CH-231-A

Algorithms and Data Structures

ADS

Lecture 13

Dr. Kinga Lipskoch

Spring 2020

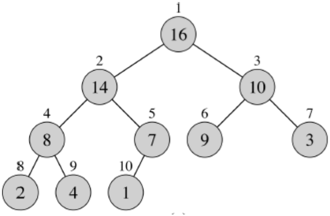
Heap Sort: Motivation

- Try to develop an in-situ sorting algorithm with asymptotic runtime $\Theta(n \lg n)$.
- Use a sophisticated data structure to support the computations.

Heap: Data structure

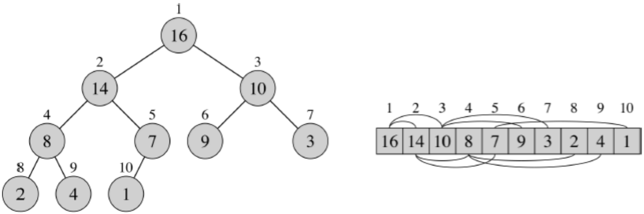
Defintion:

A (binary) heap data-structure is an array which can be viewed as a nearly complete binary tree: each level is completely full except possibly the last level, which is filled from left to right.



Heap as an Array (1)

A heap can be stored as an array:

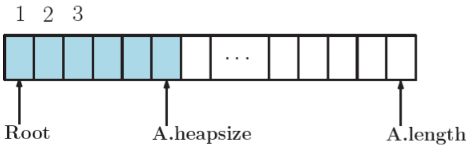


Heap as an Array (2)

The array A representing the heap has two attributes:

- $A.length$
- $A.heapsize$

such that $0 \leq A.heapsize \leq A.length$.
There are only $A.heapsize$ valid elements of the heap.

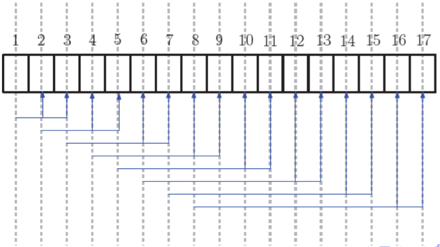


$A[1]$ is the root of the heap (root of the binary tree).

Heap as an Array (3)

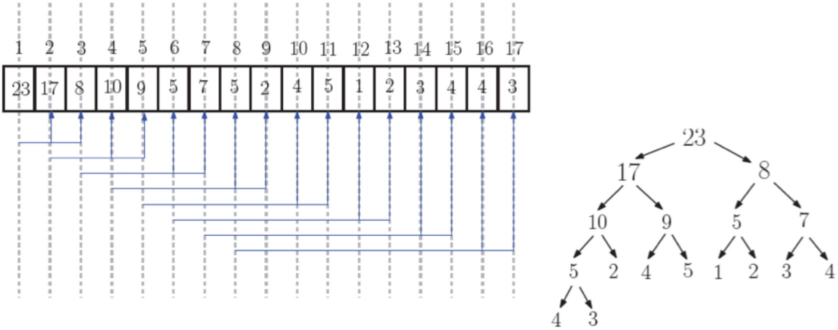
Given the index i of an element of A, we can calculate:

- `Parent(i): return floor(i/2);`
  `// Right shift by 1 bit`
- `Left(i): return 2i;`
  `// Left shift by 1 bit`
- `Right(i): return 2i + 1;`
  `// Left shift by 1 bit and set LSB to 1`



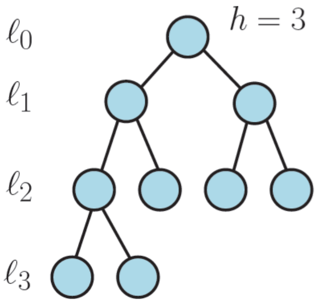
Max-Heap Property

In a max-heap, for every node i (other than the root), $A[Parent(i)] \geq A[i]$.



Recall: Height of a Tree

- The height of a node x is the length of the longest simple downward path from x to a leaf.
- The height of a tree is the height of its root.



Heap Height (1)

Theorem:
A heap with n elements has height $h = \lfloor \lg n \rfloor$.

Heap Height (2)

Proof:
Heap height h implies that there are $h+1$ levels (levels 0 to h).
As a heap is a nearly complete binary tree, the last guaranteed complete level is level $h-1$.
The level h may be incomplete, but it has at least one element.

The number of elements in complete levels 0 to $h-1$ is
$1+2+2^2+...+2^{h-1}=2^h-1$.
So, $n>2^h-1$ or (since it is an integer) $n\geq 2^h$.

