Heaps

Advanced Programming and Algorithmic Design

Alberto Casagrande
Email: acasagrande@units.it

a.a. 2018/2019

Heaps

Abstract Data Types to store values, totally ordered w.r.t. $\preceq$

They (efficiently) support the following tasks:

- building a heap from a set of data
- finding the minimum w.r.t. $\preceq$
- extracting the minimum w.r.t. $\preceq$
- decreasing [illegible] one of the values w.r.t. $\preceq$
- inserting a new value

Heaps

Abstract Data Types to store values, totally ordered w.r.t. $\preceq$

They (efficiently) support the following tasks:

- building a heap from a set of data
- finding the minimum w.r.t. $\preceq$
- extracting the minimum w.r.t. $\preceq$
- decreasing the one of the values w.r.t. $\preceq$
- inserting a new value

A min-heap is a heap s.t. $\preceq$ is $\leq$

A max-heap is a heap s.t. $\preceq$ is $\geq$ here the min is a max

Heaps

They can be used to implement priority queues

The next element to be extracted minimizes a priority *criterion*

E.g., In emergencies, more serious patients must be served first

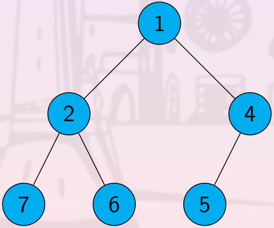
Their conditions may change in time and become more and more serious

Binary Heaps

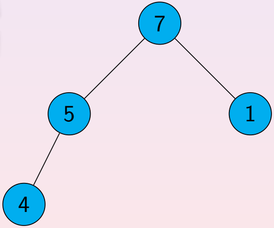
Binary Heaps

Are nearly complete binary trees (i.e., it is complete up to the second-last level and all leaves of the last level are on the left)

The relation $parent(p) \preceq p$ holds for any node (heap property)



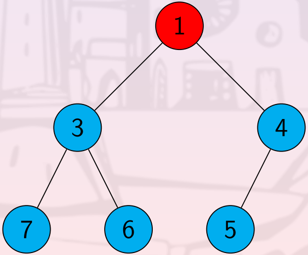
Min-heap



Max-heap

A Possible Representation (Not the Only One)

Use an **array**: the first position stores the root key



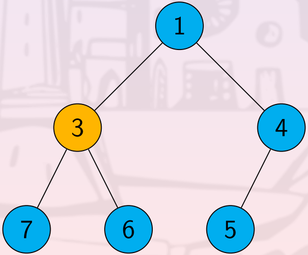
Min-heap

1	2	3	4	5	6
$\boxed{1}$	3	4	7	6	5

A Possible Representation (Not the Only One)

Use an **array**: the first position stores the root key

The *i*-th position of the array represents a node whose:



Min-heap

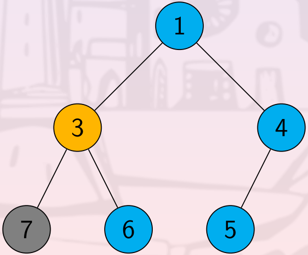
1	2	3	4	5	6
1	3	4	7	6	5

A Possible Representation (Not the Only One)

Use an **array**: the first position stores the root key

The *i*-th position of the array represents a node whose:

- left child has index $2 * i$



Min-heap

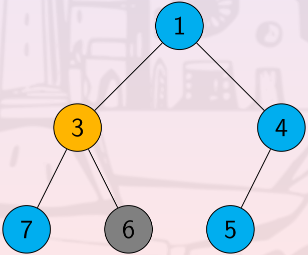
1	2	3	4	5	6
1	3	4	7	6	5

A Possible Representation (Not the Only One)

Use an **array**: the first position stores the root key

The i-th position of the array represents a node whose:

- left child has index $2 * i$
- right child has index $2 * i + 1$



Min-heap

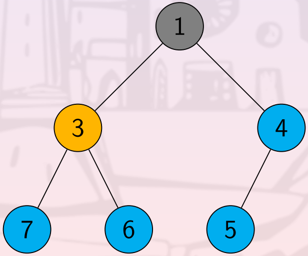
1	2	3	4	5	6
1	3	4	7	6	5

A Possible Representation (Not the Only One)

Use an **array**: the first position stores the root key

The i-th position of the array represents a node whose:

- left child has index $2 * i$
- right child has index $2 * i + 1$
- parent has index $\lfloor i/2 \rfloor$



Min-heap

1	2	3	4	5	6
1	3	4	7	6	5

drawback: tough to deal with node addiction

Array-based Representation: Few Useful Functions

H.size will denote the heap size

```
def LEFT(i):
  return 2*i
enddef

def RIGHT(i):
  return 2*i+1
enddef

def PARENT(i):
  return floor(i/2)
enddef
```

