# Lecture #2 Sorting. Heapsort

## Fundamental Algorithms

Rodica Potolea, Camelia Lemnaru and Ciprian Oprișa

Technical University of Cluj-Napoca
Computer Science

October 2024

# Agenda

1. Lecture #1 review
2. Sorting Problem
3. Heapsort

# Agenda

1. Lecture #1 review
2. Sorting Problem
3. Heapsort

# Correctness

- How do we know an algorithm is correct?
- Testing never shows an algorithm is correct. It can only show it is INCORRECT (by finding bugs)
- Absence of evidence != Evidence of absence
- Correctness MUST be proven!
    - if the *pre-conditions* are satisfied, the *post-conditions* will be true when the algorithm *terminates*
    - *partial correctness* = whenever preconditions are satisfied, the postconditions are true
    - *total correctness* = partial correctness + termination condition

# Complexity

- Evaluate **time** and **space** requirements
- **Time** as an estimation of the **amount of work** done
  - As an expression of #*atomic* operations
  - Depends on the *size* of the input data (n)
  - Depends on *case* (best, worst, average to be evaluated)
- **Space** requirements as an expression of **supplementary** memory
  - Need algorithms using *constant extra space*
  - Sometimes, *lgn* extra space is accepted

# Complexity

- Time = amount of work = as a function of $n$ (size of input data)
- We need its asymptotic growth
- Lower bound $\Omega$ depends on the **problem**
- Upper bound $O$ depends on the **algorithm**
- **Efficiency** – compare algorithms (their corresponding $O$ function) among each other – one is more/less efficient
- **Optimality** – $\Omega = O$ in the *worst case* scenario - compare an *algorithm* with the *problem* lower bound

# Stability

- The property of an algorithm to preserve the relative order of equal elements from the input (initial/original data) in the output (final data/result)
- Desired property, when and why?

# Agenda

# Sorting Problem

Assume you want to sort a collection of 3 elements: $a_1, a_2$ and $a_3$

# Sorting Problem

Assume you want to sort a collection of 3 elements: $a_1, a_2$ and $a_3$

- How many comparison *decisions* do you have to make, to cover all possibilities?

# Sorting Problem

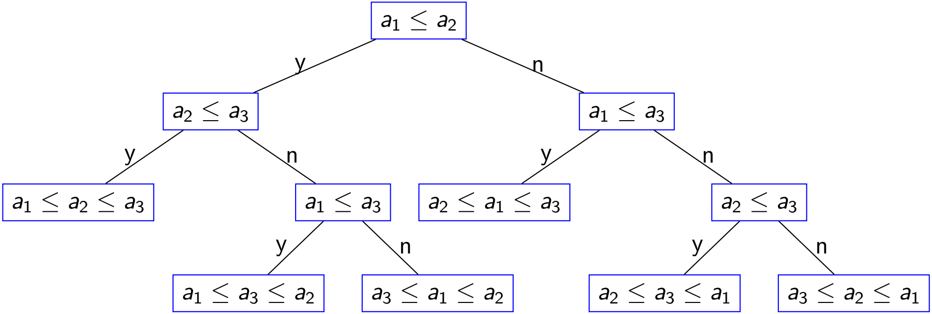
Assume you want to sort a collection of 3 elements: $a_1, a_2$ and $a_3$

- How many comparison *decisions* do you have to make, to cover all possibilities?
  - How many possibilities are there?

# Sorting Problem

Assume you want to sort a collection of 3 elements: $a_1, a_2$ and $a_3$

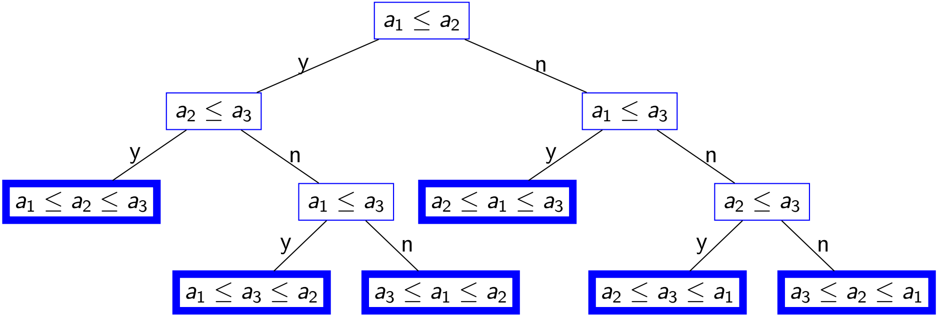
- How many comparison *decisions* do you have to make, to cover all possibilities?
    - How many possibilities are there?



# Sorting Problem Ω

Assume you want to sort a collection of 3 elements: $a_1, a_2$ and $a_3$

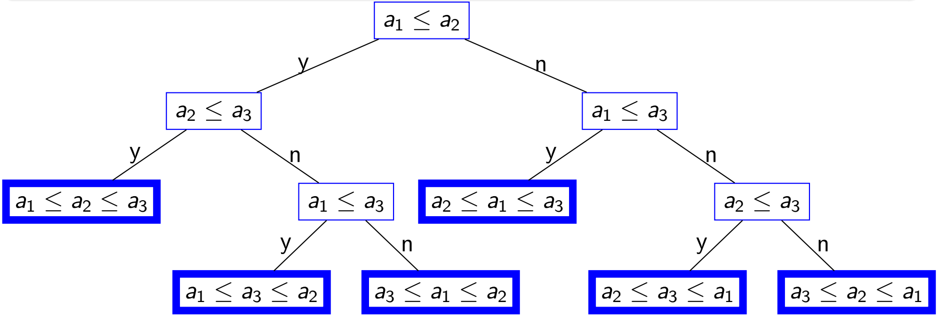
- How many comparison *decisions* do you have to make, to cover any of the potential input sequences?
  - How many possibilities (potential inputs) are there?



# Sorting Problem Ω

## Lemma

*Any comparison-based sorting algorithm performs $\Omega(n \cdot lg(n))$ comparisons in the worst case to sort n objects*




- leaves = each possible answer for any given input
- How many leaves? ($\ell$)

# Sorting Problem $\Omega$

## Lemma

*Any comparison-based sorting algorithm performs $\Omega(n \cdot lg(n))$ comparisons in the worst case to sort $n$ objects*

- $\ell = n!$

# Sorting Problem $\Omega$

**Lemma**

*Any comparison-based sorting algorithm performs $\Omega(n \cdot lg(n))$ comparisons in the worst case to sort $n$ objects*

- $\ell = n!$
- Worst case running time $\cong$ what from the tree?

# Sorting Problem Ω

## Lemma

*Any comparison-based sorting algorithm performs $\Omega(n \cdot lg(n))$ comparisons in the worst case to sort $n$ objects*

- $\ell = n!$
- Worst case running time $\cong$ height of the tree $(h_T)$

# Sorting Problem $\Omega$

**Lemma**

*Any comparison-based sorting algorithm performs $\Omega(n \cdot lg(n))$ comparisons in the worst case to sort n objects*

- $\ell = n!$
- Worst case running time $\cong$ height of the tree ($h_T$)
- $h_T \quad ? \quad \ell$

# Sorting Problem Ω

## Lemma

*Any comparison-based sorting algorithm performs $\Omega(n \cdot lg(n))$ comparisons in the worst case to sort n objects*

- $\ell = n!$
- Worst case running time $\cong$ height of the tree ($h_T$)
- $h_T$ ? $\ell$
  - (hint) What is the maximum #leaves (max $\ell$) for a tree of height $h_T$ ?

# Sorting Problem $\Omega$

## Lemma

*Any comparison-based sorting algorithm performs $\Omega(n \cdot lg(n))$ comparisons in the worst case to sort $n$ objects*

- $\ell = n!$
- Worst case running time $\cong$ height of the tree ($h_T$)
- $h_T \quad ? \quad \ell$
  - $h_T > log_2(\ell)$ (why?)

# Sorting Problem $\Omega$

## Lemma

*Any comparison-based sorting algorithm performs $\Omega(n \cdot lg(n))$ comparisons in the worst case to sort $n$ objects*

- $\ell = n!$
- Worst case running time $\cong$ height of the tree ($h_T$)
- $h_T \quad ? \quad \ell$
  - $h_T > log_2(\ell)$
    $h_T > log_2(n!) = log_2(1 \cdot 2 \cdot 3 \cdot ... \cdot n)$

# Sorting Problem $\Omega$

## Lemma

*Any comparison-based sorting algorithm performs $\Omega(n \cdot lg(n))$ comparisons in the worst case to sort $n$ objects*

- $\ell = n!$
- Worst case running time $\cong$ height of the tree ($h_T$)
- $h_T \quad ? \quad \ell$
  - $h_T > log_2(\ell)$
    $h_T > log_2(n!) = log_2(1 \cdot 2 \cdot 3 \cdot ... \cdot n)$
    $= log_2 1 + log_2 2 + \ldots + log_2 n$

# Sorting Problem Ω

## Lemma

*Any comparison-based sorting algorithm performs $\Omega(n \cdot lg(n))$ comparisons in the worst case to sort n objects*

- $\ell = n!$
- Worst case running time $\cong$ height of the tree ($h_T$)
- $h_T \quad ? \quad \ell$
    - $h_T > log_2(\ell)$

      $h_T > log_2(n!) = log_2(1 \cdot 2 \cdot 3 \cdot ... \cdot n)$

      $= log_2 1 + log_2 2 + \ldots + log_2 n$

      $\geq log_2 \frac{n}{2} + \ldots + log_2 n \qquad //take\ only\ second\ half\ of\ sum$

# Sorting Problem $\Omega$

## Lemma

*Any comparison-based sorting algorithm performs $\Omega(n \cdot lg(n))$ comparisons in the worst case to sort n objects*

- $\ell = n!$
- Worst case running time $\cong$ height of the tree ($h_T$)
- $h_T \quad ? \quad \ell$
  - $h_T > log_2(\ell)$
    $h_T > log_2(n!) = log_2(1 \cdot 2 \cdot 3 \cdot ... \cdot n)$
    $= log_2 1 + log_2 2 + \ldots + log_2 n$
    $\geq log_2 \frac{n}{2} + \ldots + log_2 n \qquad$ *//take only second half of sum*
    $\geq \frac{n}{2} \cdot log_2 \frac{n}{2} \qquad$ *//replace all terms with first*

# Sorting Problem $\Omega$

## Lemma

*Any comparison-based sorting algorithm performs $\Omega(n \cdot lg(n))$ comparisons in the worst case to sort n objects*

- $\ell = n!$
- Worst case running time $\cong$ height of the tree ($h_T$)
- $h_T \quad ? \quad \ell$
  - $h_T > log_2(\ell)$

    $h_T > log_2(n!) = log_2(1 \cdot 2 \cdot 3 \cdot ... \cdot n)$

    $= log_2 1 + log_2 2 + \ldots + log_2 n$

    $\geq log_2 \frac{n}{2} + \ldots + log_2 n \qquad //take\ only\ second\ half\ of\ sum$

    $\geq \frac{n}{2} \cdot log_2 \frac{n}{2} \qquad //replace\ all\ terms\ with\ first$

    $= \Omega(nlgn) \qquad //ignore\ constants$

# Agenda

1. Lecture #1 review
2. Sorting Problem
3. Heapsort

# Heapsort

- Sorting with the aid of a *heap* structure

# Heapsort

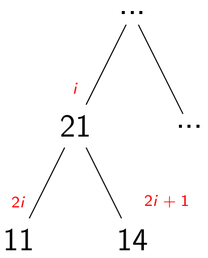
- Sorting with the aid of a *heap* structure
- **Heap** = **Array**, viewed as (*logical perspective*) a *complete* **Binary Tree**; a **partial order relation** is established.

