DATA STRUCTURES (AND ALGORITHMS)

Heap. ADT Priority Queue.

Lect. PhD. Diana-Lucia Miholca

2023 - 2024

Babeş - Bolyai University
Faculty of Mathematics and Computer Science

In the previous lecture...

- Binary Search Trees

Today

- Binary Heap
- ADT Priority Queue

Binary Heap - Introduction

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

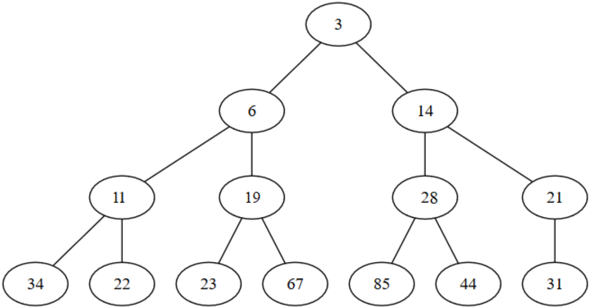
A binary heap is a data structure that is particularly efficient for representing Priority Queues.

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	11	19	28	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.



Heap - Properties

If the elements of the array are: $a_1, a_2, a_3, ..., a_n$, 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 $[i/2]$

Heap - Definition

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

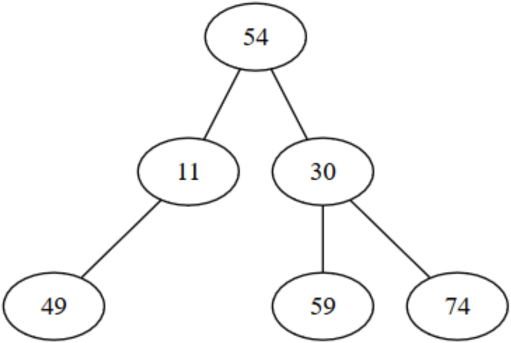
Are the following binary trees heaps?

- 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

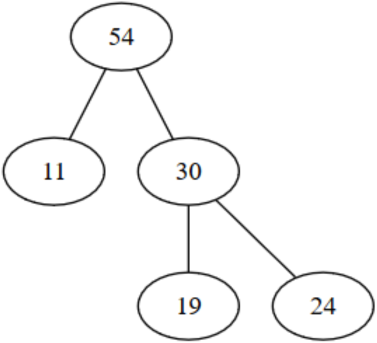
Are the following binary trees heaps?

- 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.



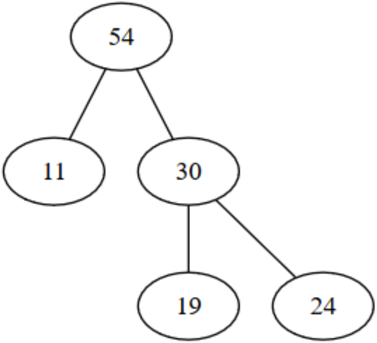
Heap - Examples

Is this a heap?



Heap - Examples

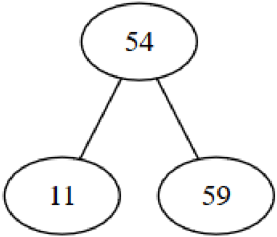
Is this a heap?



Correct answer: No

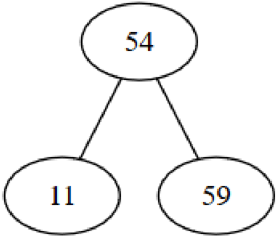
Heap - Examples

Is this a heap?



Heap - Examples

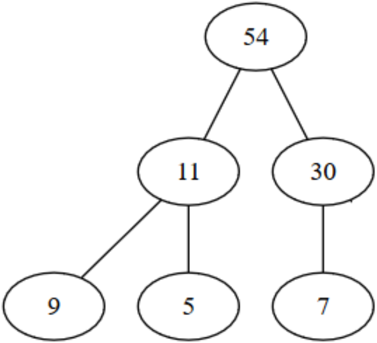
Is this a heap?