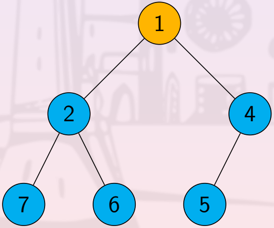
```
def GET_ROOT():
  return 1
enddef

def IS_ROOT(i):
  return i == 1
enddef

def IS_VALID_NODE(H, i):
  return H.size ≥ i
enddef
```

Finding the Minimum

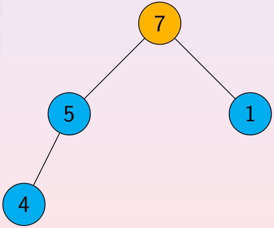
Finding the Minimum

The minimum w.r.t. $\preceq$ is in the root of the heap

If this was not the case, the heap property did not hold



Min-heap



Max-heap

Finding the Minimum: Pseudo-Code

The minimum w.r.t. $\preceq$ is the root's key

```
def HEAP_MINIMUM(H):
  return H.root
enddef
```

For array-based representation, we can rephrase it as...

```
def HEAP_MINIMUM(H):
  return H[0]
enddef
```

In both the cases, the complexity is $\Theta(1)$

Removing the Minimum

Removing the Minimum

We must preserve both:

- heap topological structure
- heap property

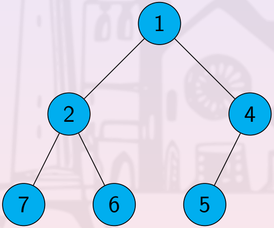
Removing the Minimum

We must preserve both:

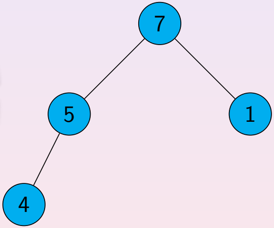
- heap topological structure
- heap property

Removing the Minimum and Preserving Topology

Replace the root's key by that of the rightmost leaf of the last level



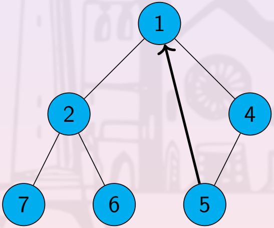
Min-heap



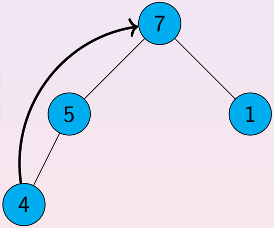
Max-heap

Removing the Minimum and Preserving Topology

Replace the root's key by that of the rightmost leaf of the last level



Min-heap

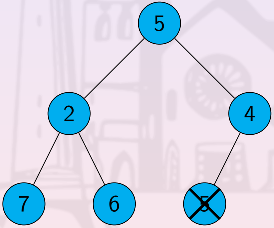


Max-heap

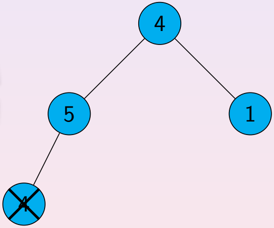
Removing the Minimum and Preserving Topology

Replace the root's key by that of the rightmost leaf of the last level

Delete the the rightmost leaf of the last-level



Min-heap

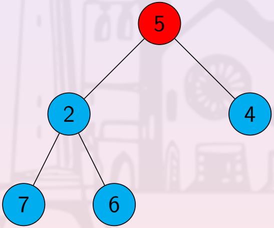


Max-heap

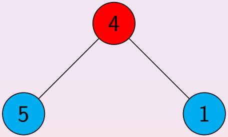
Removing the Minimum and Preserving Topology

Replace the root's key by that of the rightmost leaf of the last level

Delete the the rightmost leaf of the last-level



Min-heap



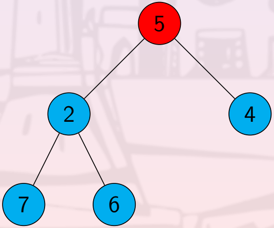
Max-heap

The heap property may be lost (only in one point)!

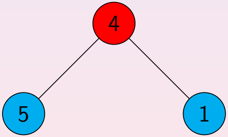
root

Restoring the Heap Property (HEAPIFY)

- find the node n, among the root and its children, whose key is minimum w.r.t. $\preceq$



Min-heap

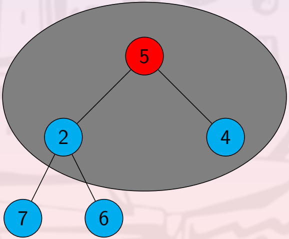


Max-heap

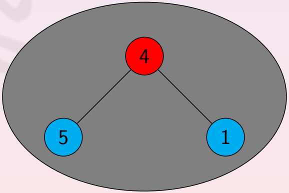
Restoring the Heap Property (HEAPIFY)