# Heap structure

- **Heap** = **Array**, viewed as (*logical perspective*) a *complete* **Binary Tree**; a **partial order relation** is established.

H =

| | $i$ | | $2i$ | $2i+1$ | |
|---|---|---|---|---|---|
| ... | 21 | ... | 11 | 14 | ... |



# Heap structure

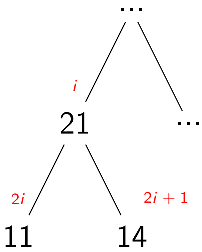
- **Heap** = **Array**, viewed as (*logical perspective*) a *complete* **Binary Tree**; a **partial order relation** is established.

| | | $i$ | | $2i$ | $2i+1$ | |
|---|---|---|---|---|---|---|
| H = | ... | 21 | ... | 11 | 14 | ... |

$H[i] \geq H[2i]$

$H[i] \geq H[2i+1]$

$H[2i] \quad ? \quad H[2i+1]$



# Heap structure

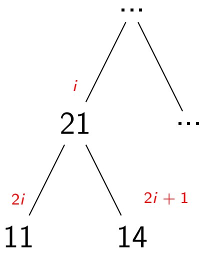
- **Heap** = **Array**, viewed as (*logical perspective*) a *complete* **Binary Tree**; a **partial order relation** is established.

| | | $i$ | | $2i$ | $2i+1$ | |
|---|---|---|---|---|---|---|
| H = | ... | 21 | ... | 11 | 14 | ... |

$H[i] \geq H[2i]$

$H[i] \geq H[2i+1]$

$\cancel{H[2i] \quad ? \quad H[2i+1]}$



# Heap structure

- **Heap** $=$ **Array**, viewed as (*logical perspective*) a *complete* **Binary Tree**; a **partial order relation** is established.

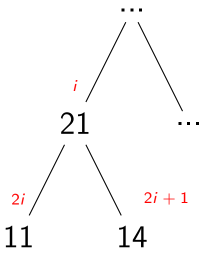
| | | $i$ | | $2i$ | $2i+1$ | |
|---|---|---|---|---|---|---|
| H = | ... | 21 | ... | 11 | 14 | ... |

$H[i] \geq H[2i]$

$H[i] \geq H[2i+1]$

$\cancel{H[2i] \quad ? \quad H[2i+1]}$

other P.O. relations can be defined



# Heap structure – consequences

- Which is a maximal subset (from the tree) on which the *partial order* relation becomes a *total* order relation?

- Considering the heap property, what consequence (*post condition*) follows?

# Heap structure – consequences

- Which is a maximal subset (from the tree) on which the *partial order* relation becomes a *total* order relation?
  - a **branch**
- Considering the heap property, what consequence (*post condition*) follows?

# Heap structure – consequences

- Which is a maximal subset (from the tree) on which the *partial order* relation becomes a *total* order relation?
  - a **branch**
- Considering the heap property, what consequence (*post condition*) follows?
  - the **root** contains the maximum value

# Heap structure – procedures

- *Heapify*

# Heap structure – procedures

- *Heapify*
  - "Adds " the root to 2 left and right children rooted heaps

# Heap structure – procedures

- *Heapify*
  - "Adds " the root to 2 left and right children rooted heaps
- *Build-Heap*

# Heap structure – procedures

- *Heapify*
  - "Adds " the root to 2 left and right children rooted heaps
- *Build-Heap*
  - Constructs the whole heap structure (on an arbitrary array), by repeatedly applying *Heapify*

# Heap structure – procedures

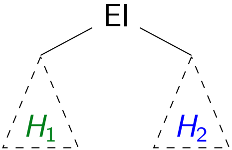
- *Heapify*
  - "Adds " the root to 2 left and right children rooted heaps
- *Build-Heap*
  - Constructs the whole heap structure (on an arbitrary array), by repeatedly applying *Heapify*
- *Heapsort*

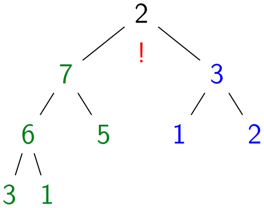
# Heap structure – procedures

- *Heapify*
  - “Adds ” the root to 2 left and right children rooted heaps
- *Build-Heap*
  - Constructs the whole heap structure (on an arbitrary array), by repeatedly applying *Heapify*
- *Heapsort*
  - Sorts by repeatedly extracting the root of the heap and placing it in the appropriate position of the sorted array

# *Heapify*(Reconstituie heap)

- Pre-condition: 2 heaps ($H_1$, $H_2$)
- Goal: add a single element $El$ s.t. the triple ($El$, $H_1$, $H_2$) generates a new, larger heap ($H$)
- Post-condition: 1 single heap $H$, containing all the elements (from) $El$, $H_1$ and $H_2$
- Strategy: *top–down*, **sink the root** to its correct place in the heap



# *Heapify*

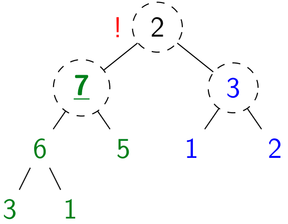
$\textsc{Heapify}(H, i)$

$\quad largest = \textsc{index\_of\_max}(H[i], H[left(i)], H[right(i)])$
$\quad$ **if** $largest \,!= i$ // one child larger than root at least
$\qquad H[i] \leftrightarrow H[largest]$ // swap w. largest child
$\qquad \textsc{Heapify}(H, largest)$ // continue down the heap

# *Heapify*

HEAPIFY$(H, i)$

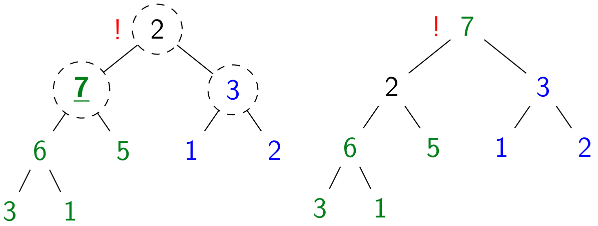
$largest = \text{INDEX\_OF\_MAX}(H[i], H[left(i)], H[right(i)])$
**if** $largest != i$ // one child larger than root at least
  $H[i] \leftrightarrow H[largest]$ // swap w. largest child
  HEAPIFY$(H, largest)$ // continue down the heap



# *Heapify*

HEAPIFY($H, i$)

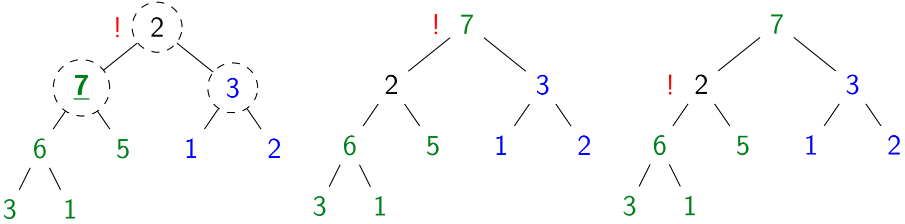
$largest$ = INDEX_OF_MAX($H[i], H[left(i)], H[right(i)]$)
**if** $largest \mathrel{!}= i$ // one child larger than root at least
    $H[i] \leftrightarrow H[largest]$ // swap w. largest child
    HEAPIFY($H, largest$) // continue down the heap



# Heapify

HEAPIFY$(H, i)$

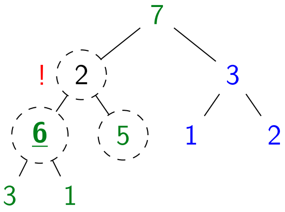
$largest$ = INDEX_OF_MAX$(H[i], H[left(i)], H[right(i)])$
**if** $largest != i$ // one child larger than root at least
    $H[i] \leftrightarrow H[largest]$ // swap w. largest child
    HEAPIFY$(H, largest)$ // continue down the heap



# *Heapify*

HEAPIFY($H, i$)

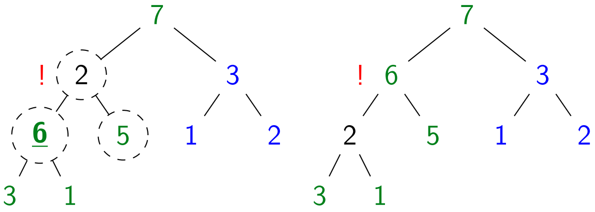
$largest = \text{INDEX\_OF\_MAX}(H[i], H[left(i)], H[right(i)])$
**if** $largest != i$ // one child larger than root at least
  $H[i] \leftrightarrow H[largest]$ // swap w. largest child
  HEAPIFY($H, largest$) // continue down the heap



