]  $\leftarrow$  e

bubble-up(heap, heap.length)

**end-subalgorithm**



## Bubble-up for restoring the heap property:

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

*//heap - a heap*

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



## Bubble-up for restoring the heap property:

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

*//heap - a heap*

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

$\text{pos} \leftarrow p$

$\text{elem} \leftarrow \text{heap.elems}[p]$

$\text{parent} \leftarrow [p / 2]$

**while**  $\text{pos} > 1$  **and**  $\text{elem} > \text{heap.elems}[\text{parent}]$  **execute**

*//move parent down*

$\text{heap.elems}[\text{pos}] \leftarrow \text{heap.elems}[\text{parent}]$

$\text{pos} \leftarrow \text{parent}$

$\text{parent} \leftarrow [\text{pos}/2]$

**end-while**

$\text{heap.elems}[\text{pos}] \leftarrow \text{elem}$

**end-subalgorithm**

## Heap - add



What is the time complexity of *bubble-up*?



## Heap - add



What is the time complexity of *bubble-up*?



$O(\log_2 n)$

## Heap - add



What is the time complexity of *bubble-up*?



$O(\log_2 n)$



What is a best case scenario?

# Heap - add



What is the time complexity of *bubble-up*?



$O(\log_2 n)$



What is a best case scenario?



When the last (newly added) element is already less than or equal to its parent.

# Heap - add



What is the time complexity of *bubble-up*?



$O(\log_2 n)$



What is a best case scenario?



When the last (newly added) element is already less than or equal to its parent.



What is the time complexity of *add*?

# Heap - add



What is the time complexity of *bubble-up*?



$O(\log_2 n)$



What is a best case scenario?



When the last (newly added) element is already less than or equal to its parent.



What is the time complexity of *add*?

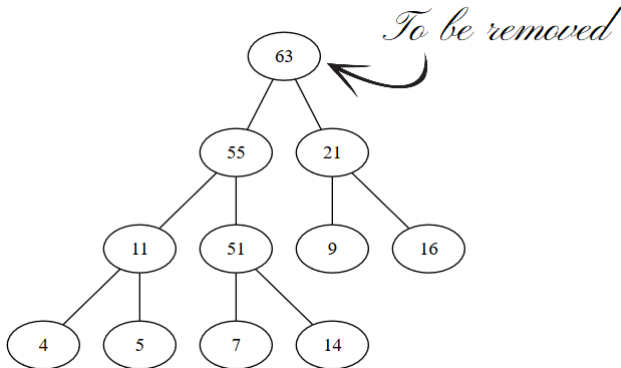


$O(\log_2 n)$  *amortized*

## Heap - Remove - example

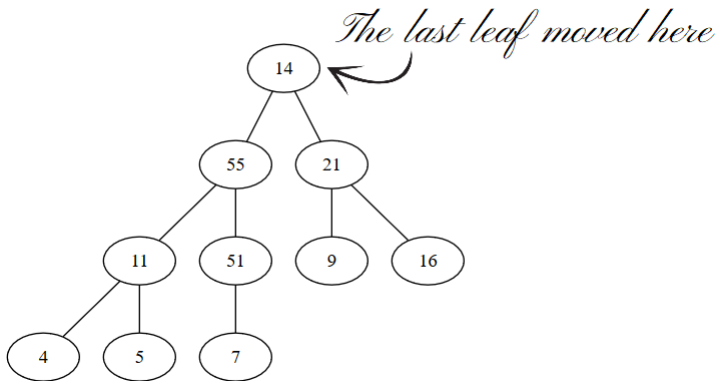


From a heap we can only remove the root element.