- find the node n, among the root and its children, whose key is minimum w.r.t. $\preceq$

it'll be one of the first 2 children



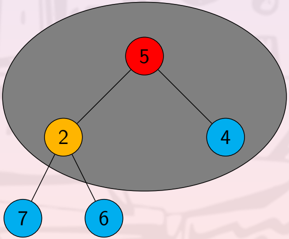
Min-heap



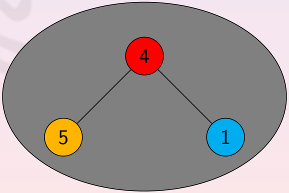
Max-heap

Restoring the Heap Property (HEAPIFY)

- find the node n, among the root and its children, whose key is minimum w.r.t. $\preceq$



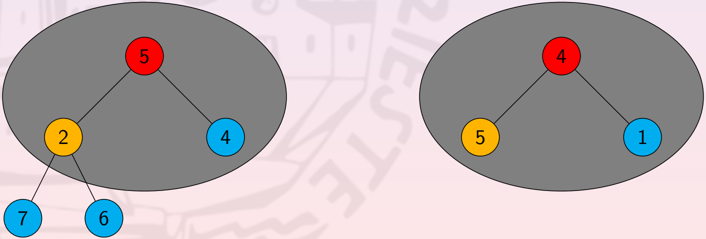
Min-heap



Max-heap

Restoring the Heap Property (HEAPIFY)

- find the node n, among the root and its children, whose key is minimum w.r.t. $\preceq$
- if the root's key is minimum, done!

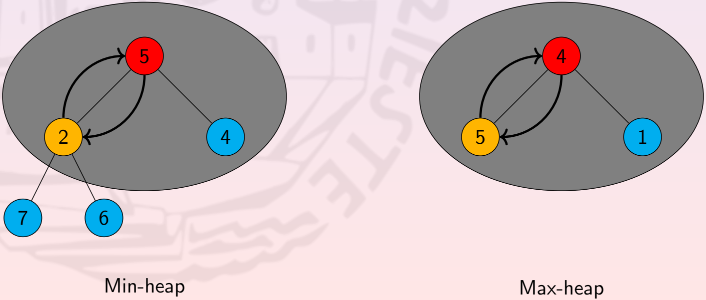


Min-heap

Max-heap

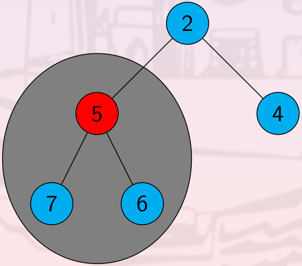
Restoring the Heap Property (HEAPIFY)

- find the node n, among the root and its children, whose key is minimum w.r.t. $\preceq$
- if the root's key is minimum, done!
- otherwise, swap n's and root's keys

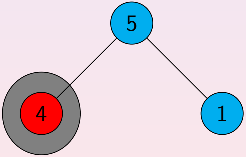


Restoring the Heap Property (HEAPIFY)

- find the node n, among the root and its children, whose key is minimum w.r.t. $\preceq$
- if the root's key is minimum, done!
- otherwise, swap n's and root's keys
- repeat on the sub-tree rooted on n



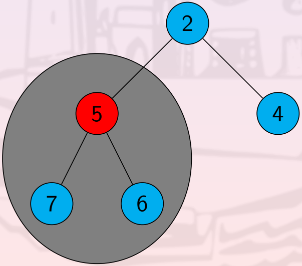
Min-heap



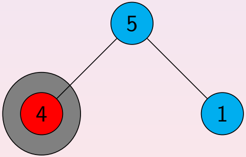
Max-heap

Restoring the Heap Property (HEAPIFY)

- **find the node n, among the root and its children, whose key is minimum w.r.t. $\preceq$**
- if the root's key is minimum, done!
- otherwise, swap n's and root's keys
- repeat on the sub-tree rooted on n



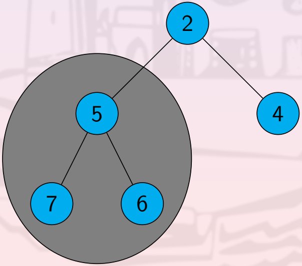
Min-heap



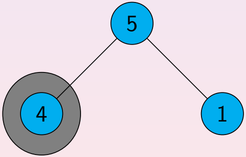
Max-heap

Restoring the Heap Property (HEAPIFY)

- find the node n, among the root and its children, whose key is minimum w.r.t. $\preceq$
- **if the root's key is minimum, done!**
- otherwise, swap n's and root's keys
- repeat on the sub-tree rooted on n



