Lecture - 5 Heaps, Heapsort

CS313E - Elements of Software Design

Kia Teymourian

05/11/2022

Agenda

Priority Queue

A Priority Queue is a data structure that implements a set of S elements, each associated with a specific key and supporting the following operation to be done on this data structure:

- **insert(S, x):** The insert operation adds an element x into the existing Set S
- **max(S)**: The max operation returns the element with the largest key out of the Set S.
- **extract_Max(S):** It returns the element with the largest key and remove it from the Set S.
- **Increase_Key(S, x, k)**: It increases the value of x's key to the new value k (assumed to be as large as x's current key value).

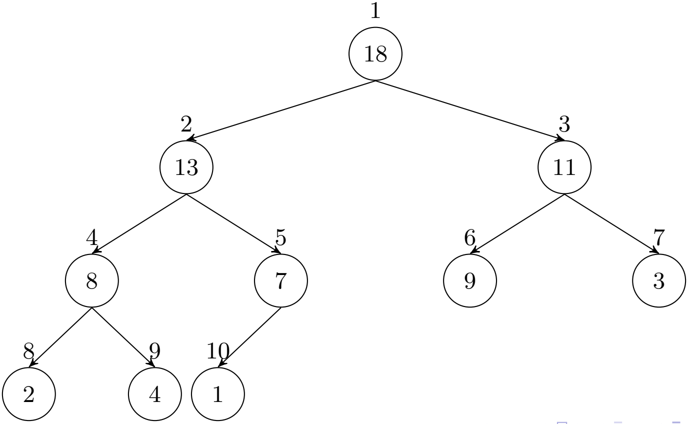
A **Heap** can used in implementation of a Priority Queue.

What is a Heap?

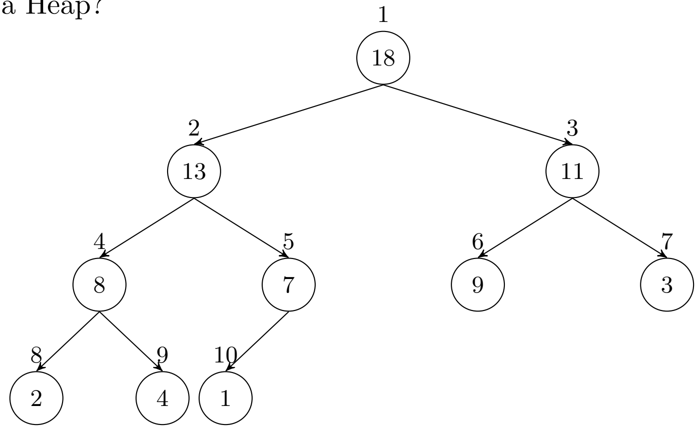
1	2	3	4	5	6	7	8	9	10
18	13	11	8	7	9	3	2	4	1

What is a Heap?

1	2	3	4	5	6	7	8	9	10
18	13	11	8	7	9	3	2	4	1



What is a Heap?

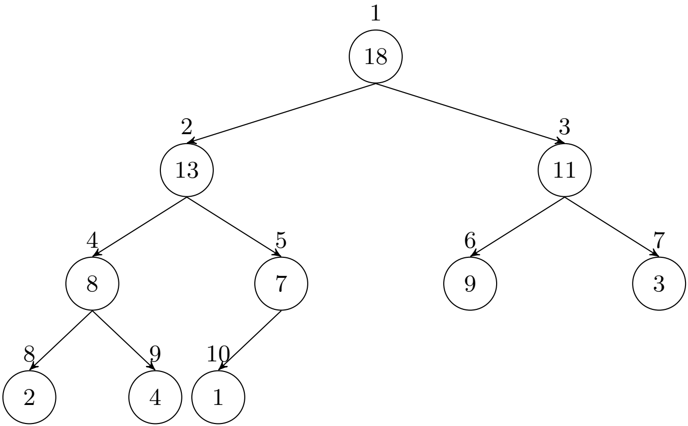


- An array can be visualized in binary tree form.
- There are two kinds of binary heaps: **max-heaps** and **min-heaps**.
- The values in the nodes satisfy a **heap property**.

Max Heap

If a Heap has the **max-heap property** it is called **max-heap**.