# *Heapify*

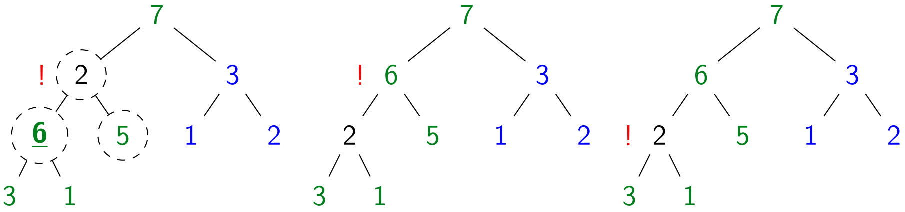
$\textsc{Heapify}(H, i)$

$largest = \textsc{index\_of\_max}(H[i], H[left(i)], H[right(i)])$
**if** $largest != i$ // one child larger than root at least
    $H[i] \leftrightarrow H[largest]$ // swap w. largest child
    $\textsc{Heapify}(H, largest)$ // continue down the heap



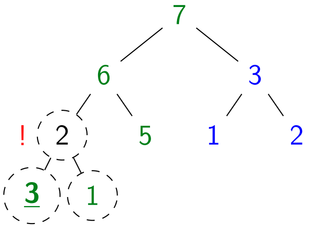
# *Heapify*

```
HEAPIFY(H, i)
    largest = INDEX_OF_MAX(H[i], H[left(i)], H[right(i)])
    if largest != i // one child larger than root at least
        H[i] ↔ H[largest] // swap w. largest child
        HEAPIFY(H, largest) // continue down the heap
```



# *Heapify*

```
HEAPIFY(H, i)
    largest = INDEX_OF_MAX(H[i], H[left(i)], H[right(i)])
    if largest != i // one child larger than root at least
        H[i] ↔ H[largest] // swap w. largest child
        HEAPIFY(H, largest) // continue down the heap
```



# Heapify

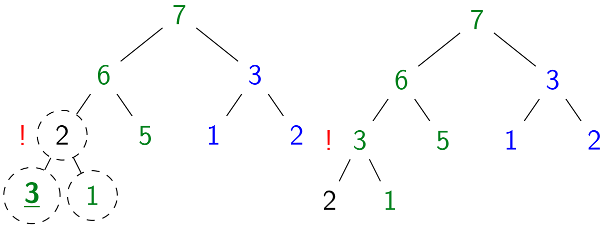
HEAPIFY($H, i$)

$largest = \text{INDEX\_OF\_MAX}(H[i], H[left(i)], H[right(i)])$

**if** $largest != i$ // one child larger than root at least

  $H[i] \leftrightarrow H[largest]$ // swap w. largest child

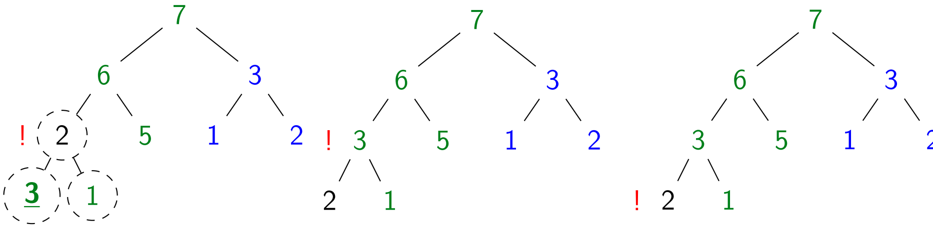
  HEAPIFY($H, largest$) // continue down the heap



# *Heapify*

HEAPIFY($H, i$)

$largest$ = INDEX_OF_MAX($H[i], H[left(i)], H[right(i)]$)
**if** $largest != i$ // one child larger than root at least
    $H[i] \leftrightarrow H[largest]$ // swap w. largest child
    HEAPIFY($H, largest$) // continue down the heap



# *Heapify* - running time

$\textsc{Heapify}(H, i)$

```
largest = INDEX_OF_MAX(H[i], H[left(i)], H[right(i)])
if largest != i // one child larger than root at least
      H[i] ↔ H[largest] // swap w. largest child
      HEAPIFY(H, largest) // continue down the heap
```

- $O(1)$ running time at one level (per rec. calls)

# *Heapify* - running time

```
HEAPIFY(H, i)
largest = INDEX_OF_MAX(H[i], H[left(i)], H[right(i)])
if largest != i // one child larger than root at least
        H[i] ↔ H[largest] // swap w. largest child
        HEAPIFY(H, largest) // continue down the heap
```

- $O(1)$ running time at one level (per rec. calls)
- How many recursive calls? (levels)

# *Heapify* - running time

$\textsc{Heapify}(H, i)$

$largest = \text{INDEX\_OF\_MAX}(H[i], H[left(i)], H[right(i)])$
**if** $largest != i$ // one child larger than root at least
    $H[i] \leftrightarrow H[largest]$ // swap w. largest child
    $\textsc{Heapify}(H, largest)$ // continue down the heap

- $O(1)$ running time at one level (per rec. calls)
- How many recursive calls? (levels)
  - Best: none

# *Heapify* - running time

$\textsc{Heapify}(H, i)$

$largest = \textsc{index\_of\_max}(H[i], H[left(i)], H[right(i)])$
**if** $largest != i$ // one child larger than root at least
    $H[i] \leftrightarrow H[largest]$ // swap w. largest child
    $\textsc{Heapify}(H, largest)$ // continue down the heap

- $O(1)$ running time at one level (per rec. calls)
- How many recursive calls? (levels)
  - Best: none
  - Worst: repeated down to the level of the leaves

# *Heapify* - running time

$\textsc{Heapify}(H, i)$

```
largest = INDEX_OF_MAX(H[i], H[left(i)], H[right(i)])
if largest != i // one child larger than root at least
        H[i] ↔ H[largest] // swap w. largest child
        HEAPIFY(H, largest) // continue down the heap
```

- $O(1)$ running time at one level (per rec. calls)
- How many recursive calls? (levels)
  - Best: none
  - Worst: repeated down to the level of the leaves
    - *Intuitive:* height of a complete BT $\cong lgn$

# *Heapify* - running time

$\textsc{Heapify}(H, i)$

```
largest = INDEX_OF_MAX(H[i], H[left(i)], H[right(i)])
if largest != i // one child larger than root at least
      H[i] ↔ H[largest] // swap w. largest child
      HEAPIFY(H, largest) // continue down the heap
```

- $O(1)$ running time at one level (per rec. calls)
- How many recursive calls? (levels)
  - Best: none
  - Worst: repeated down to the level of the leaves
    - *Intuitive:* height of a complete BT $\cong lgn$
    - *Exact:* The last row of the tree is *exactly half full*, and we go on that branch (Why?)

# *Heapify* - running time

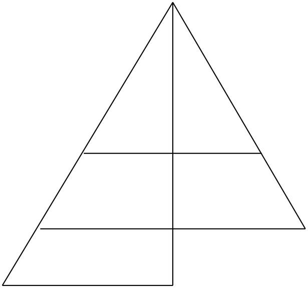
- Worst: The last row of the tree is *exactly half full*, and we go on that branch $->T(n)=?$

# *Heapify* - running time

- Worst: The last row of the tree is *exactly half full*, and we go on that branch $- > T(n) = ?$
  - In complete BT, half of the number of nodes are leaves

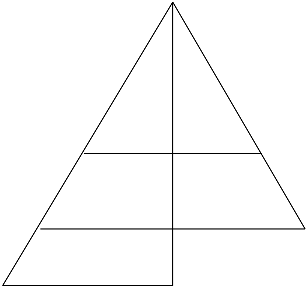
# *Heapify* - running time

- Worst: The last row of the tree is *exactly half full*, and we go on that branch $->$ $T(n) =?$
  - In complete BT, half of the number of nodes are leaves



# *Heapify* - running time

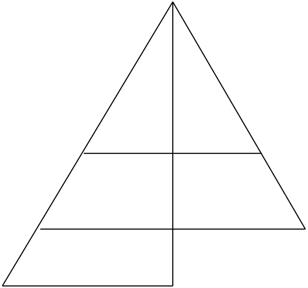
- Worst: The last row of the tree is *exactly half full*, and we go on that branch $->T(n)=?$
  - In complete BT, half of the number of nodes are leaves



level $h$

# *Heapify* - running time

- Worst: The last row of the tree is *exactly half full*, and we go on that branch $->$ $T(n) =?$
  - In complete BT, half of the number of nodes are leaves

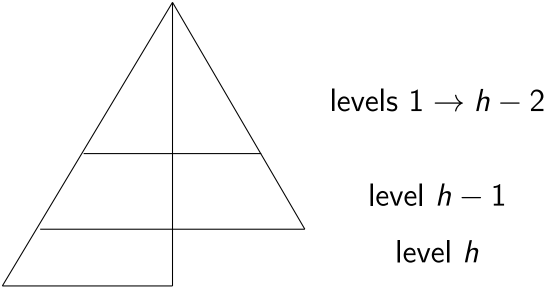


level $h-1$

level $h$

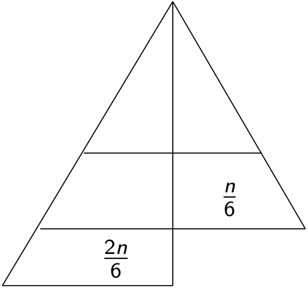
# *Heapify* - running time

- Worst: The last row of the tree is *exactly half full*, and we go on that branch $->T(n)=?$
  - In complete BT, half of the number of nodes are leaves



# *Heapify* - running time

- Worst: The last row of the tree is *exactly half full*, and we go on that branch $->$ $T(n) = ?$
  - In complete BT, half of the number of nodes are leaves




levels $1 \rightarrow h-2$

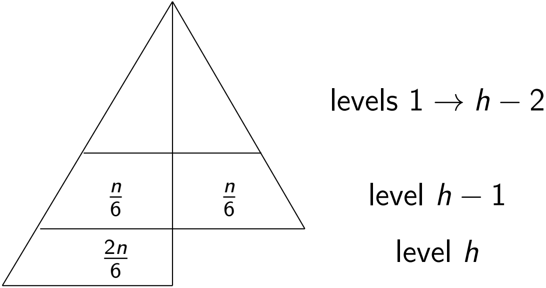
level $h-1$

level $h$

- last level and right half of previous level are leaves

# *Heapify* - running time

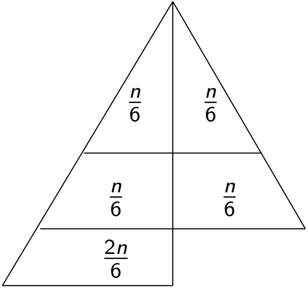
- Worst: The last row of the tree is *exactly half full*, and we go on that branch $->$ $T(n) = ?$
  - In complete BT, half of the number of nodes are leaves