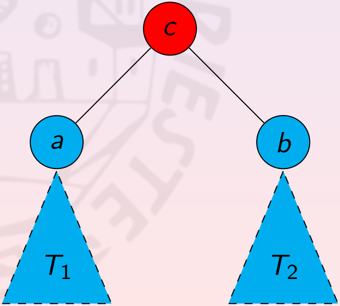
Min-heap



Max-heap

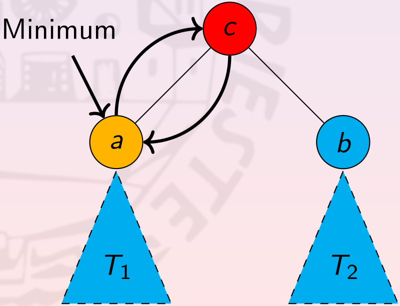
Correctness of HEAPIFY

Before the iteration: the heap property holds in T_1 and T_2



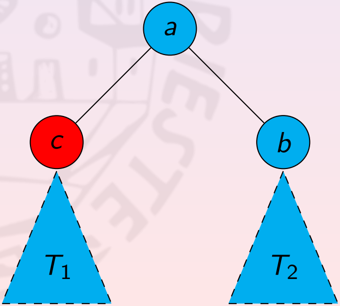
Correctness of HEAPIFY

Before the iteration: the heap property holds in T_1 and T_2



Correctness of HEAPIFY

Before the iteration: the heap property holds in T_1 and T_2

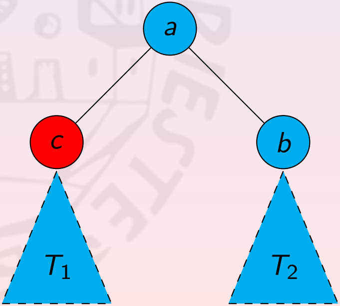


Correctness of HEAPIFY

Before the iteration: the heap property holds in T_1 and T_2

After the iteration:

- the heap property still holds in T_2 and between a and b
- T_1 has been messed up, but it is "shorter" than the original tree and all the keys in T_1 are greater than a



Removing the Minimum: Complexity

Replacing the root's key costs $\Theta(1)$

Removing the Minimum: Complexity

Replacing the root's key costs $\Theta(1)$

For each iteration of `HEAPIFY`:

- 2 comparisons to find the minimum
- 1 swap at most

The distance from a leaf is decreased by one at each iteration

The total cost of `HEAPIFY` is the height of the heap: $O(\log n)$

on a subroot

HEAPIFY: Array-Based Pseudo-Code

```
def HEAPIFY(H, i):
  m ← i

  for j in [LEFT(i), RIGHT(i)]:
     if IS_VALID_NODE(H, j) and H[j] ⪯ H[m]:
       m ← j
     endif
  endfor

  if i != j:
    swap(H, i, j)
    HEAPIFY(H, j)
  endif
enddef
```

i is the element in which heap property does not hold

m will be index of minimum between i and its childern

Removing the Minimum: Array-Based Pseudo-Code

```
def REMOVE_MINIMUM(H, i):
  H[0] ← H[H.size]

  H.size ← H.size −1

  HEAPIFY(H, 0)
enddef
```

Builing a Heap

Building a Binary Heap (BUILD_HEAP)

Building a tree satisfying heap topology is easy

What about heap property?

Building a Binary Heap (BUILD_HEAP)

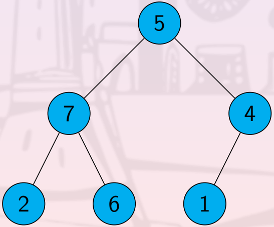
Building a tree satisfying heap topology is easy

What about heap property? Fix it bottom-up by using HEAPIFY

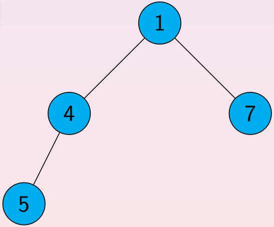
Building a Binary Heap (BUILD_HEAP)

Building a tree satisfying heap topology is easy

What about heap property? Fix it bottom-up by using HEAPIFY



Min-heap

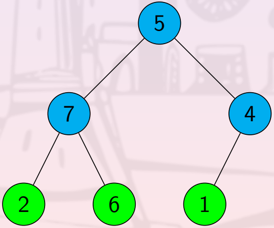


Max-heap

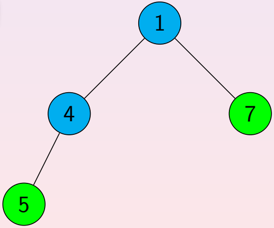
Building a Binary Heap (BUILD_HEAP)

Building a tree satisfying heap topology is easy

What about heap property? Fix it bottom-up by using HEAPIFY



Min-heap



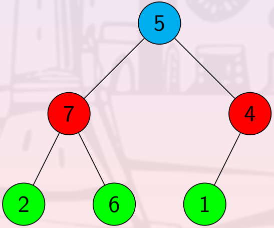
Max-heap

Building a Binary Heap (BUILD_HEAP)

