CSE 100: Algorithm Design and Analysis
Chapter 06: Heapsort

Sungjin Im

University of California, Merced

Last Update: 02-10-2023

For 37 years I've practiced 14 hours a day, and now they call me a genius.
Pablo de Sarasate (1844-1908)

CSE 100: Algorithm Design and Analysis
Chapter 06: Heapsort

Sungjin Im

University of California, Merced

Last Update: 02-10-2023

For 37 years I've practiced 14 hours a day, and now they call me a genius.
Pablo de Sarasate (1844-1908)

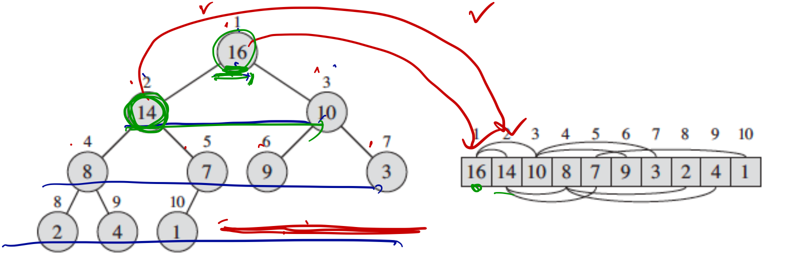
Outline

- What is a (binary) heap?
  - Learn its structural properties and operations
- Heap Applications
  - Priority queue
  - Heap-sort

(Binary) Heap

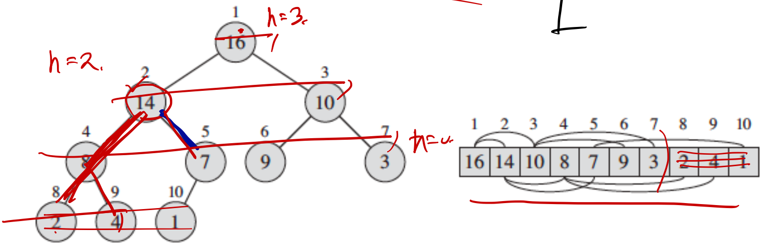
Heap A:

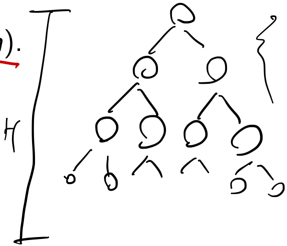
- 'almost' complete binary tree.
- Completely filled on all levels except possibly the lowest, which is filled from the left to the right.
- 1-to-1 mapping between heap and array.
- Node i means node indexed by i, which corresponds to $A[i]$.



Heap

- Height of node: # of edges on a longest simple path from the node down to a leaf.
- Height of heap: height of root $= \Theta(\log n)$.
- Other notation:
  - *A.arraysize*: # of elements in array *A*.
  - *A.heapsize*: # of elements in heap *A*.
  - Always *A.heapsize* $\leq$ *A.arraysize*.





$1 + 2 + 4 + 8 + \cdots + 2^H$

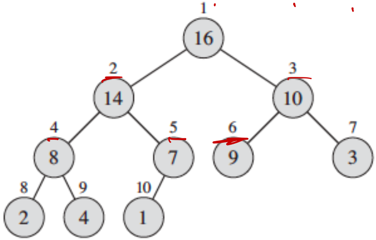
$\approx 2 \cdot 2^H - 1$

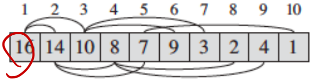
$= n$

Heap

Heap A:

- Root: $A[1]$
- Parent of $A[i]$: $A[\lfloor i/2 \rfloor]$
- Left Child of $A[i]$: $A[2i]$.
- Right Child of $A[i]$: $A[2i+1]$.

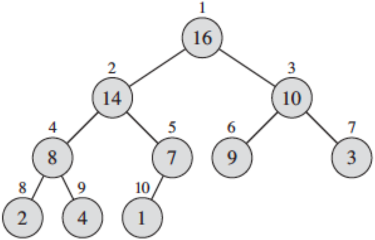


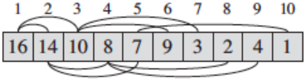


Heap

Heap A:

- Root: $A[1]$
- Parent of $A[i]$: $A[\lfloor i/2 \rfloor]$
- Left Child of $A[i]$: $A[2i]$.
- Right Child of $A[i]$: $A[2i+1]$.

