

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)

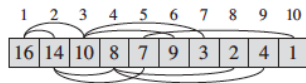
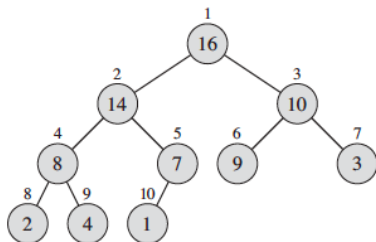
Outline

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

(Binary) Heap

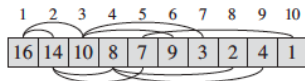
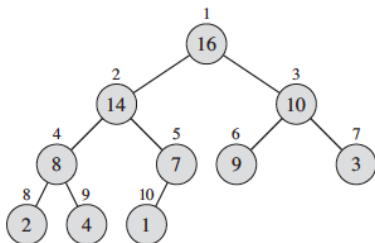
Heap A:

- ▶ '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

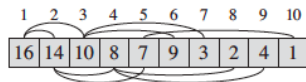
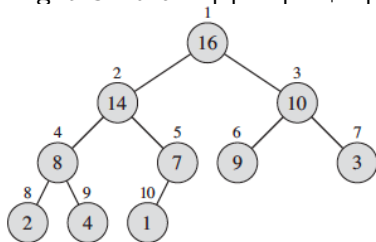
- ▶ Height of node: # of edges on a longest simple path from the node down to a leaf.
- ▶ Height of heap: height of root = $\Theta(\log n)$.
- ▶ Other notation:
 - ▶ $A.arraysize$: # of elements in array A .
 - ▶ $A.heapsize$: # of elements in heap A .
 - ▶ Always $A.heapsize \leq A.arraysize$.



Heap

Heap A:

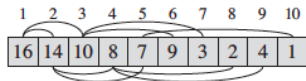
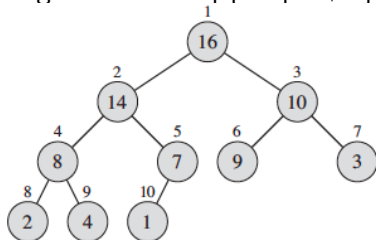
- ▶ Root: $A[1]$
- ▶ Parent of $A[i]$: $A[\lfloor i/2 \rfloor]$
- ▶ Left Child of $A[i]$: $A[2i]$.
- ▶ Right Child of $A[i]$: $A[2i + 1]$.



Heap

Heap A:

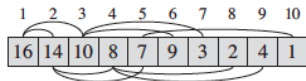
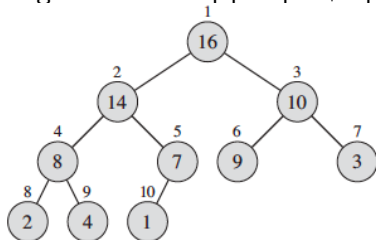
- ▶ Root: $A[1]$
- ▶ Parent of $A[i]$: $A[\lfloor i/2 \rfloor]$
- ▶ Left Child of $A[i]$: $A[2i]$.
- ▶ Right Child of $A[i]$: $A[2i + 1]$.



Heap

Heap A:

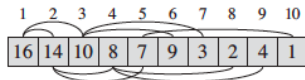
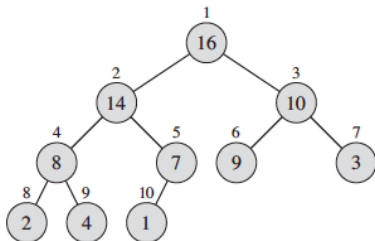
- ▶ Root: $A[1]$
- ▶ Parent of $A[i]$: $A[\lfloor i/2 \rfloor]$
- ▶ Left Child of $A[i]$: $A[2i]$.
- ▶ Right Child of $A[i]$: $A[2i + 1]$.



Question: How can we test if node i has a left child?