- last level and right half of previous level are leaves
- left half and right half of level $h-1$ contain same #nodes

# *Heapify* - running time

- Worst: The last row of the tree is *exactly half full*, and we go on that branch $->$ $T(n) = ?$
  - In complete BT, half of the number of nodes are leaves



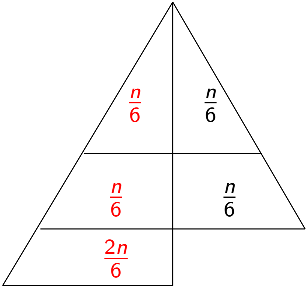

levels $1 \rightarrow h-2$

level $h-1$

level $h$

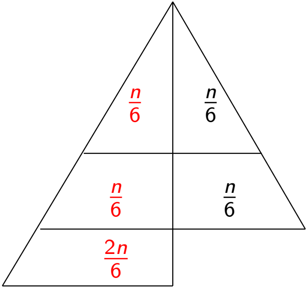
- last level and right half of previous level are leaves
- left half and right half of level $h-1$ contain same #nodes
- the rest of the levels contain, in total, as many nodes as level $h-1$

# *Heapify* - running time



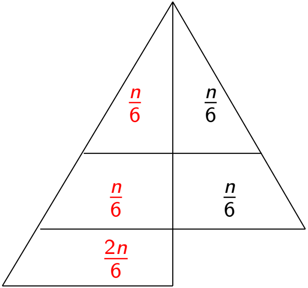

- $T(n) = T(\frac{2n}{3}) + O(1)$

# *Heapify* - running time



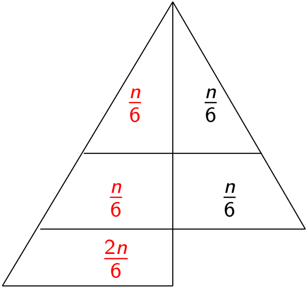

- $T(n) = T(\frac{2n}{3}) + O(1)$

# *Heapify* - running time



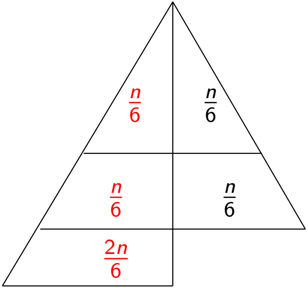

- $T(n) = T(\frac{2n}{3}) + O(1)$
  - $a = 1$

# *Heapify* - running time



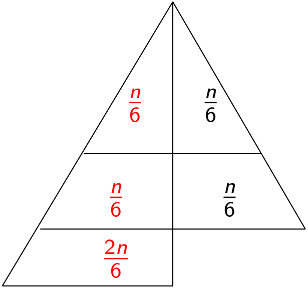

- $T(n) = T(\frac{2n}{3}) + O(1)$
  - $a = 1$
  - $b = \frac{3}{2}$

# *Heapify* - running time




- $T(n) = T(\frac{2n}{3}) + O(1)$
  - $a = 1$
  - $b = \frac{3}{2}$
  - $c = 0$

# *Heapify* - running time




- $T(n) = T(\frac{2n}{3}) + O(1)$
  - $a = 1$
  - $b = \frac{3}{2}$
  - $c = 0$
- (by Master Th., case #2): $T(n) = O(log_{\frac{3}{2}} n) = O(lgn)$

# *Build-Heap*

- *Heapify* starts from the assumption we already have 2 heaps. Where are they from?

# *Build-Heap*

- *Heapify* starts from the assumption we already have 2 heaps. Where are they from?
  - a single node is a (very basic) heap

# *Build-Heap*

- *Heapify* starts from the assumption we already have 2 heaps. Where are they from?
  - a single node is a (very basic) heap
  - all leaves are independent heaps

# *Build-Heap*

- *Heapify* starts from the assumption we already have 2 heaps. Where are they from?
  - a single node is a (very basic) heap
  - all leaves are independent heaps
  - by adding a common parent to 2 leaves, and applying *Heapify*, we obtain a new, larger, heap

# *Build-Heap*

- *Heapify* starts from the assumption we already have 2 heaps. Where are they from?
  - a single node is a (very basic) heap
  - all leaves are independent heaps
  - by adding a common parent to 2 leaves, and applying *Heapify*, we obtain a new, larger, heap
- Adopt a <u>*bottom-up*</u> strategy:

# *Build-Heap*

- *Heapify* starts from the assumption we already have 2 heaps. Where are they from?
  - a single node is a (very basic) heap
  - all leaves are independent heaps
  - by adding a common parent to 2 leaves, and applying *Heapify*, we obtain a new, larger, heap
- Adopt a *bottom-up* strategy:
  - $\frac{1}{2}$ out of all nodes are already heaps (leaves in a complete BT)

# *Build-Heap*

- *Heapify* starts from the assumption we already have 2 heaps. Where are they from?
  - a single node is a (very basic) heap
  - all leaves are independent heaps
  - by adding a common parent to 2 leaves, and applying *Heapify*, we obtain a new, larger, heap
- Adopt a <u>*bottom-up*</u> strategy:
  - $\frac{1}{2}$ out of all nodes are already heaps (leaves in a complete BT)
  - apply *Heapify* to the first non-leaf node (largest index node having at least one child)

# *Build-Heap*

- *Heapify* starts from the assumption we already have 2 heaps. Where are they from?
  - a single node is a (very basic) heap
  - all leaves are independent heaps
  - by adding a common parent to 2 leaves, and applying *Heapify*, we obtain a new, larger, heap
- Adopt a *bottom-up* strategy:
  - $\frac{1}{2}$ out of all nodes are already heaps (leaves in a complete BT)
  - apply *Heapify* to the first non-leaf node (largest index node having at least one child)
  - go to the sibling on the left (of the previously processed element), and to the same, until all elements have been processed (the "last" processed will be the root)

# *Build-Heap*

```
BUILD-HEAP(H)
  heap_size[H] = |H|
  for i = |H|/2 downto 1 // from the first non-leaf, up to the root
        Heapify(H, i) // put node at index i as root to 2 heaps
```

- **bottom-up** strategy
- Running time:

# *Build-Heap*

BUILD-HEAP($H$)

```
heap_size[H] = |H|
for i = |H|/2 downto 1 // from the first non-leaf, up to the root
        Heapify(H, i) // put node at index i as root to 2 heaps
```

- **bottom-up** strategy
- Running time:
  - Worst case, intuitive: $T(n) = \frac{n}{2} lgn$

# *Build-Heap*

```
BUILD-HEAP(H)
heap_size[H] = |H|
for i = |H|/2 downto 1 // from the first non-leaf, up to the root
        Heapify(H, i) // put node at index i as root to 2 heaps
```

- **bottom-up** strategy
- Running time:
  - Worst case, intuitive: $T(n) = \frac{n}{2} lgn$
  - ... because we apply $\frac{n}{2}$ times *Heapify*

# *Build-Heap*

```
BUILD-HEAP(H)
    heap_size[H] = |H|
    for i = |H|/2 downto 1 // from the first non-leaf, up to the root
            Heapify(H, i) // put node at index i as root to 2 heaps
```

- **bottom-up** strategy
- Running time:
  - Worst case, intuitive: $T(n) = \frac{n}{2} lgn$
  - ... because we apply $\frac{n}{2}$ times *Heapify*
  - ... which takes $O(lgn)$

# *Build-Heap*

BUILD-HEAP($H$)

1 $heap\_size[H] = |H|$
2 **for** $i = \frac{|H|}{2}$ **downto** 1 // from the first non-leaf, up to the root
3 $\quad$ $Heapify(H, i)$ // put node at index $i$ as root to 2 heaps

- **bottom-up** strategy
- Running time:
  - Worst case, intuitive: $T(n) = \frac{n}{2} lgn$
  - ... because we apply $\frac{n}{2}$ times *Heapify*
  - ... which takes $O(lgn)$
  - This is **bad**! Building the heap already takes $O(nlgn)$!?

# *Build-Heap* - running time

$\text{BUILD-HEAP}(H)$

```
heap_size[H] = |H|
for i = |H|/2 downto 1 // from the first non-leaf, up to the root
        HEAPIFY(H, i) // put node at index i as root to 2 heaps
```

- Running time - exact:

# *Build-Heap* - running time

BUILD-HEAP($H$)

$heap\_size[H] = |H|$
**for** $i = \frac{|H|}{2}$ **downto** 1 // from the first non-leaf, up to the root
    HEAPIFY($H, i$) // put node at index $i$ as root to 2 heaps

- Running time - exact:
  - for leaf parents, it only takes at most 1 step to *Heapify* each

# *Build-Heap* - running time

BUILD-HEAP($H$)

$heap\_size[H] = |H|$
**for** $i = \frac{|H|}{2}$ **downto** 1 // from the first non-leaf, up to the root
    HEAPIFY($H, i$) // put node at index $i$ as root to 2 heaps

- Running time - exact:
  - for leaf parents, it only takes at most 1 step to *Heapify* each
    - How many such nodes are there?