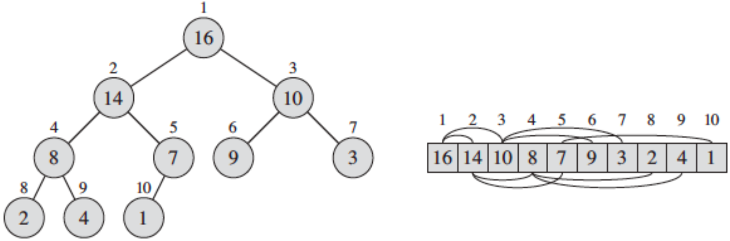




Heap

Heap A:

- Root: $A[1]$
- Parent of $A[i]$: $A[\lfloor i/2 \rfloor]$
- Left Child of $A[i]$: $A[2i]$.
- Right Child of $A[i]$: $A[2i+1]$.



Question: How can we test if node i has a left child? $2i \le A.heapsize.$

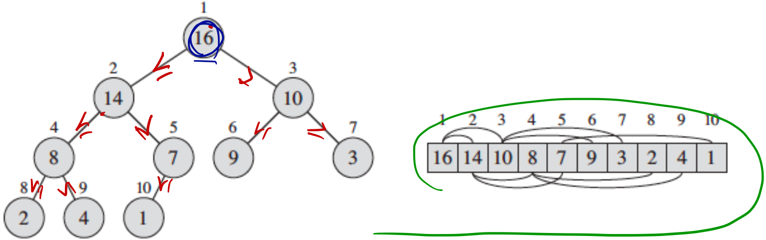
Max Heap

10

9 8

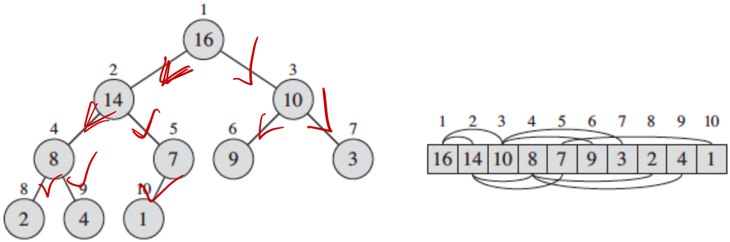
7 6 4 3

We say A is a max-heap if it satisfies the max-heap property: for *every* node i other than the root, $A[parent(i)] \geq A[i]$.



Max Heap

We say A is a max-heap if it satisfies the max-heap property: for *every* node i other than the root, $A[parent(i)] \geq A[i]$.



The largest element is stored at the root.

Min Heap

We say A is a min-heap if it satisfies the min-heap property:
for *every* node i other than the root, $A[parent(i)] \leq A[i]$.

Min Heap

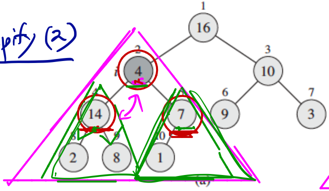
We say A is a min-heap if it satisfies the min-heap property:
for *every* node i other than the root, $A[parent(i)] \leq A[i]$.
The smallest element is stored at the root.

Building a (max-)heap

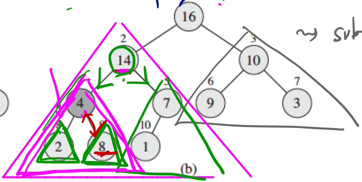
Helper function/procedure: Max-Heapify

Max-Heapify(i): If both the left and right subtrees of node i are max-heaps, make the subtree rooted at node i a max-heap.

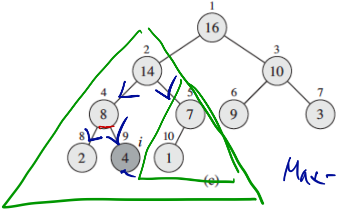
Max-Heapify (2)



Max-Heapify (4)



⇝ subtree rooted at node 3.

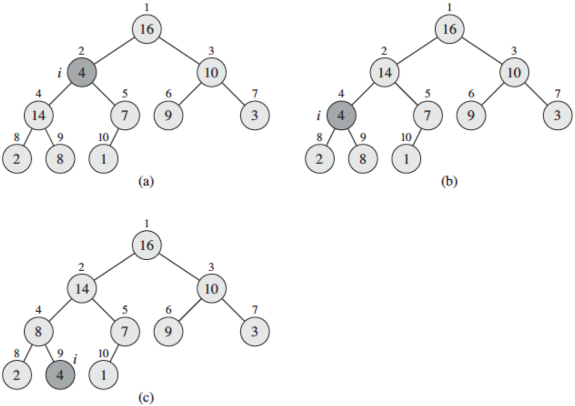


Max-Heapify (9)

Building a (max-)heap

Helper function/procedure: Max-Heapify

Max-Heapify(i): If both the left and right subtrees of node i are max-heaps, make the subtree rooted at node i a max-heap.