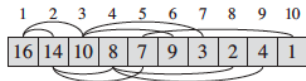
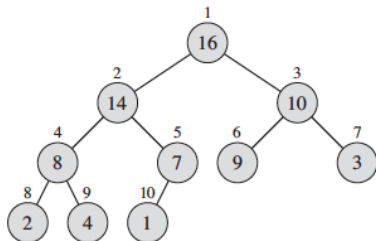
Max Heap

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



Max Heap

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



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, $A[\text{parent}(i)] \leq A[i]$.

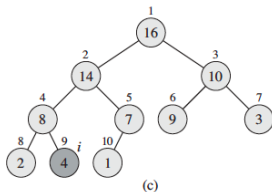
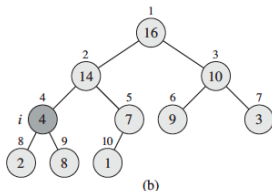
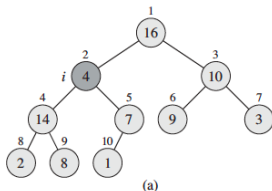
Min Heap

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

Building a (max-)heap

Helper function/procedure: Max-Heapify

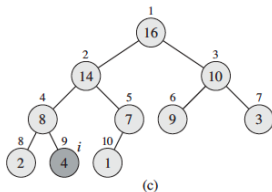
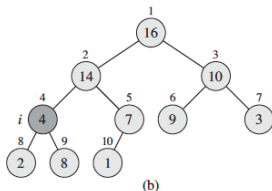
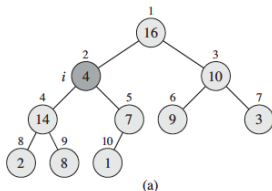
$\text{Max-Heapify}(i)$: If both the left and right subtrees of node i are max-heaps, make the subtree rooted at node i a max-heap.



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.

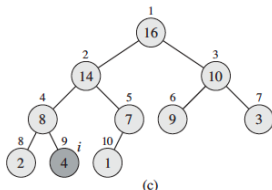
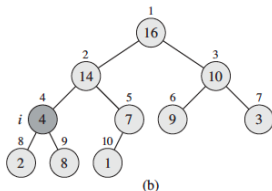
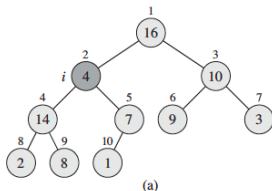


Correctness?

Building a (max-)heap

Helper function/procedure: Max-Heapify

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

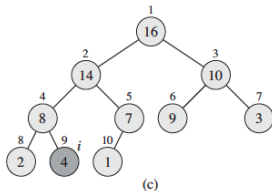
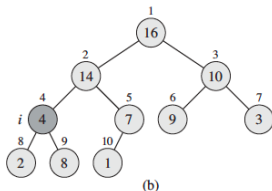
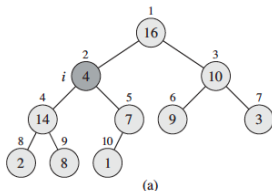


Correctness? Running time?

Building a (max-)heap

Helper function/procedure: Max-Heapify

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

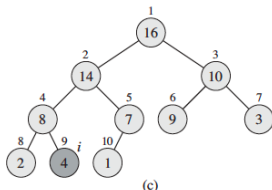
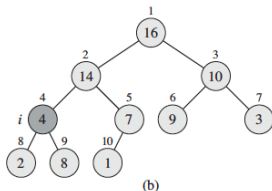
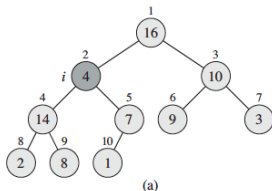


Correctness? Running time? $O(\lg n)$.

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.



Correctness? Running time? $O(\lg n)$. More precisely, $O(i$'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)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4       $largest = l$ 
5  else  $largest = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[largest]$ 
7       $largest = r$ 
8  if  $largest \neq i$ 
9      exchange  $A[i]$  with  $A[largest]$ 
10     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: $A[1\dots 10] = \{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$.