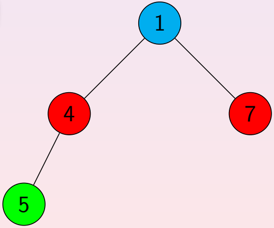
Building a tree satisfying heap topology is easy

What about heap property? Fix it bottom-up by using HEAPIFY

- fix the heaps rooted on the second-last level



Min-heap



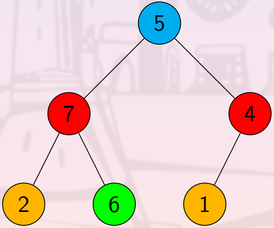
Max-heap

Building a Binary Heap (BUILD_HEAP)

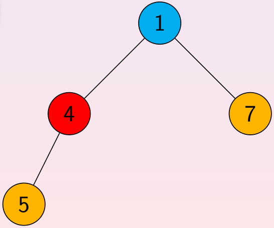
Building a tree satisfying heap topology is easy

What about heap property? Fix it bottom-up by using HEAPIFY

- fix the heaps rooted on the second-last level



Min-heap



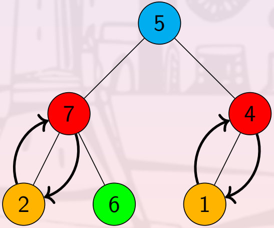
Max-heap

Building a Binary Heap (BUILD_HEAP)

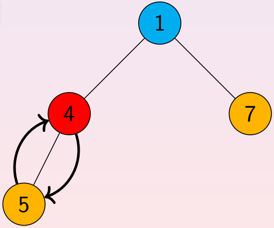
Building a tree satisfying heap topology is easy

What about heap property? Fix it bottom-up by using HEAPIFY

- fix the heaps rooted on the second-last level



Min-heap



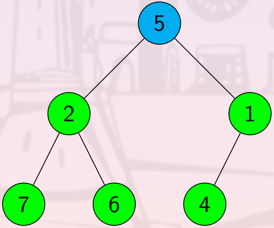
Max-heap

Building a Binary Heap (BUILD_HEAP)

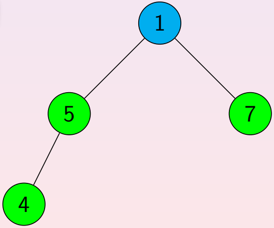
Building a tree satisfying heap topology is easy

What about heap property? Fix it bottom-up by using HEAPIFY

- fix the heaps rooted on the second-last level



Min-heap



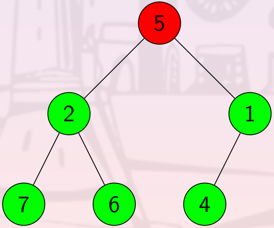
Max-heap

Building a Binary Heap (BUILD_HEAP)

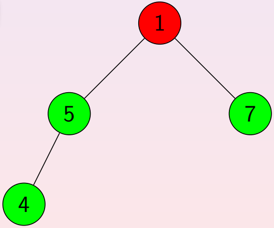
Building a tree satisfying heap topology is easy

What about heap property? Fix it bottom-up by using HEAPIFY

- fix the heaps rooted on the second-last level
- fix the heaps rooted on the third-last level



Min-heap



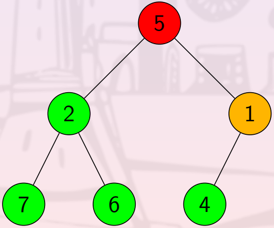
Max-heap

Building a Binary Heap (BUILD_HEAP)

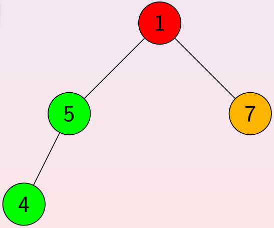
Building a tree satisfying heap topology is easy

What about heap property? Fix it bottom-up by using HEAPIFY

- fix the heaps rooted on the second-last level
- fix the heaps rooted on the third-last level



Min-heap



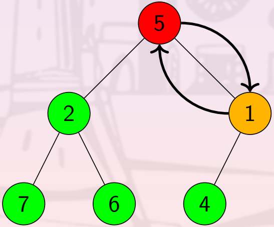
Max-heap

Building a Binary Heap (BUILD_HEAP)

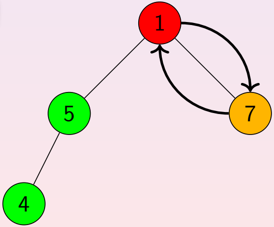
Building a tree satisfying heap topology is easy

What about heap property? Fix it bottom-up by using HEAPIFY

- fix the heaps rooted on the second-last level
- fix the heaps rooted on the third-last level