# *Build-Heap* - running time

BUILD-HEAP($H$)

```
heap_size[H] = |H|
for i = |H|/2 downto 1 // from the first non-leaf, up to the root
        HEAPIFY(H, i) // put node at index i as root to 2 heaps
```

- Running time - exact:
  - for leaf parents, it only takes at most 1 step to *Heapify* each
    - How many such nodes are there?
    - So, the total effort for all leaf parents is: $\frac{1}{2^2} \cdot n \cdot 1$

# *Build-Heap* - running time

$\textsc{Build-Heap}(H)$

$heap\_size[H] = |H|$
**for** $i = \frac{|H|}{2}$ **downto** 1 // from the first non-leaf, up to the root
    $\textsc{Heapify}(H, i)$ // put node at index $i$ as root to 2 heaps

- Running time - exact:
  - for leaf parents, it only takes at most 1 step to *Heapify* each
    - How many such nodes are there?
    - So, the total effort for all leaf parents is: $\frac{1}{2^2} \cdot n \cdot 1$
  - for half of the remaining elements, it takes 2 steps *Heapify* each

# *Build-Heap* - running time

BUILD-HEAP($H$)

$heap\_size[H] = |H|$
**for** $i = \frac{|H|}{2}$ **downto** 1 // from the first non-leaf, up to the root
    HEAPIFY($H, i$) // put node at index $i$ as root to 2 heaps

- Running time - exact:
  - for leaf parents, it only takes at most 1 step to *Heapify* each
    - How many such nodes are there?
    - So, the total effort for all leaf parents is: $\frac{1}{2^2} \cdot n \cdot 1$
  - for half of the remaining elements, it takes 2 steps *Heapify* each
    - So, the total effort for them is: $\frac{1}{2^3} \cdot n \cdot 2$

# *Build-Heap* - running time

BUILD-HEAP($H$)

```
heap_size[H] = |H|
for i = |H|/2 downto 1 // from the first non-leaf, up to the root
      HEAPIFY(H, i) // put node at index i as root to 2 heaps
```

- Running time - exact:
  - for leaf parents, it only takes at most 1 step to *Heapify* each
    - How many such nodes are there?
    - So, the total effort for all leaf parents is: $\frac{1}{2^2} \cdot n \cdot 1$
  - for half of the remaining elements, it takes 2 steps *Heapify* each
    - So, the total effort for them is: $\frac{1}{2^3} \cdot n \cdot 2$
  - for each subsequent step, the #elements halves, while the #steps required to *Heapify* each increases by 1

# *Build-Heap* - running time

BUILD-HEAP($H$)

1 $heap\_size[H] = |H|$

2 **for** $i = \frac{|H|}{2}$ **downto** 1 // from the first non-leaf, up to the root

3 HEAPIFY($H, i$) // put node at index $i$ as root to 2 heaps

- Running time - exact:

$$T(n) =$$

$$\frac{n}{2} \cdot 0+ \quad // \text{ leaves}$$
$$\frac{n}{2^2} \cdot 1+ \quad // \text{ leaf parents}$$
$$\frac{n}{2^3} \cdot 2+$$
$$\frac{n}{2^4} \cdot 3 + \ldots$$

$$= \sum_0^{[lgn]} [\frac{n}{2^{h+1}}] \cdot O(h)$$

# *Build-Heap* - running time

$$T(n) = \sum_{0}^{[lgn]} \left[\frac{n}{2^{h+1}}\right] \cdot O(h) = \frac{n}{2} \cdot \sum_{0}^{[lgn]} \left[\frac{1}{2^{h}}\right] \cdot h \ (**)$$

# *Build-Heap* - running time

$$T(n) = \sum_0^{[lgn]} [\frac{n}{2^{h+1}}] \cdot O(h) = \frac{n}{2} \cdot \sum_0^{[lgn]} [\frac{1}{2^h}] \cdot h \text{ } (**)$$

We must compute $\sum x^k \cdot k$

# *Build-Heap* - running time

$T(n) = \sum_0^{[lgn]} [\frac{n}{2^{h+1}}] \cdot O(h) = \frac{n}{2} \cdot \sum_0^{[lgn]} [\frac{1}{2^h}] \cdot h$ (**)

We must compute $\sum x^k \cdot k$

We know that: $\sum x^k = \frac{(1-x^{n+1})}{(1-x)}$

# *Build-Heap* - running time

$T(n) = \sum_0^{[lgn]} [\frac{n}{2^{h+1}}] \cdot O(h) = \frac{n}{2} \cdot \sum_0^{[lgn]} [\frac{1}{2^h}] \cdot h$ (**)

We must compute $\sum x^k \cdot k$

We know that: $\sum x^k = \frac{(1-x^{n+1})}{(1-x)}$

$\sum x^k = \frac{1}{(1-x)}$ (if $x < 1, n \to \infty$)

# *Build-Heap* - running time

$$T(n) = \sum_0^{[lgn]} [\frac{n}{2^{h+1}}] \cdot O(h) = \frac{n}{2} \cdot \sum_0^{[lgn]} [\frac{1}{2^h}] \cdot h \ (**)$$

We must compute $\sum x^k \cdot k$

We know that: $\sum x^k = \frac{(1-x^{n+1})}{(1-x)}$

$\sum x^k = \frac{1}{(1-x)}$ (if $x < 1, n \to \infty$)

$\sum k \cdot x^{k-1} = \frac{1}{(1-x)^2}$ (derive)

# *Build-Heap* - running time

$T(n) = \sum_0^{[lgn]} [\frac{n}{2^{h+1}}] \cdot O(h) = \frac{n}{2} \cdot \sum_0^{[lgn]} [\frac{1}{2^h}] \cdot h$ (**)

We must compute $\sum x^k \cdot k$

We know that: $\sum x^k = \frac{(1-x^{n+1})}{(1-x)}$

$\sum x^k = \frac{1}{(1-x)}$ (if $x < 1, n \to \infty$)

$\sum k \cdot x^{k-1} = \frac{1}{(1-x)^2}$ (derive)

$\sum k \cdot x^k = \frac{x}{(1-x)^2}$ (multiply by x) (*)

# *Build-Heap* - running time

$T(n) = \sum_{0}^{[lgn]} [\frac{n}{2^{h+1}}] \cdot O(h) = \frac{n}{2} \cdot \sum_{0}^{[lgn]} [\frac{1}{2^h}] \cdot h$ (**)

We must compute $\sum x^k \cdot k$

We know that: $\sum x^k = \frac{(1-x^{n+1})}{(1-x)}$

$\sum x^k = \frac{1}{(1-x)}$ (if $x < 1, n \to \infty$)

$\sum k \cdot x^{k-1} = \frac{1}{(1-x)^2}$ (derive)

$\sum k \cdot x^k = \frac{x}{(1-x)^2}$ (multiply by x) (*)

- make $x = \frac{1}{2}$ in (*), we get:

# *Build-Heap* - running time

$$T(n) = \sum_0^{[lgn]} [\frac{n}{2^{h+1}}] \cdot O(h) = \frac{n}{2} \cdot \sum_0^{[lgn]} [\frac{1}{2^h}] \cdot h \ (**)$$

We must compute $\sum x^k \cdot k$

We know that: $\sum x^k = \frac{(1-x^{n+1})}{(1-x)}$

$\sum x^k = \frac{1}{(1-x)}$ (if $x < 1, n \to \infty$)

$\sum k \cdot x^{k-1} = \frac{1}{(1-x)^2}$ (derive)

$\sum k \cdot x^k = \frac{x}{(1-x)^2}$ (multiply by x) (*)

- make $x = \frac{1}{2}$ in (*), we get:
- $\sum_0^{[lgn]} [\frac{1}{2^h}] \cdot h = 2$

# *Build-Heap* - running time

$T(n) = \sum_0^{[lgn]} [\frac{n}{2^{h+1}}] \cdot O(h) = \frac{n}{2} \cdot \sum_0^{[lgn]} [\frac{1}{2^h}] \cdot h$ (**)

We must compute $\sum x^k \cdot k$

We know that: $\sum x^k = \frac{(1-x^{n+1})}{(1-x)}$

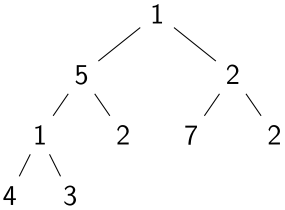
$\sum x^k = \frac{1}{(1-x)}$ (if $x < 1, n \to \infty$)

$\sum k \cdot x^{k-1} = \frac{1}{(1-x)^2}$ (derive)

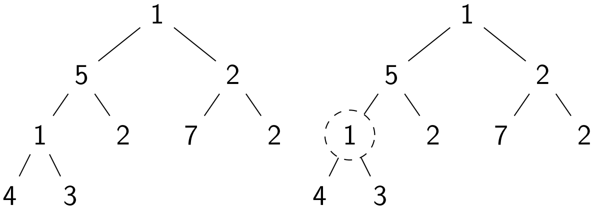
$\sum k \cdot x^k = \frac{x}{(1-x)^2}$ (multiply by x) (*)

- make $x = \frac{1}{2}$ in (*), we get:
- $\sum_0^{[lgn]} [\frac{1}{2^h}] \cdot h = 2$
- Replacing now in (**), we get: $T(n) = \frac{n}{2} \cdot 2 = O(n)$