Correct answer: No

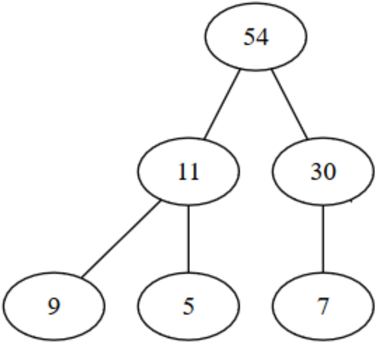
Heap - Examples

Is this a heap?



Heap - Examples

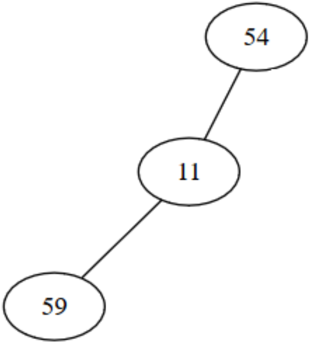
Is this a heap?



Correct answer: Yes

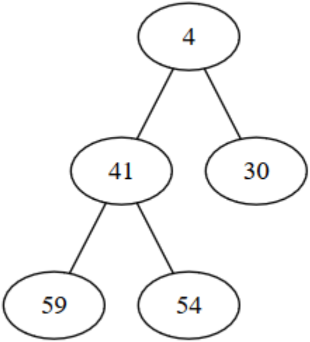
Heap - Examples

Is this a heap?



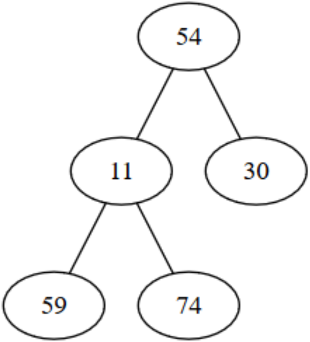
Heap - Examples

Is this a heap?



Heap - Examples

Is this a heap?



Heap - Examples

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

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

Heap - Height

What is the height of a heap with n elements?

Heap - Height

What is the height of a heap with n elements?

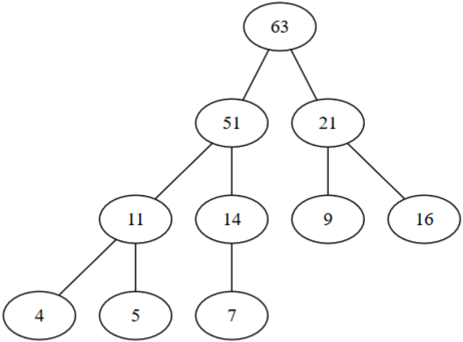
The height of a heap with n elements is $[log_2 n]$

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 - Add - Example

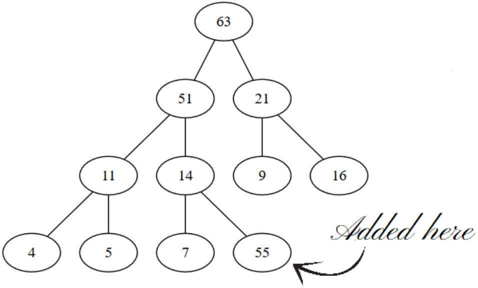
Consider the following (MAX) heap:



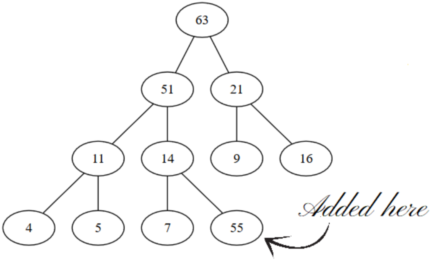
- Let's add the number 55 to the heap.

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

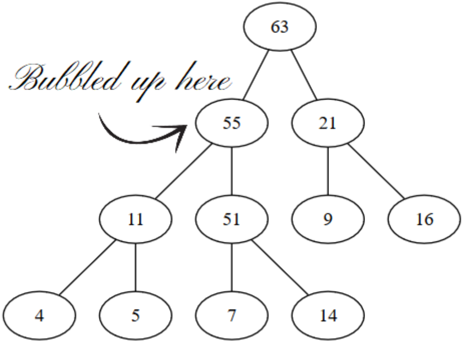


Heap property is not kept: 14 has as right child 55 and $14 < 55$.