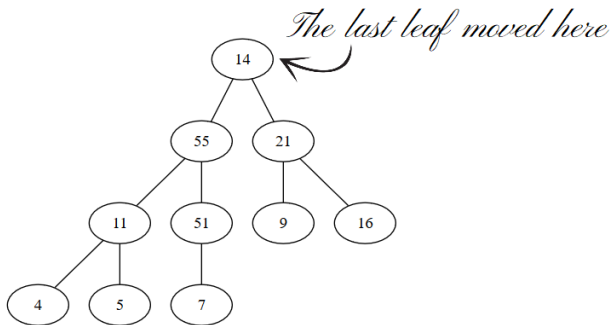


## Heap - Remove - example

- To keep the *heap structure*, when we remove the root, we replace it with the last element on the last level.



## Heap - Remove - example



The *heap property* is violated (the root is no longer the maximum element).

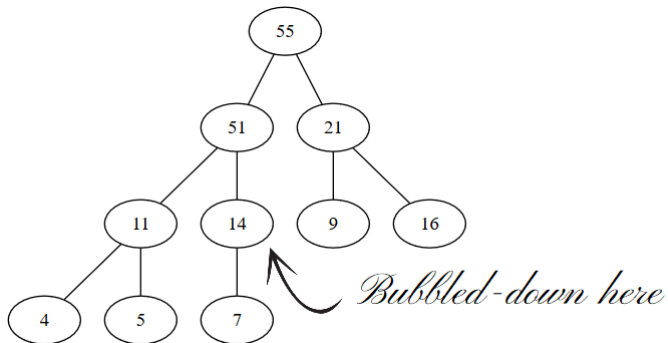


To restore the heap property, we will do a *bubble-down*: the new root element will be swapped with its maximum child, until it becomes a leaf, or until it will be greater than its both children.



# Heap - Remove - example

- When *bubble-down* ends:





## Deleting from a heap:

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

*//heap - is a heap*

*//returns the deleted element*



## Deleting from a heap:

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

*//heap - is a heap*

*//returns the deleted element*

**if** heap.length = 0 **then**

    @ error - empty heap

**end-if**

deletedElem  $\leftarrow$  heap.elems[1]

heap.elems[1]  $\leftarrow$  heap.elems[heap.length]

heap.length  $\leftarrow$  heap.length - 1

bubble-down(heap, 1)

remove  $\leftarrow$  deletedElem

**end-function**



## Bubble-down for restoring the heap property:

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



## Bubble-down for restoring the heap property:

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

pos  $\leftarrow$  p

elem  $\leftarrow$  heap.elems[p]

**while** pos  $\leq$  [heap.length/2] **execute**

maxChild  $\leftarrow$  pos\*2

**if** pos\*2+1  $\leq$  heap.length **and** heap.elems[2\*pos+1] > heap.elems[2\*pos] **then**

*//it has two children and the right is greater*

maxChild  $\leftarrow$  pos\*2 + 1

**end-if**

**if** heap.elems[maxChild] > elem **then**

tmp  $\leftarrow$  heap.elems[pos]

heap.elems[pos]  $\leftarrow$  heap.elems[maxChild]

heap.elems[maxChild]  $\leftarrow$  tmp

pos  $\leftarrow$  maxChild

**else**

pos  $\leftarrow$  heap.length + 1 *//to stop the while loop*

**end-if**

**end-while**

**end-subalgorithm**



What is the time complexity of *bubble-down*?



What is the time complexity of *bubble-down*?



$O(\log_2 n)$

## Heap - remove



What is the time complexity of *bubble-down*?



$O(\log_2 n)$



What is the time complexity of *remove*?





What is the time complexity of *bubble-down*?



$O(\log_2 n)$



What is the time complexity of *remove*?



$O(\log_2 n)$



## Real-world applications of heaps:



### Graph algorithms

- Used in Dijkstra's algorithm (shortest path) or Prim's algorithm (minimum spanning tree)



### Data compression

- Huffman coding



### Sorting (Heap-Sort)

- Used by embedded systems such as Linux Kernel



### Order statistics

- To find the  $k$ th smallest or largest element in an array



### Resources allocation

- To efficiently allocate resources in a system, such as memory blocks or CPU time, by processing requests in order of priority



Source: VectorStock.com/21245929



Consider the following queue in front of the Emergency Room.