- In a max-heap, every node i other than the root $A[parent(i)] \geq A[i]$, the value of each node is at most the value of its parent.
- Min-heap is defined analogously.



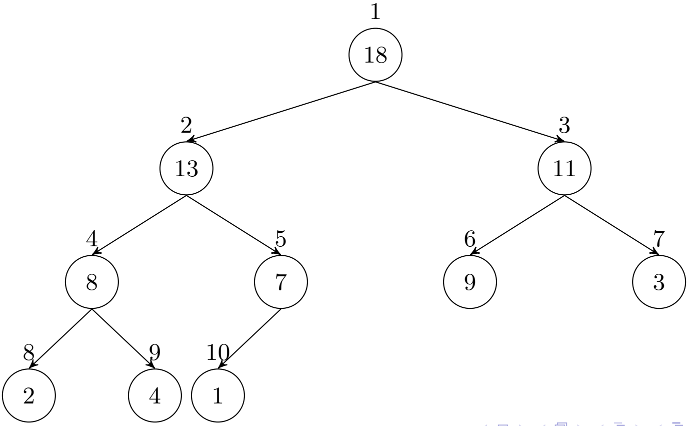
Max Heap

Operations on Heap to provide max-heap are:

- ▷ ***build_max_heap():*** Generates a Max-Heap from an unordered array.
- ▷ ***max_heapify():*** corrects a single violation of the heap property in a heap.

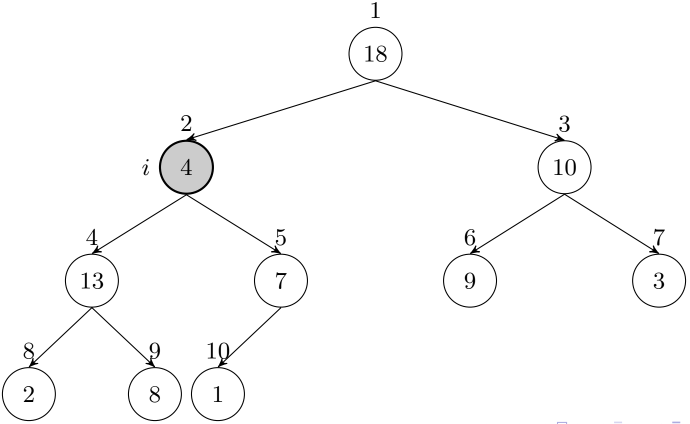
Other Heap Operations:

- ▷ *insert*(S, x), *extract_max*(S), *heapsort*(S)



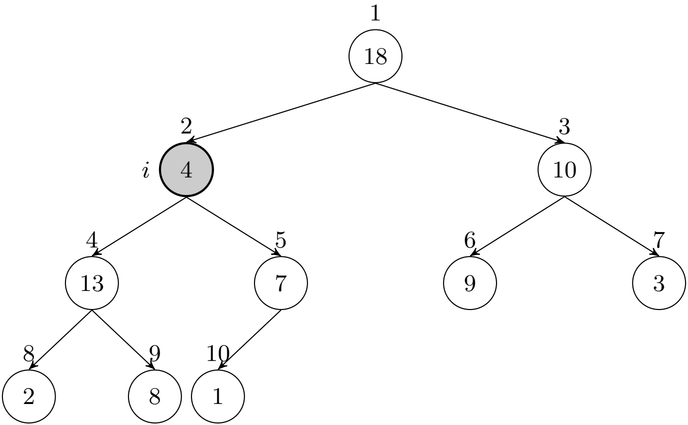
Example of a **NOT Max-Heap**

1	2	3	4	5	6	7	8	9	10
18	4	10	13	7	9	3	2	8	1



Parent, Left and Right

- Parent(i): return $\lfloor i/2 \rfloor$
- Left(i): return $2i$
- Right(i): return $2i + 1$



Heap Procedures

Basic procedures and shows how they are used in a sorting algorithm and a priority-queue data structure.