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:



Heap - Representation

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

Heap representation:

Heap:

Heap - Representation

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

Heap representation:

Heap:
- capacity: Integer
- length: Integer
- elems: TComp[]

Heap - Add

Adding a new element to the heap:

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

//heap - a heap

//e - the element to be added

Heap - Add

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 ← heap.length + 1
    heap.elems[heap.length] ← e
    bubble-up(heap, heap.length)
end-subalgorithm
```

Heap - Add

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

Heap - Add

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
    pos ← p
    elem ← heap.elems[p]
    parent ← [p / 2]
    while pos > 1 and elem > heap.elems[parent] execute
        //move parent down
        heap.elems[pos] ← heap.elems[parent]
        pos ← parent
        parent ← [pos/2]
    end-while
    heap.elems[pos] ← 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?

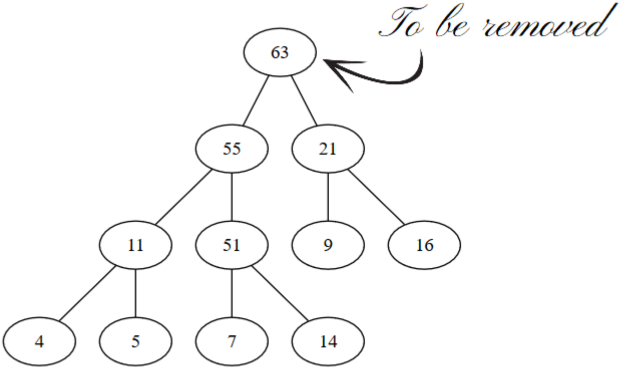
When the last (newly added) element is already less than or equal to its parent.

What is the time complexity of *add*?

$O(log_2 n)$ *amortized*

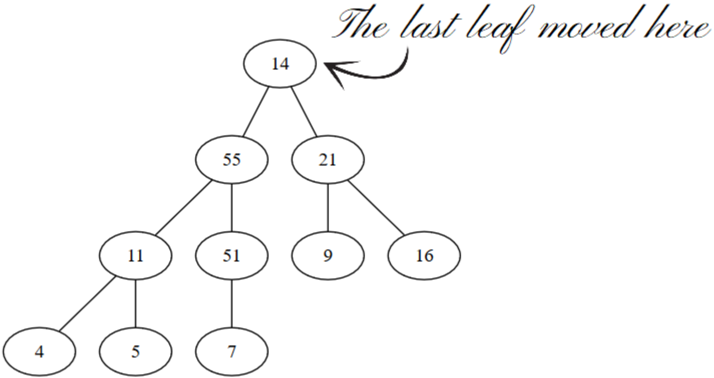
Heap - Remove - Example

From a heap we can only remove the root element.

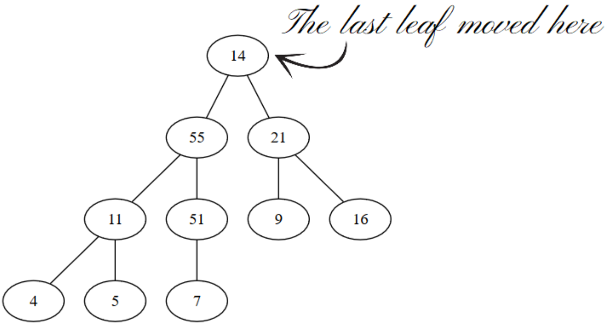


Heap - Remove - Example

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



Heap - Remove - Example

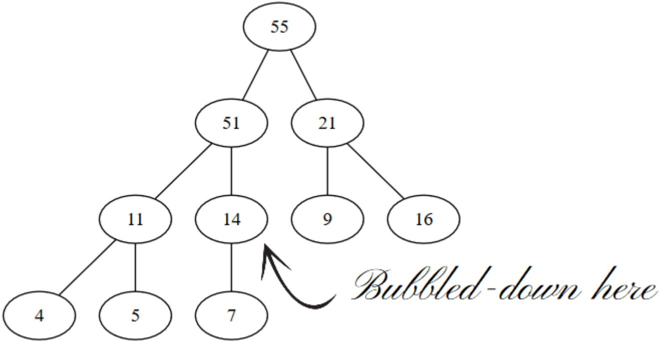


The *heap property* is violated (the root is no longer the maximum element).

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:



Heap - Remove

Deleting from a heap:

function remove(heap) **is:**

//heap - is a heap

//returns the deleted element

Heap - Remove

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 ← heap.elems[1]
    heap.elems[1] ← heap.elems[heap.length]
    heap.length ← heap.length - 1
    bubble-down(heap, 1)
    remove ← deletedElem
end-function
```