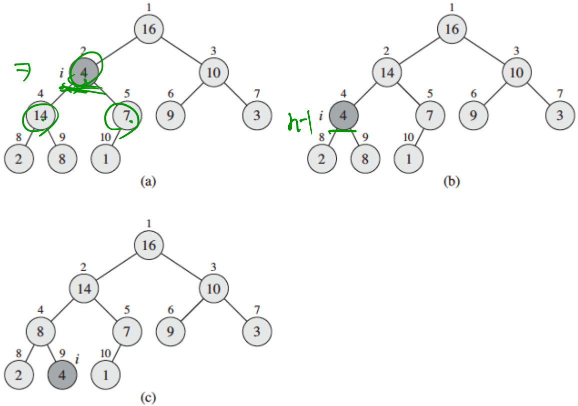
Correctness?

Building a (max-)heap

Helper function/procedure: Max-Heapify

Max-Heapify(i): If both the left and right subtrees of node i are max-heaps, make the subtree rooted at node i a max-heap.

i's height = h. ⇒

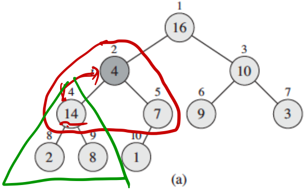


Correctness? Running time?

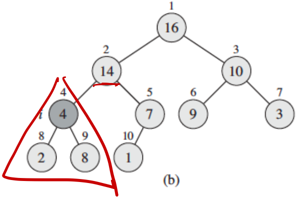
Building a (max-)heap

Helper function/procedure: Max-Heapify

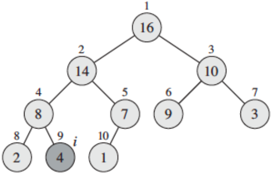
Max-Heapify(i): If both the left and right subtrees of node i are max-heaps, make the subtree rooted at node i a max-heap.



(a)



(b)



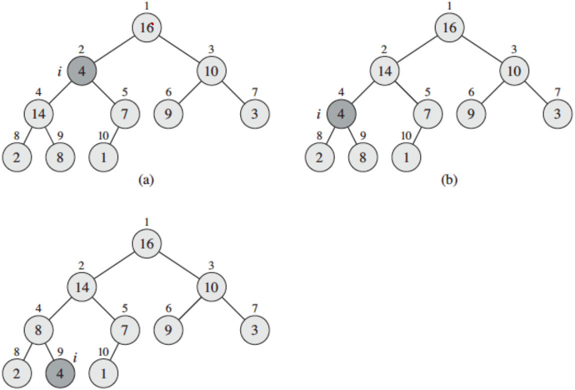
(c)

Correctness? Running time? $O(\lg n)$.

Building a (max-)heap

Helper function/procedure: Max-Heapify

Max-Heapify(i): If both the left and right subtrees of node i are max-heaps, make the subtree rooted at node i a max-heap.



(a) (b) (c)

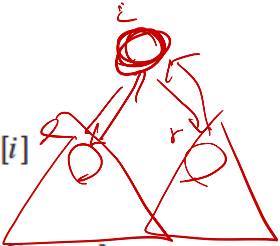
Correctness? Running time? $O(\lg n)$. More precisely, $O(i\text{'s height})$

Building a (max-)heap

Helper procedure: Max-Heapify

Max-Heapify(i): If both the left and right subtrees of node i are max-heaps, make the subtree rooted at node i a max-heap.

```
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)
```