Building a (max-)heap

BUILD-MAX-HEAP(A)

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

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

Building a (max-)heap

BUILD-MAX-HEAP(A)

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

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

1. Initialization: Each of nodes $\lfloor n/2 \rfloor + 1, \dots, n$ 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 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

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)

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

Heapsort

HEAPSORT(A)

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

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

Priority Queue

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

Max-priority queue:

- ▶ $\text{Insert}(S, x)$: inserts x into S .
- ▶ $\text{Maximum}(S)$: returns element of S with largest key.
- ▶ $\text{Extract-Max}(S)$: removes and returns element of S with largest key.
- ▶ $\text{Increase-Key}(S, x, k)$: 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:

- ▶ $\text{Insert}(S, x)$: inserts x into S .
- ▶ $\text{Minimum}(S)$: returns element of S with smallest key.
- ▶ $\text{Extract-Min}(S)$: removes and returns element of S with smallest key.
- ▶ $\text{Decrease-Key}(S, x, k)$: 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:

- ▶ $\text{Insert}(S, x)$: inserts x into S .
- ▶ $\text{Minimum}(S)$: returns element of S with smallest key.
- ▶ $\text{Extract-Min}(S)$: removes and returns element of S with smallest key.
- ▶ $\text{Decrease-Key}(S, x, k)$: 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** $A[1]$

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)

```
1  if  $A.heap-size < 1$ 
2      error “heap underflow”
3   $max = A[1]$ 
4   $A[1] = A[A.heap-size]$ 
5   $A.heap-size = A.heap-size - 1$ 
6  MAX-HEAPIFY( $A, 1$ )
7  return  $max$ 
```

Max-priority queue

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

HEAP-EXTRACT-MAX(A)

```
1  if  $A.heap-size < 1$ 
2      error “heap underflow”
3   $max = A[1]$ 
4   $A[1] = A[A.heap-size]$ 
5   $A.heap-size = A.heap-size - 1$ 
6  MAX-HEAPIFY( $A, 1$ )
7  return  $max$ 
```

RT:

Max-priority queue

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

HEAP-EXTRACT-MAX(A)

```
1  if  $A.heap-size < 1$ 
2      error “heap underflow”
3   $max = A[1]$ 
4   $A[1] = A[A.heap-size]$ 
5   $A.heap-size = A.heap-size - 1$ 
6  MAX-HEAPIFY( $A, 1$ )
7  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)

```
1  if  $key < A[i]$ 
2      error “new key is smaller than current key”
3   $A[i] = key$ 
4  while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$ 
5      exchange  $A[i]$  with  $A[\text{PARENT}(i)]$ 
6       $i = \text{PARENT}(i)$ 
```

Max-priority queue

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

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

```
1  if  $key < A[i]$ 
2      error “new key is smaller than current key”
3   $A[i] = key$ 
4  while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$ 
5      exchange  $A[i]$  with  $A[\text{PARENT}(i)]$ 
6       $i = \text{PARENT}(i)$ 
```

RT:

Max-priority queue

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

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

```
1  if  $key < A[i]$ 
2      error “new key is smaller than current key”
3   $A[i] = key$ 
4  while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$ 
5      exchange  $A[i]$  with  $A[\text{PARENT}(i)]$ 
6       $i = \text{PARENT}(i)$ 
```

RT: $O(\log n)$.

Max-priority queue

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

Max-priority queue

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

MAX-HEAP-INSERT(A, key)

- 1 $A.heap-size = A.heap-size + 1$
- 2 $A[A.heap-size] = -\infty$
- 3 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)

1 $A.heap-size = A.heap-size + 1$

2 $A[A.heap-size] = -\infty$

3 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)

1 $A.heap-size = A.heap-size + 1$

2 $A[A.heap-size] = -\infty$

3 HEAP-INCREASE-KEY($A, A.heap-size, key$)

RT: $O(\log n)$.