- The **MAX-HEAPIFY** procedure, which runs in $\boldsymbol{O(lg(n))}$ time, is the key to maintaining the max-heap property.
- The **BUILD-MAX-HEAP** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in $\boldsymbol{O(n\ lg(n))}$ time, sorts an array in place.
- The **MAX-HEAP-INSERT, HEAP-EXTRACT-MAX , HEAP-INCREASE-KEY and HEAP-MAXIMUM** procedures, which run in $\boldsymbol{O(lg(n))}$ time, allow the heap data structure to implement a priority queue.

MAX-HEAPIFY Algorithm

To maintain the max-heap property, we use MAX-HEAPIFY algorithm.

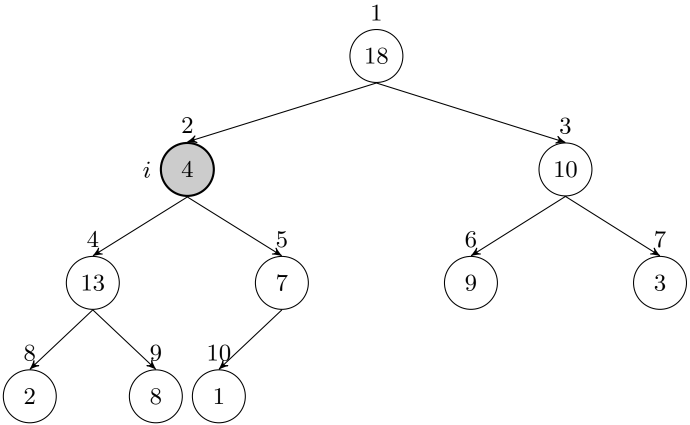
1. At each step, the largest of the elements $A[i], A[LEFT(i)]$ and $A[RIGHT(i)]$ is determined, and its index is stored in largest.
2. If $A[i]$ is largest, then already max-heap and the algorithm terminates.
3. Otherwise, one of the two sub-nodes has the largest
4. Then swap $A[i]$ with the *"largest"* which then creates the Max-Heap property for the node i
5. Now, then node *"largest"* has the $A[i]$. Now, we need to check the subtree rooted at *"largest"*, so call MAX-HEAPIFY recursively on that subtree.

MAX-HEAPIFY Algorithm

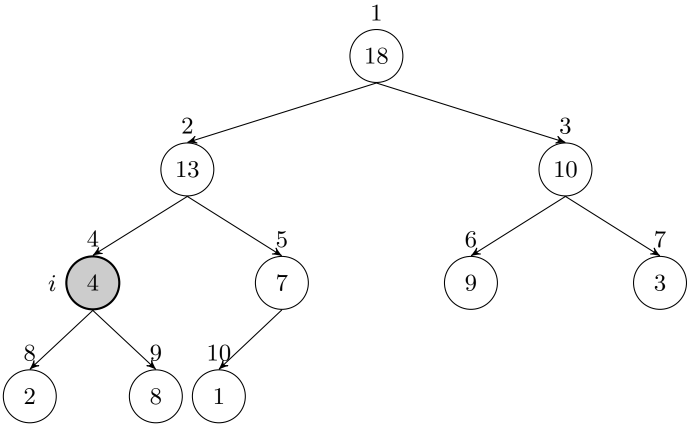
Algorithm 1 MAX-HEAPIFY(A, i)

$l = LEFT(i)$
$r = RIGHT(i)$
if $l \leq A.heapSize$ and $A[l] > A[i]$ **then**
    $largest = l$
else
    $largest = i$
end if
if $r \leq A.heapSize$ and $A[r] > A[largest]$ **then**
    $largest = r$
end if
if $largest \neq i$ **then**
    exchange $A[i]$ with $A[largest]$
    MAX-HEAPIFY(A, largest) ▷ Recursuve call
end if

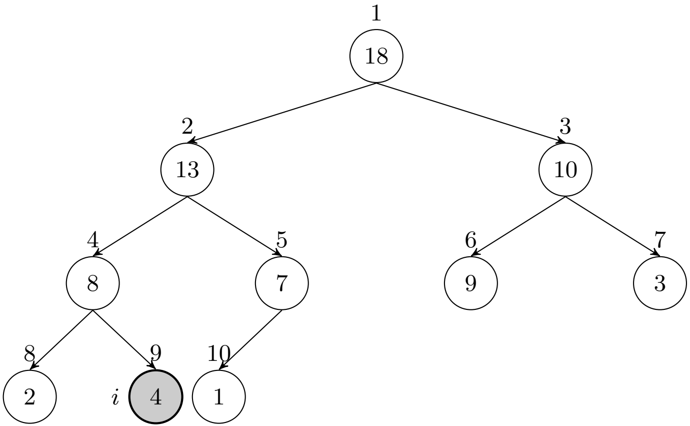
MAX-HEAPIFY Algorithm



MAX-HEAPIFY Algorithm



MAX-HEAPIFY Algorithm



Running time of MAX-HEAPIFY Algorithm

The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is:

1. The $\boldsymbol{O(1)}$ time to fix up the relationships among the elements $A[i]$, $A[Left(i)]$, and $A[Right(i)]$, **plus**
2. The time to run MAX-HEAPIFY on a subtree rooted at one of the children.

Now, consider that

- Children's subtrees each have size at most $\boldsymbol{2n/3}$ (the worst case occurs when the bottom level of the tree is exactly half full)
- Thus, the running time of MAX-HEAPIFY by the recurrence

$$\boldsymbol{T(n) < T(2n/3) + \Theta(1)}$$

- Using the **Master Theorem** case 2, we have $\boldsymbol{T(n) = O(lg(n))}$
- MAX-HEAPIFY on a node i with height h is $O(h)$

BUILD-MAX-HEAP Algorithm

The procedure BUILD-MAX-HEAP iterates through the nodes of the tree and executes MAX-HEAPIFY on each one.

Algorithm 2 BUILD-MAX-HEAP(A)

```
A.heapSize = A.lenght
for i = ⌊A.length/2⌋ downTo 1 do
    MAX-HEAPIFY(A,i)                          ▷ Recursuve call
end for
```

Run time of BUILD-MAX-HEAP Algorithm

We know that MAX-HEAPIFY on a node i with height h is $O(h)$

The total cost of BUILD-MAX-HEAP is bounded from above by

$$\sum_{h=0}^{\lfloor lg(n) \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h)$$

Run time of BUILD-MAX-HEAP Algorithm

We know that MAX-HEAPIFY on a node i with height h is $O(h)$

The total cost of BUILD-MAX-HEAP is bounded from above by

$$\sum_{h=0}^{\lfloor lg(n) \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h)$$

$$= O\left(n \sum_{h=0}^{\lfloor lg(n) \rfloor} \frac{h}{2^h}\right)$$

Run time of BUILD-MAX-HEAP Algorithm

$$O\left(n \sum_{h=0}^{\lfloor lg(n) \rfloor} \frac{h}{2^h}\right)$$

We know from Summation that:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Run time of BUILD-MAX-HEAP Algorithm

$$O\left(n \sum_{h=0}^{\lfloor lg(n) \rfloor} \frac{h}{2^h}\right)$$

We know from Summation that:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Now, use the above summation formula and evaluate the last summation by substituting $x = 1/2$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2}$$

$$= 2$$

Run time of BUILD-MAX-HEAP Algorithm

$$O\left(n\sum_{h=0}^{\lfloor lg(n)\rfloor}\frac{h}{2^h}\right) = O\left(n\sum_{h=0}^{\infty}\frac{h}{2^h}\right)$$
$$= O\left(2n\right)$$
$$= \mathbf{O(n)}$$

We can build a max-heap from an unordered array in **linear time**.

Extract-Max

Algorithm 3 HEAP-EXTRACT-MAX

if $A.heapSize < 1$ **then**
    error "heap underflow"
end if
$max = A[1]$
$A[1] = A[A.heapSize]$
$A.heapSize = A.heapSize - 1$
MAX-HEAPIFY(A,1) ▷ Recursuve call
return max

What is the running time of HEAP-EXTRACT-MAX?

Extract-Max

Algorithm 4 HEAP-EXTRACT-MAX

if $A.heapSize < 1$ **then**
    error "heap underflow"
end if
$max = A[1]$
$A[1] = A[A.heapSize]$
$A.heapSize = A.heapSize - 1$
MAX-HEAPIFY(A,1) ▷ Recursuve call
return max

What is the running time of HEAP-EXTRACT-MAX?