Heap - Remove

Bubble-down for restoring the heap property:

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

Heap - Remove

Bubble-down for restoring the heap property:

```
subalgorithm bubble-down(heap, p) is:
    pos ← p
    elem ← heap.elems[p]
    while pos ≤ [heap.length/2] execute
        maxChild ← pos*2
        if pos*2+1 ≤ heap.length and heap.elems[2*pos+1]>heap.elems[2*pos] then
        //it has two children and the right is greater
            maxChild ← pos*2 + 1
        end-if
        if heap.elems[maxChild] > elem then
            tmp ← heap.elems[pos]
            heap.elems[pos] ← heap.elems[maxChild]
            heap.elems[maxChild] ← tmp
            pos ← maxChild
        else
            pos ← heap.length + 1 //to stop the while loop
        end-if
    end-while
end-subalgorithm
```

Heap - Remove

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

Heap - Remove

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*?

Heap - Remove

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

$O(log_2 n)$

What is the time complexity of *remove*?

$O(log_2 n)$

Heap - Applications

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 kth 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?

ADT Priority Queue

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.

ADT Priority Queue

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:

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

Priority Queue - Interface

- init (pq, R)
  - **descr:** creates a new empty priority queue
  - **pre:** R is a relation over the priorities, $R : TPriority \times TPriority$
  - **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

Priority Queue - Interface

- 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)
  - **descr:** checks if the priority queue is empty (it has no elements)
  - **pre:** $pq \in \mathcal{PQ}$
  - **post:**

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

Priority Queue - Interface

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

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

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*

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.

Representing a Priority Queue using a Max-Heap

A Priority Queue element's representation:

PQElement:

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:

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:**

Priority Queue represented on Heap - init

Creating an empty PQ:

subalgorithm init(pq, R) **is:**
//RȘ TPriority $\times$ *TPriority* $\rightarrow$ {True, 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 ← pq.length + 1
    pq.elems[pq.length].e ← e
    pq.elems[pq.length].p ← 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 ← p
    elem ← pq.elems[p]
    parent ← [p / 2]
    while pos > 1 and not(pq.R(pq.elems[parent].p, elem.p)) execute
        //move parent down
        pq.elems[pos] ← pq.elems[parent]
        pos ← parent
        parent ← [pos/2]
    end-while
    pq.elems[pos] ← 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 ← (pq.elems[1].e, pq.elems[1].p)
    pq.elems[1] ← pq.elems[pq.length]
    pq.length ← pq.length - 1
    bubble-down(pq, 1)
    remove ← deletedElem
end-function
```

Heap - remove

Bubble-down for restoring the heap property:

```
subalgorithm bubble-down(pq, p) is:
    pos ← p
    elem ← pq.elems[p]
    while pos ≤ [pq.length/2] execute
        maxChild ← pos*2
        if pos*2+1 ≤ 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 ← pos*2 + 1
        end-if
        if pq.R(pq.elems[maxChild], elem) then
            tmp ← pq.elems[pos]
            pq.elems[pos] ← pq.elems[maxChild]
            pq.elems[maxChild] ← tmp
            pos ← maxChild
        else
            pos ← 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

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 ← (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 ← (pq.elems[1].e, pq.elems[1].p)
end-function

Complexity: $\Theta(1)$

Priority Queue represented on Heap - isEmpty

Cheking if a Priority Queue is empty:

function isEmpty(pq) **is:**
//pq - is a pq
    if pq.length = 0 **then**
        isEmpty ← True
    else
        isEmpty ← 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

Priority Queue - Applications

Applications of priority queues:

Graph algorithms

- Implementation of Priority Queues used in Dijkstra’s algorithm (shortest path) or Prim’s algorithm (minimum spanning tree)