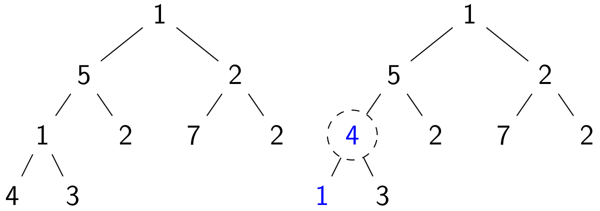
# *Build-Heap* - example



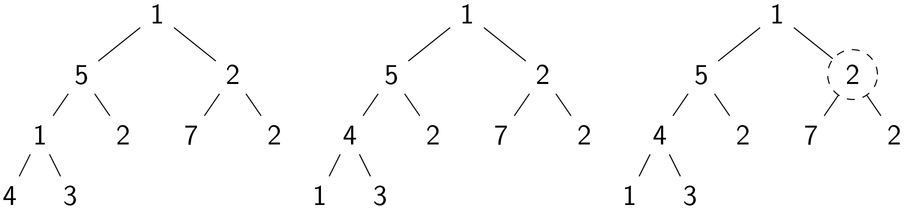
# *Build-Heap* - example



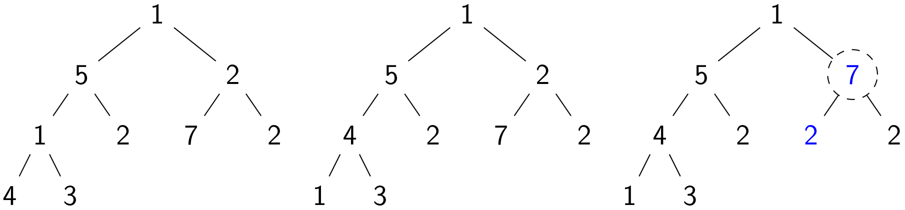
# *Build-Heap* - example



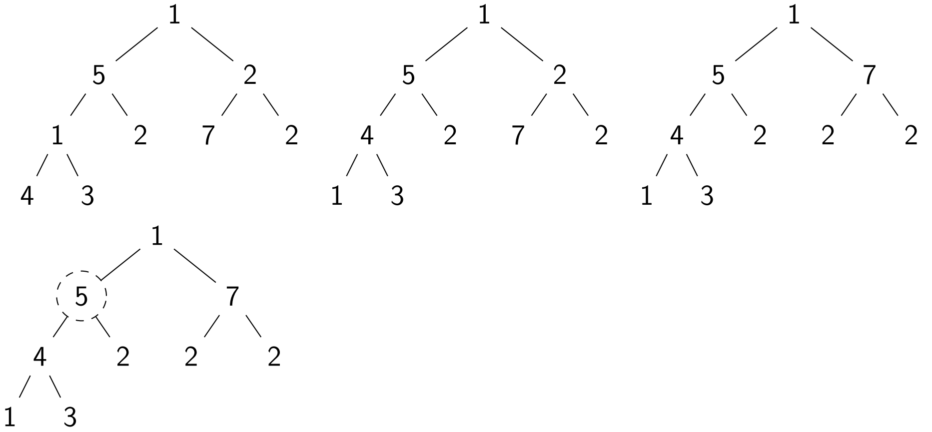
# *Build-Heap* - example



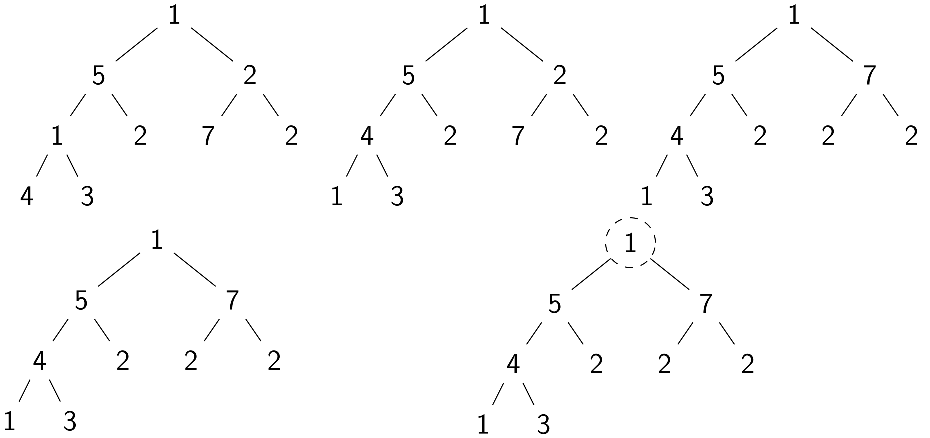
# *Build-Heap* - example



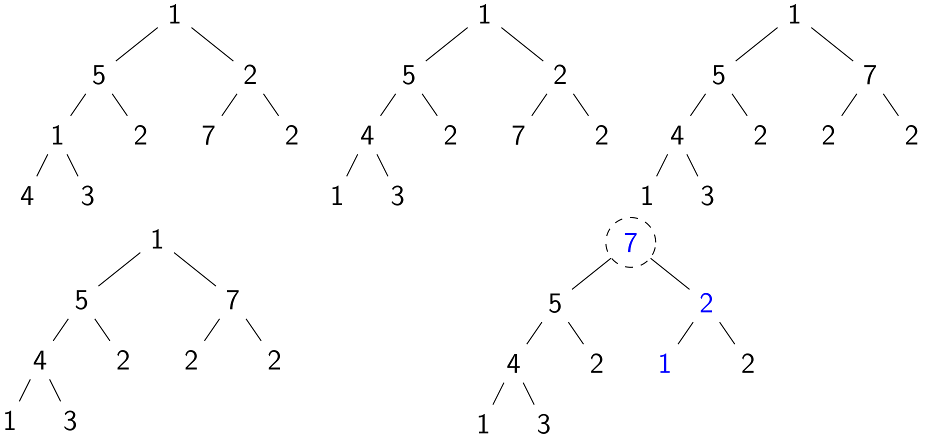
# *Build-Heap* - example



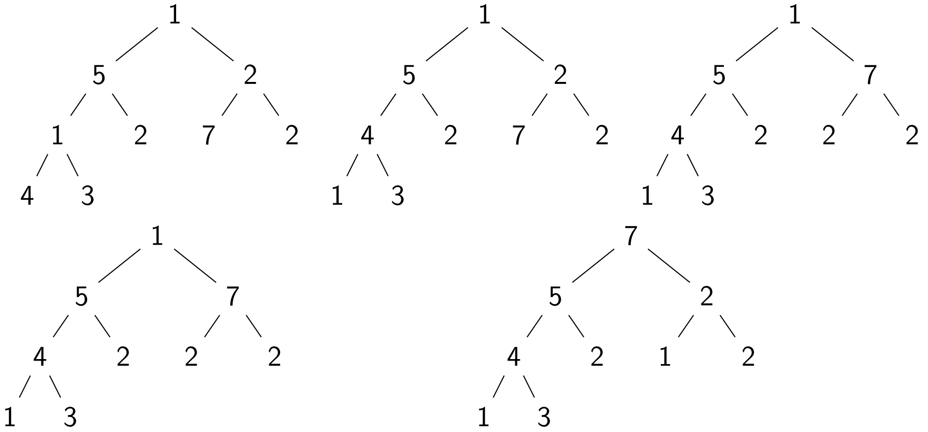
# *Build-Heap* - example



# *Build-Heap* - example



# *Build-Heap* - example



# *Heapsort*

1 *Build-Heap* initially to construct a heap, have the max at the top of the heap

# *Heapsort*

1 *Build-Heap* initially to construct a heap, have the max at the top of the heap
2 *swap the top element (root) with the bottom one (last leaf)*

# *Heapsort*

1. *Build-Heap* initially to construct a heap, have the max at the top of the heap
2. *swap the top element (root) with the bottom one (last leaf)*
    - move the max element in the last position of the array, where it belongs in the ordered array

# *Heapsort*

1. *Build-Heap* initially to construct a heap, have the max at the top of the heap
2. *swap the top element (root) with the bottom one (last leaf)*
   - move the max element in the last position of the array, where it belongs in the ordered array
3. except for the **first** and **last** elements in the array, we have a heap
   - but **last** in the right position in the sorted array, so ...

# *Heapsort*

1. *Build-Heap* initially to construct a heap, have the max at the top of the heap
2. *swap the top element (root) with the bottom one (last leaf)*
    - move the max element in the last position of the array, where it belongs in the ordered array
3. except for the **first** and **last** elements in the array, we have a heap
    - but **last** in the right position in the sorted array, so ...
    - *decrease heap size by 1*

# *Heapsort*

1. *Build-Heap* initially to construct a heap, have the max at the top of the heap
2. *swap the top element (root) with the bottom one (last leaf)*
   - move the max element in the last position of the array, where it belongs in the ordered array
3. except for the **first** and **last** elements in the array, we have a heap
   - but **last** in the right position in the sorted array, so ...
   - *decrease heap size by 1*
   - *apply Heapify on the* **first** *element, in the new, smaller heap* ...

# *Heapsort*

1. *Build-Heap* initially to construct a heap, have the max at the top of the heap
2. *swap the top element (root) with the bottom one (last leaf)*
   - move the max element in the last position of the array, where it belongs in the ordered array
3. except for the **first** and **last** elements in the array, we have a heap
   - but **last** in the right position in the sorted array, so ...
   - *decrease heap size by 1*
   - *apply Heapify on the* **first** *element, in the new, smaller heap* ...
   - to obtain a heap of size reduced by 1 (the last element in the array is no longer part of the heap)

# *Heapsort*

1 *Build-Heap* initially to construct a heap, have the max at the top of the heap
2 *swap the top element (root) with the bottom one (last leaf)*
   - move the max element in the last position of the array, where it belongs in the ordered array
3 except for the **first** and **last** elements in the array, we have a heap
   - but **last** in the right position in the sorted array, so ...
   - *decrease heap size by 1*
   - *apply Heapify on the* **first** *element, in the new, smaller heap* ...
   - to obtain a heap of size reduced by 1 (the last element in the array is no longer part of the heap)
4 repeat steps 2-3 until the heap size becomes 1

# *Heapsort*

HEAPSORT($H$)

```
BUILD-HEAP(H)
for i = |H| downto 2 // do for all array positions from last to sec.
        H[1] ↔ H[i]// swap root with last element in current heap
        heap_size[H] = heap_size[H] − 1 // decrease heap size
        HEAPIFY(H, 1) // repair heap
```

# *Heapsort* complexity

HEAPSORT($H$)

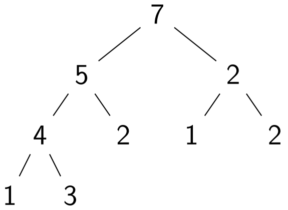
BUILD-HEAP($H$) // $O(n)$
**for** $i = |H|$ **downto** 2 // $n - 1$ times
    $H[1] \leftrightarrow H[i]$ // $O(1)$
    $heap\_size[H] = heap\_size[H] - 1$ // $O(1)$
    HEAPIFY($H, 1$) // $O(lgn)$

$$T(n) = O(n) + (n-1)[O(1) + O(lgn)]$$
$$T(n) = O(nlgn) = \Omega(nlgn)$$

- *Optimal* algorithm!