Who should be the next person checked by the doctor?



The ADT Priority Queue is a container in which each element has an associated *priority*.



In a Priority Queue, the access to the elements is restricted: we can access only the element with the highest priority.



Due to the restricted access, a Priority Queue is said to have a **HPF - Highest Priority First** policy.



In order to work in a more general manner, we can define a relation  $\mathcal{R}$  on the set of priorities:  $\mathcal{R} : TPriority \times TPriority$ .



When we refer to *the element with the highest priority* we mean the highest priority given by the relation  $\mathcal{R}$ .

- If the relation  $\mathcal{R} = "\geq"$ , the element with the *highest priority* is the one for which the value of the priority is the largest.
- If the relation  $\mathcal{R} = "\leq"$ , the element with the *highest priority* is the one for which the value of the priority is the lowest.

# Priority Queue - Domain

- The domain of the ADT Priority Queue:

$$\mathcal{PQ} = \{pq \mid pq \text{ is a priority queue with elements } (e, p), e \in TElem, p \in TPriority\}$$

- **init** (pq, R)
  - **descr:** creates a new empty priority queue
  - **pre:**  $R$  is a relation over the priorities,  $R : \text{TPriority} \times \text{TPriority}$
  - **post:**  $pq \in \mathcal{PQ}$ ,  $pq$  is an empty priority queue

# Priority Queue - Interface

- `destroy(pq)`
  - **descr:** destroys a priority queue
  - **pre:**  $pq \in \mathcal{PQ}$
  - **post:**  $pq$  has been destroyed



- `push(pq, e, p)`
  - **descr:** pushes (adds) a new element to the priority queue
  - **pre:**  $pq \in \mathcal{PQ}, e \in TElem, p \in TPriority$
  - **post:**  $pq' \in \mathcal{PQ}, pq' = pq \oplus (e, p)$

# Priority Queue - Interface

- **pop(pq)**
  - **descr:** pops (removes) from the priority queue the element with the highest priority. It returns both the element and its priority
  - **pre:**  $pq \in \mathcal{PQ}$ ,  $pq$  is not empty
  - **post:**  $pop = (e, p)$ ,  $e \in TElem$ ,  $p \in TPriority$ ,  $e$  is the element with the highest priority from  $pq$ ,  $p$  is its priority.  
 $pq' \in \mathcal{PQ}$ ,  $pq' = pq \ominus (e, p)$
  - **throws:** an exception if the priority queue is empty.

# Priority Queue - Interface

- **top(pq)**
  - **descr:** returns from the priority queue the element with the highest priority and its priority. It does not modify the priority queue.
  - **pre:**  $pq \in \mathcal{PQ}$ ,  $pq$  is not empty
  - **post:**  $top = (e, p)$ ,  $e \in TElem$ ,  $p \in TPriority$ ,  $e$  is the element with the highest priority from  $pq$ ,  $p$  is its priority.
  - **throws:** an exception if the priority queue is empty.

# Priority Queue - Interface

- `isEmpty(pq)`
  - **Description:** checks if the priority queue is empty (it has no elements)
  - **Pre:**  $pq \in \mathcal{PQ}$
  - **Post:**

$$isEmpty = \begin{cases} \text{true, if } pq \text{ has no elements} \\ \text{false, otherwise} \end{cases}$$



Priority queues cannot be iterated, so they don't have an *iterator* operation.

# Priority Queue - Representation



Data structures can be used to represent a Priority Queue:

- Dynamic Array
- Linked List
- Binary Heap

# Priority Queue - Representation



If we opt for a Dynamic Array or a Linked List we have to decide in which order to store the elements:

- We can keep the elements ordered by their priorities.




Where would you put the element with the highest priority?

# Priority Queue - Representation



If we opt for a Dynamic Array or a Linked List we have to decide in which order to store the elements:

- We can keep the elements ordered by their priorities.
  -  Where would you put the element with the highest priority?
    - ✓ At the beginning for linked lists and at the end for arrays.
- We also can keep the elements in the order in which they have been inserted.