Min-heap



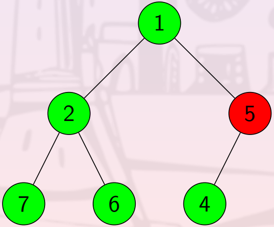
Max-heap

Building a Binary Heap (BUILD_HEAP)

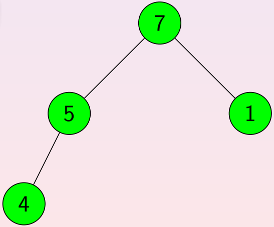
Building a tree satisfying heap topology is easy

What about heap property? Fix it bottom-up by using HEAPIFY

- fix the heaps rooted on the second-last level
- fix the heaps rooted on the third-last level



Min-heap



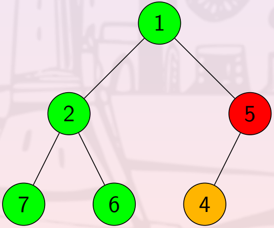
Max-heap

Building a Binary Heap (BUILD_HEAP)

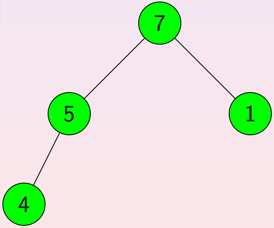
Building a tree satisfying heap topology is easy

What about heap property? Fix it bottom-up by using HEAPIFY

- fix the heaps rooted on the second-last level
- fix the heaps rooted on the third-last level



Min-heap



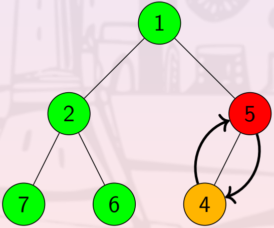
Max-heap

Building a Binary Heap (BUILD_HEAP)

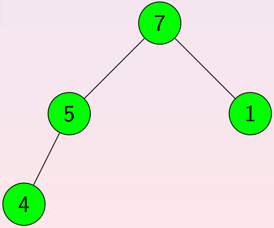
Building a tree satisfying heap topology is easy

What about heap property? Fix it bottom-up by using HEAPIFY

- fix the heaps rooted on the second-last level
- fix the heaps rooted on the third-last level



Min-heap



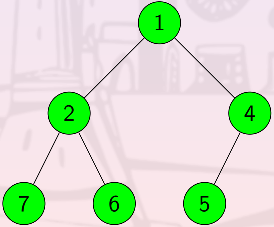
Max-heap

Building a Binary Heap (BUILD_HEAP)

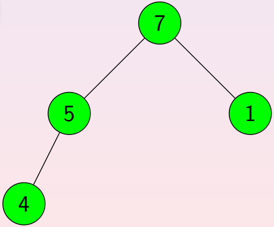
Building a tree satisfying heap topology is easy

What about heap property? Fix it bottom-up by using HEAPIFY

- fix the heaps rooted on the second-last level
- fix the heaps rooted on the third-last level



Min-heap



Max-heap

Building a Binary Heap (BUILD_HEAP)

Building a tree satisfying heap topology is easy

What about heap property? Fix it bottom-up by using HEAPIFY

- fix the heaps rooted on the second-last level
- fix the heaps rooted on the third-last level
- ...

Min-heap

Max-heap

Complexity of BUILD_HEAP

Heapify is O(height) and I call it for every h

HEAPIFY costs $O(h)$ (i.e., $\leq c * h$) on a tree having height h

If the considered tree has n nodes:

- its height is $\lfloor \log_2 n \rfloor$
- it contains at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ at height h

Complexity of BUILD_HEAP

heap has height log_2 n#nodes having height h is l= (n/2^(h+1)).

The costs $T_{\mathrm{bh}}(n)$ of executing BUILD_HEAP on a n-sized tree is:

$$
\begin{aligned}
T_{\mathrm{bh}}(n) &= \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{n}{2^{h+1}} * (c * h) \\
&= c * \frac{n}{2} * \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \\
&\leq c * \frac{n}{2} * \sum_{h=0}^{\infty} \frac{h}{2^h} = c * \frac{n}{2} * \frac{1/2}{(1 - 1/2)^2} \\
&\leq c * n \in O(n)
\end{aligned}
$$

(number of time we apply

BUILD_HEAP: Pseudo-Code

```
def BUILD_HEAP_AUX(H, node):
   if IS_VALID_NODE(H, node):
      BUILD_HEAP_AUX(H, LEFT(node))
      BUILD_HEAP_AUX(H, RIGHT(node))

      HEAPIFY(H, node)
   endif
enddef

def BUILD_HEAP(A):
   H ← BUILD_HEAP_TREE(A)

   BUILD_HEAP_AUX(H, GET_ROOT(H))
enddef
```

BUILD_HEAP: Array-Based Pseudo-Code

The array-based representation helps in avoiding recursion

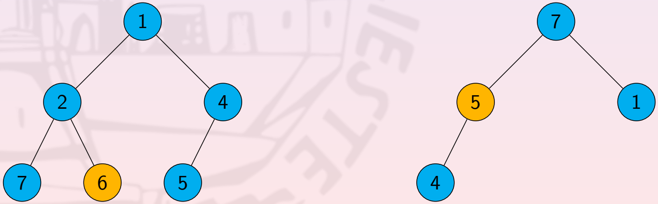
Finding the nodes of the *i*-th level is easy ...