# *Heapsort* - example (*Build-Heap* done)

green - sorted; black - heap; blue - swapped; red - violate heap

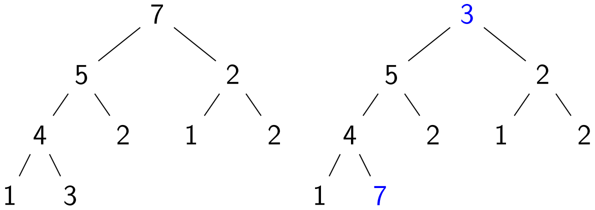



HEAPSORT($H$)

... // initial stuff...
**for** $i = |H|$ **downto** 2
    $H[1] \leftrightarrow H[i]$
    $heap\_size[H] - -$
    HEAPIFY(H,1)

# *Heapsort* - example (*Build-Heap* done)

green - sorted; black - heap; blue - swapped; red - violate heap

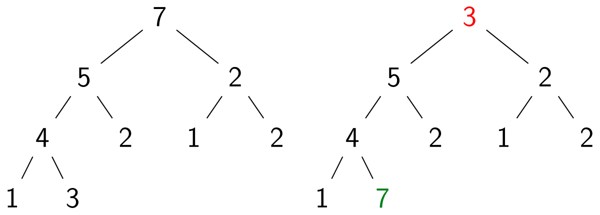



HEAPSORT($H$)

... // initial stuff...
**for** $i = |H|$ **downto** 2
$H[1] \leftrightarrow H[i]$
$heap\_size[H] --$
HEAPIFY(H,1)

# *Heapsort* - example (*Build-Heap* done)

green - sorted; black - heap; blue - swapped; red - violate heap

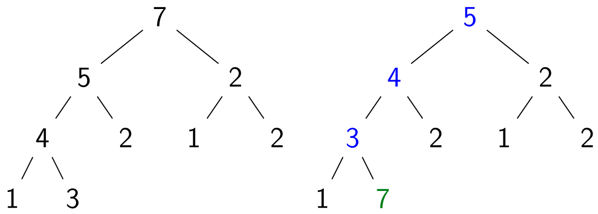



HEAPSORT($H$)

```
... // initial stuff...
for i = |H| downto 2
    H[1] ↔ H[i]
    heap_size[H] − −
    HEAPIFY(H,1)
```

# *Heapsort* - example (*Build-Heap* done)

green - sorted; black - heap; blue - swapped; red - violate heap

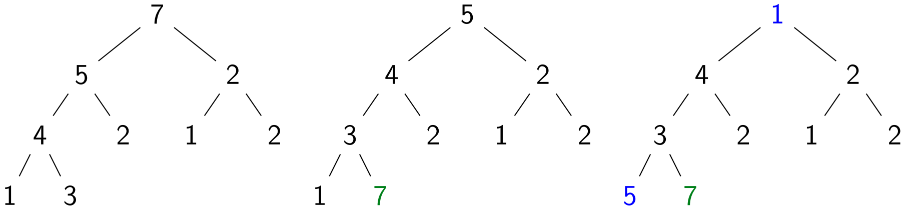



HEAPSORT($H$)

```
... // initial stuff...
for i = |H| downto 2
    H[1] ↔ H[i]
    heap_size[H] − −
    HEAPIFY(H,1)
```

# *Heapsort* - example (*Build-Heap* done)

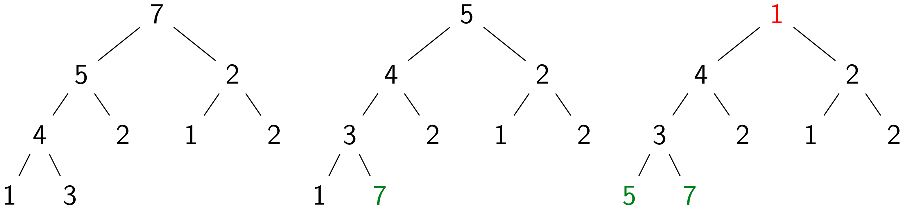
green - sorted; black - heap; blue - swapped; red - violate heap




```
HEAPSORT(H)
    ... // initial stuff...
    for i = |H| downto 2
        H[1] ↔ H[i]
        heap_size[H] − −
        HEAPIFY(H,1)
```

# *Heapsort* - example (*Build-Heap* done)

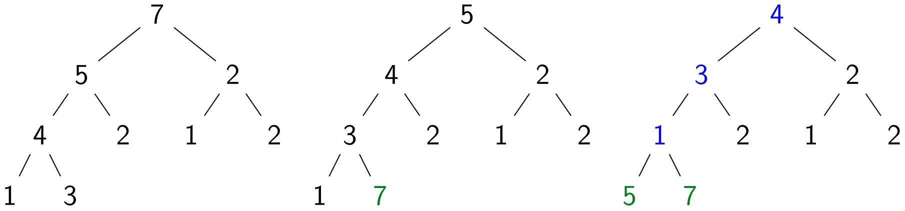
green - sorted; black - heap; blue - swapped; red - violate heap




```
HEAPSORT(H)
    ... // initial stuff...
    for i = |H| downto 2
        H[1] ↔ H[i]
        heap_size[H] − −
        HEAPIFY(H,1)
```

# *Heapsort* - example (*Build-Heap* done)

green - sorted; black - heap; blue - swapped; red - violate heap




HEAPSORT($H$)

... // initial stuff...

**for** $i = |H|$ **downto** 2

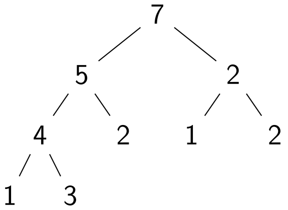
$H[1] \leftrightarrow H[i]$

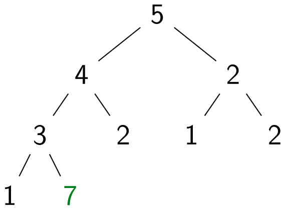
$heap\_size[H] - -$

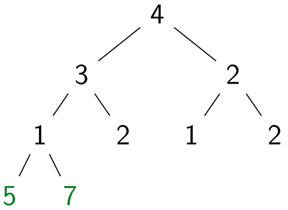
HEAPIFY(H,1)

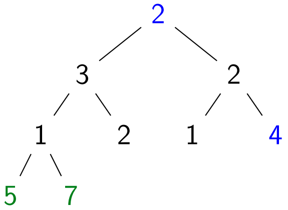
# *Heapsort* - example (*Build-Heap* done)

green - sorted; black - heap; blue - swapped; red - violate heap




HEAPSORT($H$)

... // initial stuff...

**for** $i = |H|$ **downto** 2

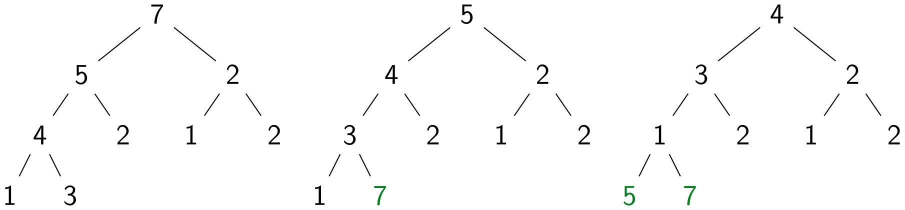
$H[1] \leftrightarrow H[i]$

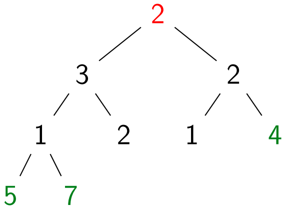
$heap\_size[H] - -$

HEAPIFY(H,1)

# *Heapsort* - example (*Build-Heap* done)

green - sorted; black - heap; blue - swapped; red - violate heap





HEAPSORT($H$)

... // initial stuff...

**for** $i = |H|$ **downto** 2

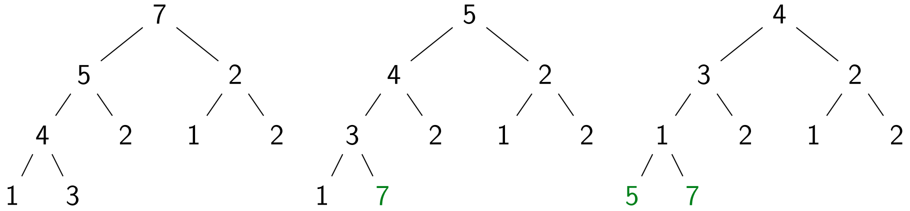
$H[1] \leftrightarrow H[i]$

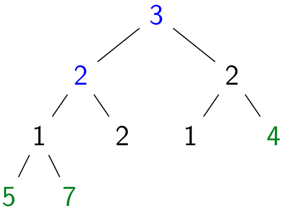
$heap\_size[H] - -$

HEAPIFY(H,1)

# *Heapsort* - example (*Build-Heap* done)

green - sorted; black - heap; blue - swapped; red - violate heap




HEAPSORT($H$)

... // initial stuff...

**for** $i = |H|$ **downto** 2

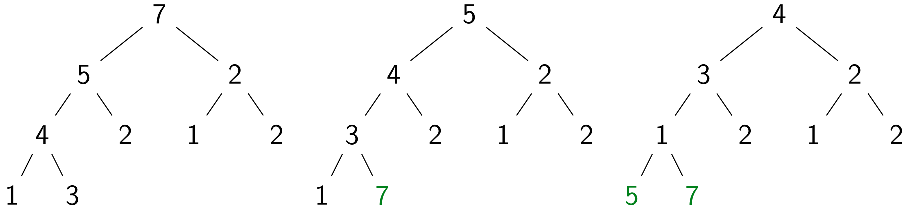
$H[1] \leftrightarrow H[i]$

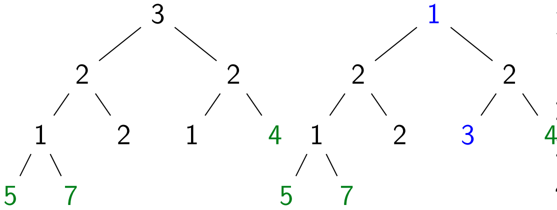
*heap_size*$[H]--$

HEAPIFY(H,1)

# *Heapsort* - example (*Build-Heap* done)

green - sorted; black - heap; blue - swapped; red - violate heap




$\textsc{Heapsort}(H)$

... // initial stuff...