Building a (max-)heap

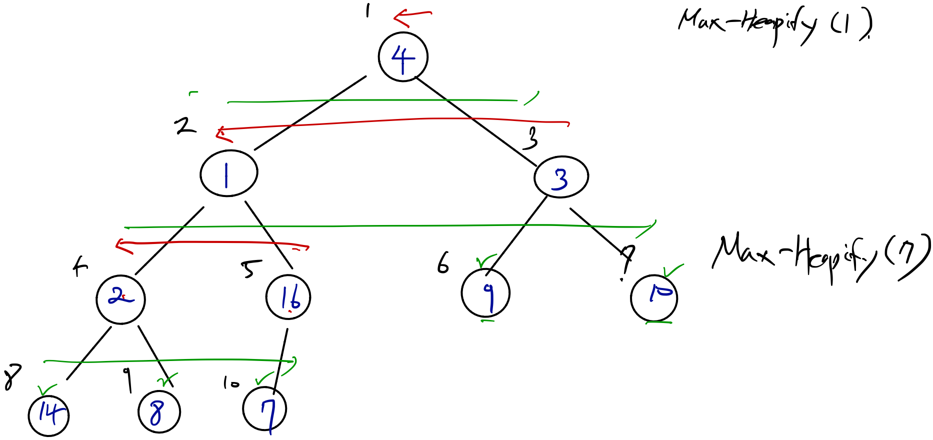
Helper procedure: Max-Heapify

Max-Heapify(i): If both the left and right subtrees of node i are max-heaps, make the subtree rooted at node i a max-heap.

To do: Write a pseudocode on your own.

Building a (max-)heap

Example: $A[1...10] = \{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$.



Building a (max-)heap

$i =$ A.length is okay.

BUILD-MAX-HEAP(A)

1 $A.heap\text{-}size = A.length$
2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1
3 MAX-HEAPIFY(A, i)

* Nodes $\lfloor n/2 \rfloor + 1$, ..., n are leaves.

Building a (max-)heap

BUILD-MAX-HEAP(A)

```
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

$\Rightarrow O(\log n)$ $* n.$

Loop Invariant: At start of each iteration of For loop, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.

1. Initialization: Each of nodes $\lfloor n/2 \rfloor + 1, \ldots, n$ is a leaf, thus the root of a max-heap.
2. Maintenance: Children of node $i \leq \lfloor n/2 \rfloor$ are indexed higher than i, so are roots of max-heaps. Max-Heapify makes node i a max-heap root. $i--$ reestablishes the loop invariant for the next iteration
3. Termination: When $i = 0$. By the invariant, node 1 is the root of a max-heap.

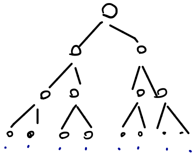
Building a Max- heap

Running time

Naive: $O(n)$ iterations of the for loop. Each call of Max-Heapify takes $O(\log n)$ time. So, $O(n \log n)$.

Building a heap

Running time



R.T #

$\Rightarrow$ $C * \log n$. 1 $= \frac{n}{2^{\log_2 n}}$

$\log n - 1$ 2.

$C * h$

$\Rightarrow$ $C * 1$ $n/2$.

Naive: $O(n)$ iterations of the for loop. Each call of Max-Heapify takes $O(\log n)$ time. So, $O(n \log n)$.

Better analysis:
$\#$ of nodes of height $h = O(n/2^{h+1})$.
Height of heap: $O(\log n)$
Max-Heapify on each node of height h takes $O(h)$ time
So, $\sum_{h=0}^{O(\log n)} O(n/2^{h+1}) \cdot O(h) = n \cdot O(\sum_{h=0}^{O(\log n)} \frac{h}{2^h}) =$
$n \cdot O(\sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$.

Heapsort

Example: $A[1...10] = \{16, 14, 10, 8, 7, 9, 3, 2, 4, 1\}$.

Heapsort

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
```

Heapsort

```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)                          → O(n)
    for i = A.length downto 2
        exchange A[1] with A[i]                → O(1)
        A.heap-size = A.heap-size − 1          → O(1)
        MAX-HEAPIFY(A, 1)                      → O(lg n)
                                               *n.
```

Running time: $O(n \log n)$.

Priority Queue

- Maintains a dynamic set S of elements.
- Each element has a *key*/value.

Max-priority queue:

- Insert(S, x): inserts x into S.
- Maximum(S): returns element of S with largest key.
- Extract-Max(S): removes and returns element of S with largest key.
- Increase-Key(S, x, k): increases x's key to k.

Application: scheduling

Priority Queue

- Maintains a dynamic set S of elements.
- Each element has a *key* (value).

Min-priority queue:

- Insert(S, x): inserts x into S.
- Minimum(S): returns element of S with smallest key.
- Extract-Min(S): removes and returns element of S with smallest key.
- Decrease-Key(S, x, k): decreases x's key to k.

Application: event-driven simulator

Priority Queue

- Maintains a dynamic set S of elements.
- Each element has a *key* (value).

Min-priority queue:

- Insert(S, x): inserts x into S.
- Minimum(S): returns element of S with smallest key.
- Extract-Min(S): removes and returns element of S with smallest key.
- Decrease-Key(S, x, k): decreases x's key to k.