# Priority Queue - Representation



Complexity of the main operations for the two representation options:

Operation	Sorted	Non-sorted
push		

# Priority Queue - Representation



Complexity of the main operations for the two representation options:

Operation	Sorted	Non-sorted
push	$O(n)$	

# Priority Queue - Representation



Complexity of the main operations for the two representation options:

Operation	Sorted	Non-sorted
push	$O(n)$	$\Theta(1)$
pop		

# Priority Queue - Representation



Complexity of the main operations for the two representation options:

Operation	Sorted	Non-sorted
push	$O(n)$	$\Theta(1)$
pop	$\Theta(1)$	

# Priority Queue - Representation



Complexity of the main operations for the two representation options:

Operation	Sorted	Non-sorted
push	$O(n)$	$\Theta(1)$
pop	$\Theta(1)$	$\Theta(n)$
top		

# Priority Queue - Representation



Complexity of the main operations for the two representation options:

Operation	Sorted	Non-sorted
push	$O(n)$	$\Theta(1)$
pop	$\Theta(1)$	$\Theta(n)$
top	$\Theta(1)$	

# Priority Queue - Representation



Complexity of the main operations for the two representation options:

Operation	Sorted	Non-sorted
push	$O(n)$	$\Theta(1)$
pop	$\Theta(1)$	$\Theta(n)$
top	$\Theta(1)$	$\Theta(n)$

# Priority Queue - Representation on a binary heap



When representing a Priority Queue using a Max Binary Heap:



When an element is pushed to the priority queue, it is simply added to the heap.



When an element is popped from the priority queue, the root is removed from the heap.



*Top* simply returns the pair in the root of the heap.



# Priority Queue - Representation



Let's complete our table with the complexity of the operations if we use a heap as representation:

Operation	Sorted	Non-sorted	Heap
push	$O(n)$	$\Theta(1)$	$O(\log_2 n)$
pop	$\Theta(1)$	$\Theta(n)$	$O(\log_2 n)$
top	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

# Representing a Priority Queue using a Max-Heap



## A Priority Queue element's representation:

PQElement:

e: TElem

p: TPriority



## Priority Queue representation using a Max-Heap:

PriorityQueue:

capacity: Integer

length: Integer

elems: PQElement[]

R: TPriority  $\times$  TPriority  $\rightarrow$  {True, False}

# Priority Queue represented on Heap - init



## Creating an empty PQ:

**subalgorithm** init(pq, R) **is:**

*//R\$ TPriority  $\times$  TPriority  $\rightarrow \{\text{True}, \text{False}\}$*

*pq.R  $\leftarrow$  R*

*pq.length  $\leftarrow$  0*

*pq.capacity  $\leftarrow$  @ an initial capacity*

*@ allocate pq.elems of capacity pq.capacity*

**end-subalgorithm**

# Priority Queue represented on Heap - add



## Inserting a new element into the PQ:

**subalgorithm** push(pq, e, p) **is:**

*//pq - a pq*

*//e - the element to be added*

*//p - the priority of the element to be added*

**if** pq.length = pq.capacity **then**

    @ resize

**end-if**

pq.length  $\leftarrow$  pq.length + 1

pq.elems[pq.length].e  $\leftarrow$  e

pq.elems[pq.length].p  $\leftarrow$  p

bubble-up(pq, pq.length)

**end-subalgorithm**

# Priority Queue represented on Heap - push



## Bubble-up for restoring the heap property:

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

*//pq - a Priority Queue*

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

pos  $\leftarrow$  p

elem  $\leftarrow$  pq.elems[p]

parent  $\leftarrow$  [p / 2]

**while** pos > 1 **and not**(pq.R(pq.elems[parent].p, elem.p)) **execute**

*//move parent down*

pq.elems[pos]  $\leftarrow$  pq.elems[parent]

pos  $\leftarrow$  parent

parent  $\leftarrow$  [pos/2]

**end-while**

pq.elems[pos]  $\leftarrow$  elem

**end-subalgorithm**

## Priority Queue represented on Heap - push - complexity



What is the time complexity of *push*?