Data compression

- Huffman coding

Resources allocation

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

Artificial Intelligence

- A* search algorithm

Operating Systems

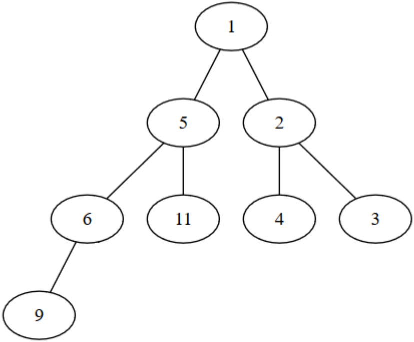
- In the load balancing algorithms

Heap-sort - Naive approach

- The initial sequence: [6, 1, 3, 9, 11, 4, 2, 5]

Heap-sort - Naive approach

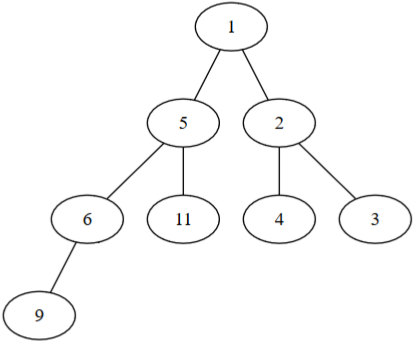
- The initial sequence: [6, 1, 3, 9, 11, 4, 2, 5]
  - If we **add all elements into a min-heap**, we get:



Heap-sort - Naive approach

- The initial sequence: [6, 1, 3, 9, 11, 4, 2, 5]

  - If we **add all elements into a min-heap**, we get:



- Then, if we **remove all the elements**, one-by-one, we obtain: 1, 2, 3, 4, 5, 6, 9, 11.

Heap-sort - Naive approach

What is the time complexity of the heap-sort algorithm previously described?

Heap-sort - Naive approach

What is the time complexity of the heap-sort algorithm previously described?

✓ $O(n \cdot log_2 n)$

Heap-sort - Naive approach

What is the time complexity of the heap-sort algorithm previously described?

✓ $O(n \cdot log_2 n)$

What is the extra space complexity of the heap-sort algorithm previously described?

Heap-sort - Naive approach

What is the time complexity of the heap-sort algorithm previously described?

✓ $O(n \cdot log_2 n)$

What is the extra space complexity of the heap-sort algorithm previously described?

✓ $\Theta(n)$

Heap-sort - Better approach

- We start by transforming the unsorted array into a max-heap.
  - The second half of the array contain leaves, so they can be left where they are.
  - Starting from the last non-leaf element (and going towards the beginning of the array), we just call *bubble-down* for every element.

Heap-sort - Better approach

Heap-sort - better approach:

function build-max-heap(a, n) **is:**
//a - an array of lenght n
    heap.elems $\leftarrow$ a
    heap.length $\leftarrow$ n
    heap.capacity $\leftarrow$ n
    for i $\leftarrow$ [n/2], 1, -1 **execute**
        bubble-down(heap, i)
    end-for
    build-max-heap $\leftarrow$ heap
end-function

Time complexity: $O(n)$

Heap-sort - Better approach

- After transforming the unsorted array into a max-heap:
  - The maximum element is stored in the root, so at index 1 $\Rightarrow$ we swap it with the one at the last index
  - We discard the last element in the heap, by decrementing the length of the heap
  - The root element is the only that may violate the heap property $\Rightarrow$ we *bubble-down* it
  - We repeat the process until the length of the heap is 1

Heap-sort - Better approach

Heap-sort - better approach:

subalgorithm heapsort(a, n) **is:**
//a - an array of lenght n
    heap ← build-max-heap(a, n)
    for i ← n, 2, -1 **execute**
        aux ← a[i]
        a[i] ← a[1]
        a[1] ← aux
        heap.length ← heap.length-1
        bubbleDown(a, 1)
    end-for
end-subalgorithm

Heap-sort - Complexity

- Time complexity of this heap-sort is $O(n \cdot log_2 n)$.
  - *build-max-heap* runs in $O(n)$.
  - *bubble-down* runs in $O(log_2 n)$ and we call it $n - 1$ times.
- Extra-space complexity of this approach is $\Theta(1)$.

Bibliography

- David M. Mount, *Lecture notes for the course Data Structures* (CMSC 420), at the Dept. of Computer Science, University of Maryland, College Park, 2001
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Third Edition, The MIT Press, 2009
- Narasimha Karumanchi, *Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles*, Fifth Edition, 2016
- Clifford A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*, Third Edition, 2010

Thank you

- THANK YOU

HEAP

PRIORITY

QUEUE