Application: event-driven simulator

Max-priority queue

Heap-Maximum(A): Return the maximum key.

Max-priority queue

Heap-Maximum(A): Return the maximum key.

HEAP-MAXIMUM(A)
1 **return** $A[1]$ $\rightarrow O(1)$

Max-priority queue

Heap-Maximum(A): Return the maximum key.

HEAP-MAXIMUM(A)
1 **return** $A[1]$

RT:

Max-priority queue

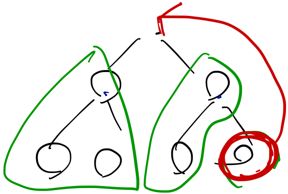
Heap-Maximum(A): Return the maximum key.

HEAP-MAXIMUM(A)
1 **return** $A[1]$

RT: $\Theta(1)$.

Max-priority queue

Heap-Extract-Max(A): Extract the maximum key.



Max-priority queue

Heap-Extract-Max(A): Extract the maximum key.

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
        error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size − 1
MAX-HEAPIFY(A, 1)
return max
```

Max-priority queue

Heap-Extract-Max(A): Extract the maximum key.

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

RT:

Max-priority queue

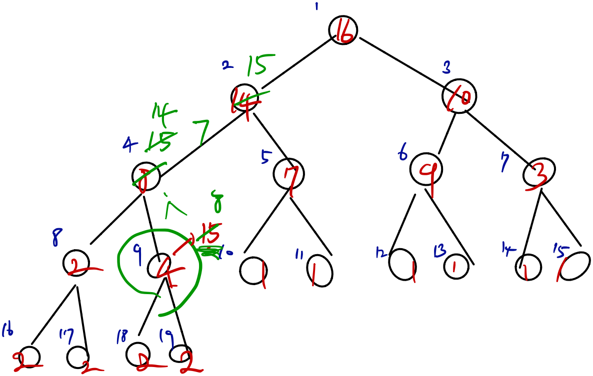
Heap-Extract-Max(A): Extract the maximum key.

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

RT: $O(\log n)$.

Max-priority queue

Heap-Increase-Key(A, i, key): Increase node i's key value to key.
Example: Heap-Increase-Key($A, 9, 15$) when
$A[1...19] = \{16, 14, 10, 8, 7, 9, 3, 2, 4, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2\}$.



Max-priority queue

Heap-Increase-Key(A, i, key): Increase node i's key value to key.

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

Max-priority queue

Heap-Increase-Key(A, i, key): Increase node i's key value to key.

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

RT:

Max-priority queue

Heap-Increase-Key(A, i, key): Increase node i's key value to key.

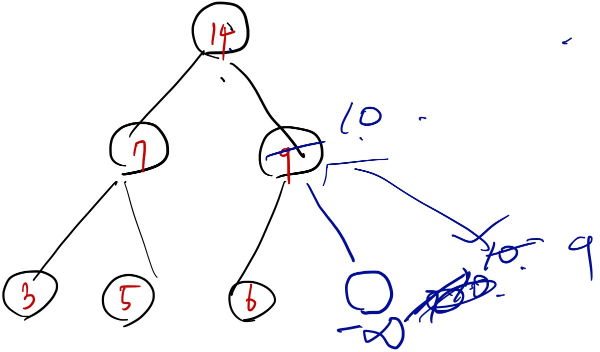
```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
        error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)
```

RT: $O(\log n)$.

Max-priority queue

Insert(A, *key*): Add (an element of) value *key* to A.

10.



Max-priority queue

Insert(A, *key*): Add (an element of) value *key* to A.

```
MAX-HEAP-INSERT(A, key)
A.heap-size = A.heap-size + 1
A[A.heap-size] = -∞
HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Max-priority queue

Insert(A, *key*): Add (an element of) value *key* to A.

```
MAX-HEAP-INSERT(A, key)
    A.heap-size = A.heap-size + 1
    A[A.heap-size] = −∞
    HEAP-INCREASE-KEY(A, A.heap-size, key)
```

RT:

Max-priority queue

Insert(A, key): Add (an element of) value *key* to A.

MAX-HEAP-INSERT(A, key)

$A.heap\text{-}size = A.heap\text{-}size + 1$
$A[A.heap\text{-}size] = -\infty$
HEAP-INCREASE-KEY($A, A.heap\text{-}size, key$)

RT: $O(\log n)$.