If all levels 0 to h were complete, the number of elements would be
$2^{h+1}-1$.
So, $n\leq 2^{h+1}-1$.

Heap Height (3)

Proof (continued):
Combining the two inequalities:
$2^h \leq n \leq 2^{h+1} - 1$

As $2^{h+1} > 2^{h+1} - 1 \geq 2^h$ for $h \geq 0$,
$h + 1 > \lg(2^{h+1} - 1) \geq h$

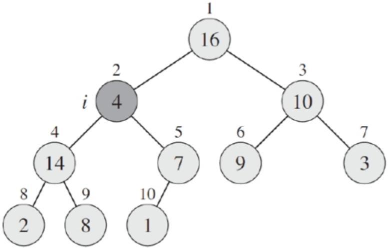
Thus, $\lg(2^{h+1} - 1) = h + \alpha$ with $\alpha \in [0, 1)$, which leads to
$h \leq \lg n \leq h + \alpha$ with $\alpha \in [0, 1)$.

Hence, $h = \lfloor \lg n \rfloor$.

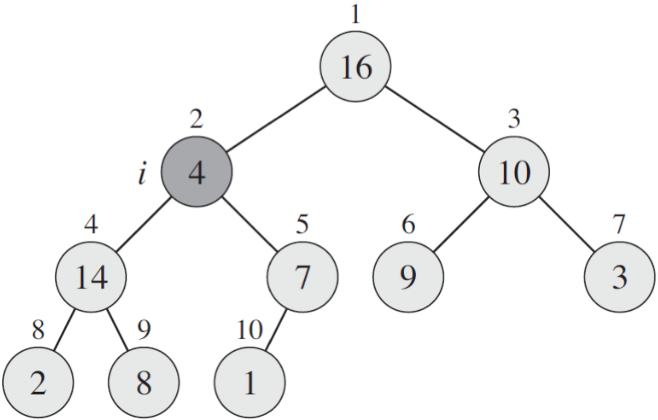
Max-Heapify(A, i) (1)

Precondition:

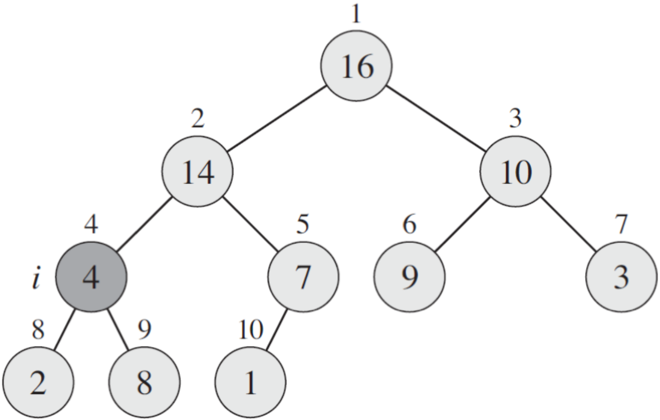
When *Max-Heapify*(A, i) is called, binary-trees rooted at *Left*(i) and *Right*(i) are valid max-heaps, but $A[i]$ may be smaller than its children.



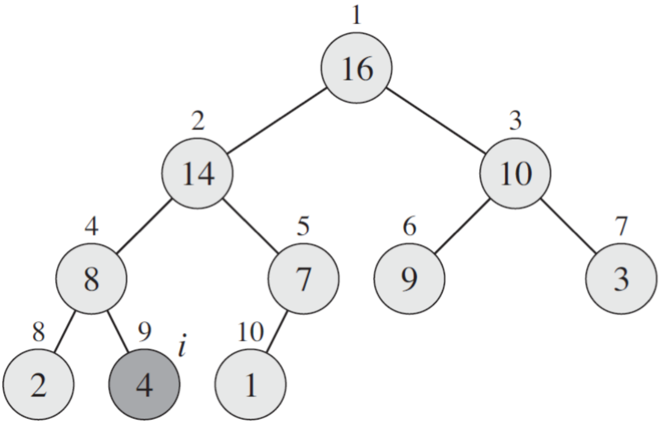
Max-Heapify(A, i) (2)



Max-Heapify(A, i) (3)



Max-Heapify(A, i) (4)



Max-Heapify(A, i) (5)

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Max-Heapify(A, i) (6)

Time complexity:

$T(n) = O(h) = O(\lg n)$,

as in the worst case the element from position i has to go down all the way to the last level.

In the expression above h is the height of the element from position i.