The running time of HEAP-EXTRACT-MAX is $O(lg(n))$

It is doing just a constant amount of work on top of the $O(lg(n))$ time for MAX-HEAPIFY.

HEAP-INCREASE-KEY

This implements the increase-KEY operation and keeps max-heap property.

- It repeatedly compares an element to its parent,
- Exchanging their keys and continuing if the element's key is larger,
- Terminating if the element's key is smaller because the max-heap holds.

HEAP-INCREASE-KEY

This implements the increase-KEY operation and keeps max-heap property.

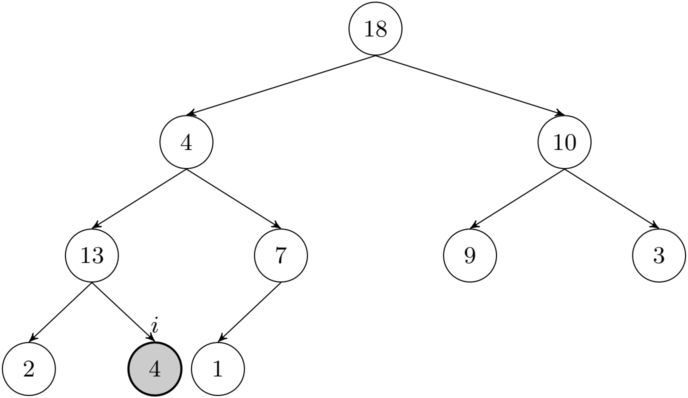
- It repeatedly compares an element to its parent,
- Exchanging their keys and continuing if the element's key is larger,
- Terminating if the element's key is smaller because the max-heap holds.

Algorithm 6 HEAP-INCREASE-KEY(A, i, key)

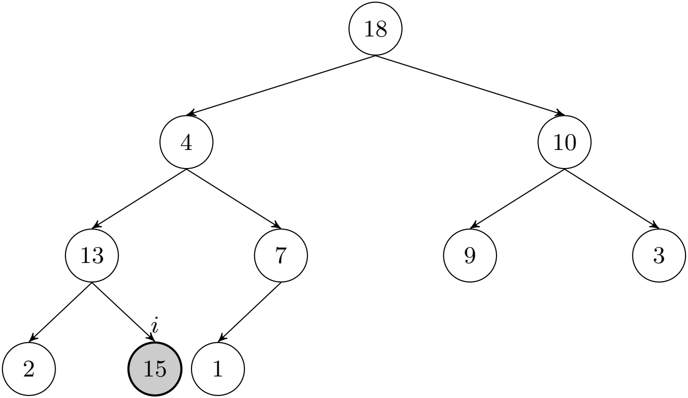
if $Key < A[i]$ **then**
    error "new key is smaller than current key"
end if
$A[i] = key$
while $i > 1$ and $A[PARENT(i)] < A[i]$ **do**
    Exchange $A[i]$ with $A[PARENT(i)]$
    $i = PARENT(i)$
end while