**for** $i = |H|$ **downto** 2

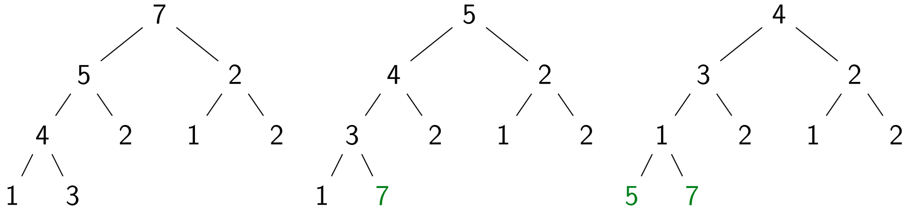
$\quad H[1] \leftrightarrow H[i]$

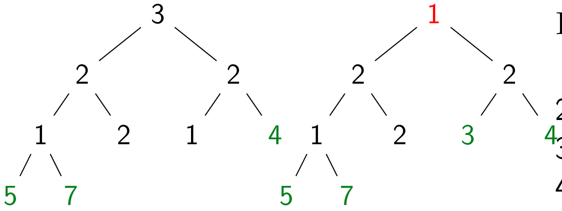
$\quad heap\_size[H] --$

$\quad \textsc{Heapify}(\mathrm{H}, 1)$

# *Heapsort* - example (*Build-Heap* done)

green - sorted; black - heap; blue - swapped; red - violate heap

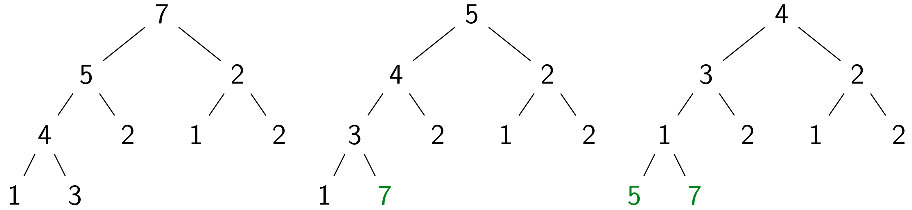


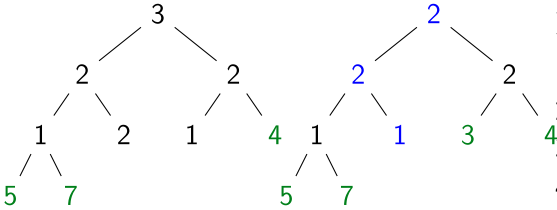




```
HEAPSORT(H)
    ... // initial stuff...
    for i = |H| downto 2
        H[1] ↔ H[i]
        heap_size[H] − −
        HEAPIFY(H,1)
```

# *Heapsort* - example (*Build-Heap* done)

green - sorted; black - heap; blue - swapped; red - violate heap




HEAPSORT($H$)

... // initial stuff...

**for** $i = |H|$ **downto** 2

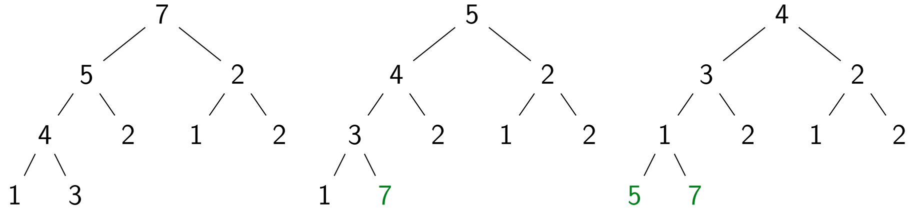
$H[1] \leftrightarrow H[i]$

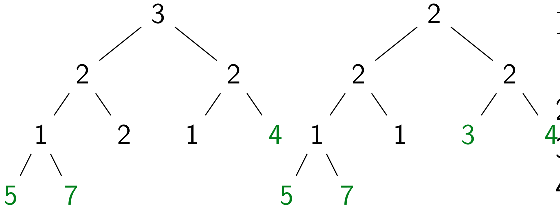
*heap_size*$[H]--$

HEAPIFY(H,1)

# *Heapsort* - example (*Build-Heap* done)

green - sorted; black - heap; blue - swapped; red - violate heap




$\textsc{Heapsort}(H)$

... // initial stuff...

**for** $i = |H|$ **downto** 2

$H[1] \leftrightarrow H[i]$

$heap\_size[H] - -$

$\textsc{Heapify}(\mathrm{H}, 1)$

# Heap as a Data Structure

- *Build-Heap* used when all elements are known in advance

# Heap as a Data Structure

- *Build-Heap* used when all elements are known in advance
- What if you need to accommodate a dynamic collection?

# Heap as a Data Structure

- *Build-Heap* used when all elements are known in advance
- What if you need to accommodate a dynamic collection?
  - need insert/delete operations!

# Heap as a Data Structure

- *Build-Heap* used when all elements are known in advance
- What if you need to accommodate a dynamic collection?
  - need insert/delete operations!
  - need *size* associated with the structure

# Heap as a Data Structure

- *Build-Heap* used when all elements are known in advance
- What if you need to accommodate a dynamic collection?
  - need insert/delete operations!
  - need *size* associated with the structure
- Operations?

# Heap as a Data Structure

- *Build-Heap* used when all elements are known in advance
- What if you need to accommodate a dynamic collection?
  - need insert/delete operations!
  - need *size* associated with the structure
- Operations?
  - delete: *HEAP-POP*, a.k.a. *HEAP-EXTRACT-MAX*

# Heap as a Data Structure

- *Build-Heap* used when all elements are known in advance
- What if you need to accommodate a dynamic collection?
  - need insert/delete operations!
  - need *size* associated with the structure
- Operations?
  - delete: *HEAP-POP*, a.k.a. *HEAP-EXTRACT-MAX*
    - extract the top from the heap

# Heap as a Data Structure

- *Build-Heap* used when all elements are known in advance
- What if you need to accommodate a dynamic collection?
  - need insert/delete operations!
  - need *size* associated with the structure
- Operations?
  - delete: *HEAP-POP*, a.k.a. *HEAP-EXTRACT-MAX*
    - extract the top from the heap
  - insert: *HEAP-PUSH*, a.k.a. *MAX-HEAP-INSERT*

# Heap as a Data Structure

- *Build-Heap* used when all elements are known in advance
- What if you need to accommodate a dynamic collection?
  - need insert/delete operations!
  - need *size* associated with the structure
- Operations?
  - delete: *HEAP-POP*, a.k.a. *HEAP-EXTRACT-MAX*
    - extract the top from the heap
  - insert: *HEAP-PUSH*, a.k.a. *MAX-HEAP-INSERT*
    - add one item to the heap

# Heap as a Data Structure - *POP-HEAP*

POP-HEAP($H$)

```
if heap_size[H] < 1 // empty heap
        return
max = H[1] // save root (max)
H[1] = H[heap_size[H]] // move bottom element to root
heap_size[H] = heap_size[H] − 1 // decrease heap size
HEAPIFY(H,1) // push element in root position down, to restore he
return max // return max element
```

- Complexity?

# Heap as a Data Structure - *POP-HEAP*

```
POP-HEAP(H)
if heap_size[H] < 1 // empty heap
        return
max = H[1] // save root (max)
H[1] = H[heap_size[H]] // move bottom element to root
heap_size[H] = heap_size[H] − 1 // decrease heap size
HEAPIFY(H,1) // push element in root position down, to restore he
return max // return max element
```

- Complexity?
  - $T(n) = O(lgn)$

# Heap as a Data Structure - *PUSH-HEAP*

$\text{PUSH-HEAP}(H, key)$

$heap\_size[H] = heap\_size[H] + 1$ // increase heap size
$H[heap\_size[H]] = key$
$i = heap\_size[H]$
**while** $i > 1$ and $H[\text{PARENT}(i)] < H[i]$
    $H[i] \leftrightarrow H[\text{PARENT}(i)]$
    $i = \text{PARENT}(i)$

- Complexity?

# Heap as a Data Structure - *PUSH-HEAP*

$\text{PUSH-HEAP}(H, key)$

$heap\_size[H] = heap\_size[H] + 1$ // increase heap size
$H[heap\_size[H]] = key$
$i = heap\_size[H]$
**while** $i > 1$ and $H[\text{PARENT}(i)] < H[i]$
    $H[i] \leftrightarrow H[\text{PARENT}(i)]$
    $i = \text{PARENT}(i)$

- Complexity?
  - $T(n) = O(lgn)$

# Heap as a Data Structure - *PUSH-HEAP*

PUSH-HEAP($H$, *key*)

$heap\_size[H] = heap\_size[H] + 1$ // increase heap size
$H[heap\_size[H]] = key$
$i = heap\_size[H]$
**while** $i > 1$ and $H[\text{PARENT}(i)] < H[i]$
    $H[i] \leftrightarrow H[\text{PARENT}(i)]$
    $i = \text{PARENT}(i)$

- Complexity?
  - $T(n) = O(lgn)$
- Complexity of *building the heap* by repeated inserts?

# Heap as a Data Structure - *PUSH-HEAP*

PUSH-HEAP$(H, key)$

$heap\_size[H] = heap\_size[H] + 1$ // increase heap size
$H[heap\_size[H]] = key$
$i = heap\_size[H]$
**while** $i > 1$ and $H[\text{PARENT}(i)] < H[i]$
    $H[i] \leftrightarrow H[\text{PARENT}(i)]$
    $i = \text{PARENT}(i)$

- Complexity?
  - $T(n) = O(lgn)$
- Complexity of *building the heap* by repeated inserts?
  - $T(n) = 2 \cdot 1 + 4 \cdot 2 + 8 \cdot 3 + ... + \frac{n}{2} \cdot lgn$

# Heap as a Data Structure - *PUSH-HEAP*

$\text{PUSH-HEAP}(H, \mathit{key})$

```
heap_size[H] = heap_size[H] + 1 // increase heap size
H[heap_size[H]] = key
i = heap_size[H]
while i > 1 and H[PARENT(i)] < H[i]
        H[i] ↔ H[PARENT(i)]
        i = PARENT(i)
```

- Complexity?
  - $T(n) = O(lgn)$
- Complexity of *building the heap* by repeated inserts?
  - $T(n) = 2 \cdot 1 + 4 \cdot 2 + 8 \cdot 3 + ... + \frac{n}{2} \cdot lgn$
  - $T(n) = O(nlgn)$

# Building the heap - comparison

| Approach | Bottom-up | Top-down |
|---|---|---|
| Approach for 1 elem | sinks the root | from root, add new leaf |
| Approach for all elems | from leaves towards root | bubbles a leaf |
| Complexity | $O(n)$ | $O(n \cdot lgn)$ |
| Advantage | faster | handle variable dimension |
| Usage | sorting | priority queues |

# *Heapsort* – Conclusions

- Optimal!

# *Heapsort* – Conclusions

- Optimal!

  ... but ...

# *Heapsort* – Conclusions

- Optimal!

  ... but ...

- *Quicksort* is faster in practice! (even if not optimal, in classic approach)

# *Heapsort* – Conclusions

- Optimal!

  ... but ...

- *Quicksort* is faster in practice! (even if not optimal, in classic approach)
- Good *Quicksort* implementations are (considered) optimal

# Bibliography


- Cormen, Thomas H., et al., *"Introduction to algorithms."*, MIT press, 2009, chap. 6 and 8.1 (sorting lower bound)