... they are represented by elements in positions $[2^i, 2^{i+1} - 1]$

```
def BUILD_HEAP(A):
  A.size = |A|

   for i ← floor(|A|/2) downto 1:
     HEAPIFY(A, i)
   endfor
enddef
```

Decreasing a Key

Decreasing a Node's Key w.r.t. $\preceq$

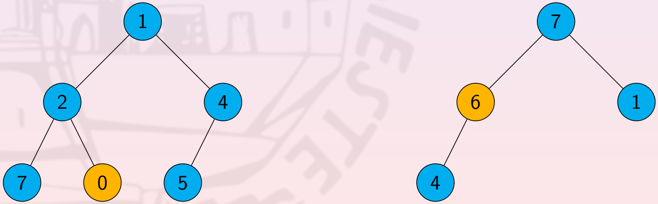


Min-heap

Max-heap

Decreasing a Node's Key w.r.t. $\preceq$

Preserves the heap property on the sub-tree rooted on the node

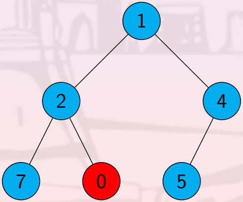


Min-heap

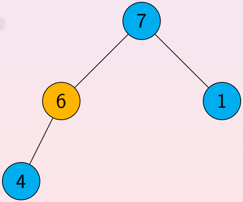
Max-heap

Decreasing a Node's Key w.r.t. $\preceq$

Preserves the heap property on the sub-tree rooted on the node, but it may broke the property w.r.t. its parent



Min-heap

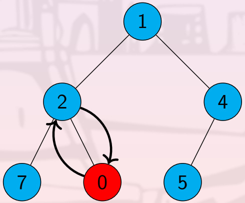


Max-heap

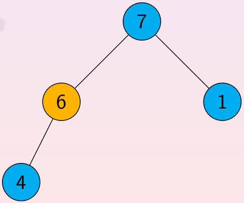
Decreasing a Node's Key w.r.t. $\preceq$

Preserves the heap property on the sub-tree rooted on the node, but it may broke the property w.r.t. its parent

Swapping the keys of the node and its parent solve the problem on the subtree rooted on the parent



Min-heap

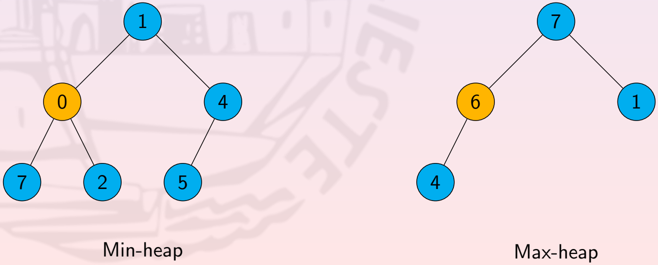


Max-heap

Decreasing a Node's Key w.r.t. $\preceq$

Preserves the heap property on the sub-tree rooted on the node, but it may broke the property w.r.t. its parent

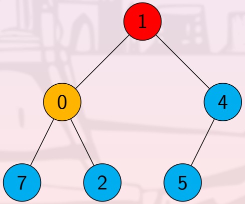
Swapping the keys of the node and its parent solve the problem on the subtree rooted on the parent



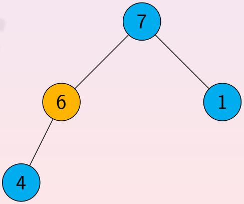
Decreasing a Node's Key w.r.t. $\preceq$

Preserves the heap property on the sub-tree rooted on the node, but it may broke the property w.r.t. its parent

Swapping the keys of the node and its parent solve the problem on the subtree rooted on the parent



Min-heap



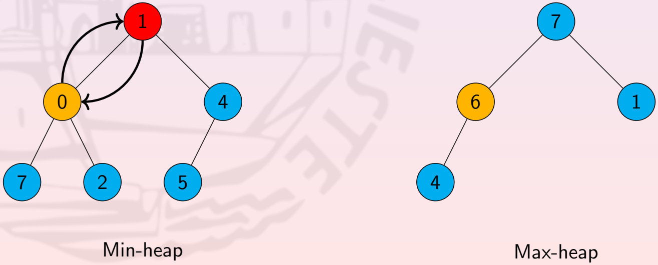
Max-heap

Decreasing a Node's Key w.r.t. $\preceq$

Preserves the heap property on the sub-tree rooted on the node, but it may broke the property w.r.t. its parent