## Priority Queue represented on Heap - push - complexity



What is the time complexity of *push*?



$O(\log_2 n)$  amortized

# Priority Queue represented on Heap - pop



## Popping from a Priority Queue:

**function** pop(pq) **is:**

*//pq - is a pq*

**if** pq.length = 0 **then**

    @ error - empty Priority Queue

**end-if**

deletedElem  $\leftarrow$  (pq.elems[1].e, pq.elems[1].p)

pq.elems[1]  $\leftarrow$  pq.elems[pq.length]

pq.length  $\leftarrow$  pq.length - 1

bubble-down(pq, 1)

remove  $\leftarrow$  deletedElem

**end-function**





## Bubble-down for restoring the heap property:

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

pos  $\leftarrow$  p

elem  $\leftarrow$  pq.elems[p]

**while** pos  $\leq$  [pq.length/2] **execute**

maxChild  $\leftarrow$  pos\*2

**if** pos\*2+1  $\leq$  pq.length **and** pq.R(pq.elems[2\*pos+1].p, pq.elems[2\*pos].p)

**then** *//it has two children and the right is greater*

maxChild  $\leftarrow$  pos\*2 + 1

**end-if**

**if** pq.R(pq.elems[maxChild], elem) **then**

tmp  $\leftarrow$  pq.elems[pos]

pq.elems[pos]  $\leftarrow$  pq.elems[maxChild]

pq.elems[maxChild]  $\leftarrow$  tmp

pos  $\leftarrow$  maxChild

**else**

pos  $\leftarrow$  pq.length + 1 *//to stop the while loop*

**end-if**

**end-while**

**end-subalgorithm**

## Priority Queue represented on Heap - pop - complexity



What is the time complexity of *pop*?

## Priority Queue represented on Heap - pop - complexity



What is the time complexity of *pop*?



$O(\log_2 n)$

# Priority Queue represented on Heap - top



## Popping from a Priority Queue:

**function** top(pq) **is:**

*//pq - is a pq*

**if** pq.length = 0 **then**

    @ error - empty Priority Queue

**end-if**

top  $\leftarrow$  (pq.elems[1].e, pq.elems[1].p)

**end-function**



Complexity:

# Priority Queue represented on Heap - top



## Popping from a Priority Queue:

**function** top(pq) **is:**

*//pq - is a pq*

**if** pq.length = 0 **then**

    @ error - empty Priority Queue

**end-if**

top  $\leftarrow$  (pq.elems[1].e, pq.elems[1].p)

**end-function**



Complexity:  $\Theta(1)$

# Priority Queue represented on Heap - isEmpty



## Checking if a Priority Queue is empty:

**function** isEmpty(pq) **is:**

*//pq - is a pq*

**if** pq.length = 0 **then**

isEmpty  $\leftarrow$  True

**else**

isEmpty  $\leftarrow$  False

**end-if**

**end-function**



Complexity:

# Priority Queue represented on Heap - isEmpty



## Checking if a Priority Queue is empty:

**function** isEmpty(pq) **is:**

*//pq - is a pq*

**if** pq.length = 0 **then**

isEmpty  $\leftarrow$  True

**else**

isEmpty  $\leftarrow$  False

**end-if**

**end-function**



Complexity:  $\Theta(1)$

# Priority Queue in programming languages



## ADT Priority Queue in programming languages:

- PriorityQueue in Java
  - represented using the Heap data structure
- PriorityQueue in Python (queue module)
  - represented using the Heap data structure
- priority\_queue in C++ STL
  - represented using the Heap data structure



► THANK  
E  
A  
P

Y P  
R I Q  
O U  
R E  
I U  
T E  
Y