The running time of HEAP-INCREASE-KEY is $O(lg(n))$

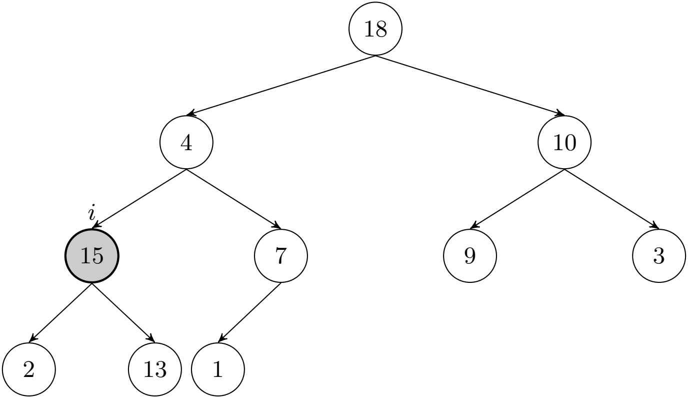
HEAP-INCREASE-KEY



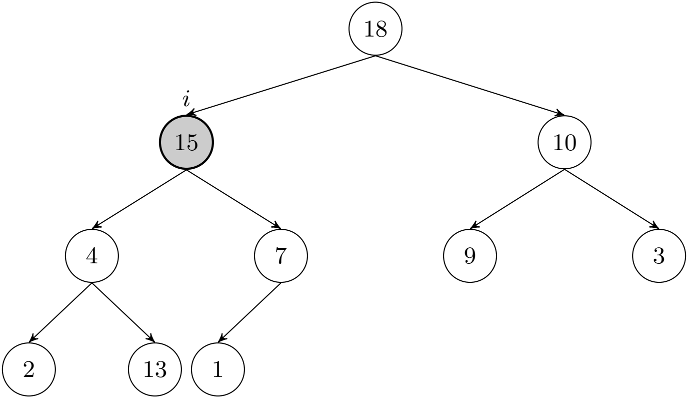
HEAP-INCREASE-KEY



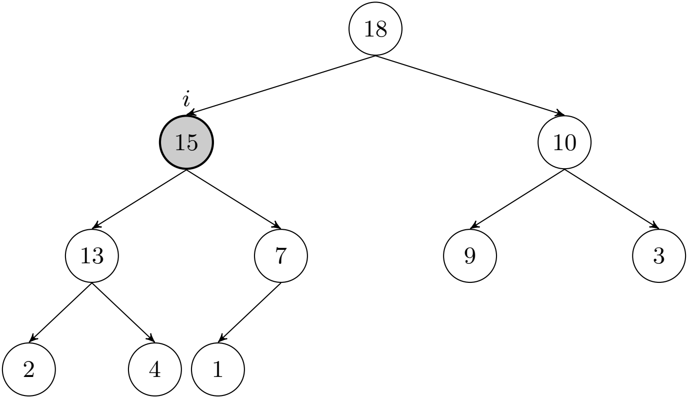
HEAP-INCREASE-KEY



HEAP-INCREASE-KEY



HEAP-INCREASE-KEY



Heap Sort Algorithm

Heap Sort Steps:

1. Build Max Heap from unordered array;
2. Find maximum element $A[1]$;
3. Swap elements $A[n]$ and $A[1]$: Now maximum element is located at the end of the array.
4. Discard node n from the heap by decrementing $heapSize$ variable
5. The new root might violate max-heap property but its children are valid max-heaps. Run **MAX-HEAPIFY(A, 1)** to fix this.
6. Go to up to the Step 2 unless heap is empty.

Heap Sort Algorithm

Algorithm 7 Heap sort Algorithm

BUILD-MAX-HEAP(A)
for $i = A.length$ **downTo** 2 **do**
    exchange $A[1]$ with $A[i]$
    $A.heapSize = A.heapSize - 1$
    MAX-HEAPIFY(A, 1)
end for

Heap Sort Algorithm

Algorithm 8 Heap sort Algorithm

```
BUILD-MAX-HEAP(A)
for i = A.length downTo 2 do
    exchange A[1] with A[i]
    A.heapSize = A.heapSize − 1
    MAX-HEAPIFY(A, 1)
end for
```

- The HEAPSORT procedure takes time $\boldsymbol{O(n\ lg(n))}$ since the call to BUILD-MAX-HEAP takes time $O(n)$
- Each of the $n-1$ calls to MAX-HEAPIFY takes time **O(lg(n))**.

Heap Visualization

Online Visualization Tools:

- http://btv.melezinek.cz/binary-heap.html
- https://visualgo.net/en/heap
- https://www.cs.csubak.edu/~msarr/visualizations/Heap.html

Closing Notes

- The **Heapsort algorithm** was invented by J. W. J. Williams in 1964.
  https://en.wikipedia.org/wiki/J._W._J._Williams
- The **BUILD-MAX-HEAP** is invented by Robert W. Floyd. (Algorithm 245 (TREESORT). Communications of the ACM, 7(12):701, 1964.)
  https://en.wikipedia.org/wiki/Robert_W._Floyd
  Floyd received the Turing Award in 1978.

Readings from CLRS Book (Introduction to Algorithms, 3rd Edition)

Chapter 6 Heapsort

- Section 6.1 Heaps
- Section 6.2 Maintaining the heap property
- Section 6.3 Building a heap
- Section 6.4 The heapsort algorithm
- Section 6.5 Priority queues