Swapping the keys of the node and its parent solve the problem on the subtree rooted on the parent

Repeat the process until the heap property is restored

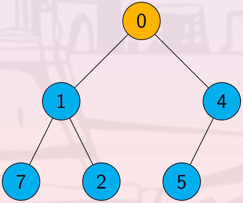


Decreasing a Node's Key w.r.t. $\preceq$

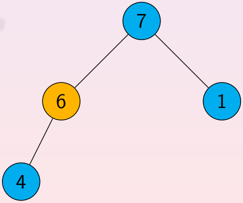
Preserves the heap property on the sub-tree rooted on the node, but it may broke the property w.r.t. its parent

Swapping the keys of the node and its parent solve the problem on the subtree rooted on the parent

Repeat the process until the heap property is restored



Min-heap



Max-heap

Decreasing a Key w.r.t. $\preceq$: Complexity

Each iteration either:

- ends the computation in time $\Theta(1)$ or
- pushes the problem one step closer to the root in time $\Theta(1)$

Since the heap height is $\lfloor \log_2 n \rfloor$, the complexity is $O(\log n)$

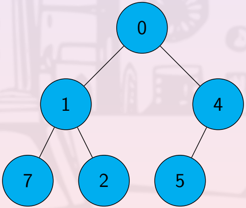
Decreasing a Key w.r.t. $\preceq$: Pseudo-Code

in array based representation, i is index of array. H[i] is actually the key

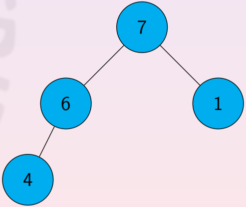
```
def HEAP_DECREASE_KEY(H, i, value):
  if H[i] ⪯ value:
    error(value+"␣is␣not␣smaller␣than␣H["+i+"]")
  endif

  H[i] ← value

  while not(IS_ROOT(i) or H[parent(i)]<= H[i] ):
    swap(H,i,PARENT(i))

    i ← PARENT(i)
  endwhile
enddef
```

Inserting a Value

Inserting a New Value



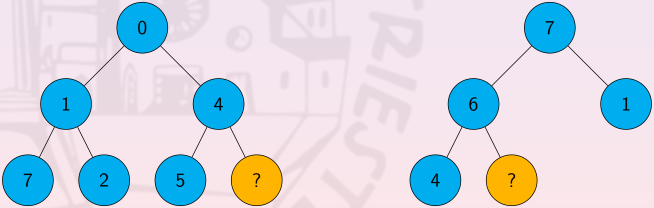
Min-heap



Max-heap

Inserting a New Value

- add a new node N preserving the heap topology

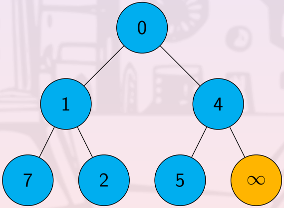


Min-heap

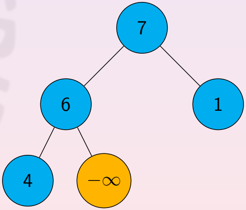
Max-heap

Inserting a New Value

- add a new node N preserving the heap topology
- set the key of N to the maximum value w.r.t. $\preceq$, e.g. ∞ for $\leq$



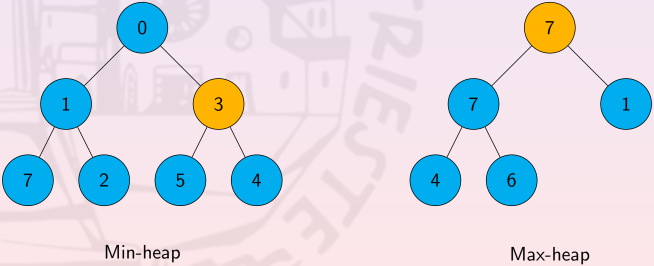
Min-heap



Max-heap

Inserting a New Value

- add a new node N preserving the heap topology
- set the key of N to the maximum value w.r.t. $\preceq$, e.g. ∞ for $\leq$
- decrease the key of N to the desired value



Inserting a New Value: Array-Based Pseudo-Code

```
def HEAP_INSERT(H, value):
  H.size ← H.size+1
  H[H.size] ← ∞⪯

  HEAP_DECREASE_KEY(H, H.size, value)
enddef
```

Has the same complexity of HEAP_DECREASE_KEY: $O(\log n)$

Summarizing complexity

DS	Building	Extracting	Inserting	Decreasing
Binary Heap	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Fibonacci Heap (Amortized)	$\Theta(n)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$