Data Structures and Algorithms

Lecture 17

Aniket Basu Roy

BITS Pilani Goa Campus

2026-02-27 Fri

Agenda

Recap: Heaps

- A heap is a nearly complete binary tree stored as an array.
- Max-heap property: $A[\text{PARENT}(i)] \geq A[i]$ for all i.

MAX-HEAPIFY

The Problem

- $A[i]$ may be smaller than its children.
- But subtrees rooted at LEFT(i) and RIGHT(i) are valid max-heaps.

MAX-HEAPIFY "floats" $A[i]$ down to restore the max-heap property.

MAX-HEAPIFY

Pseudocode

```
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: Example

MAX-HEAPIFY$(A, 2)$:

```
        16
      /    \
    (4)     10
   /  \    /  \
  14   7  9    3
 / \  /
2   8 1
```

largest $=$ LEFT$(2) = 14$, since $A[4] = 14 > A[2] = 4 \Rightarrow$ swap

MAX-HEAPIFY: Example (contd.)

After swap, recurse MAX-HEAPIFY$(A, 4)$:

```
        16
      /    \
    14      10
   /  \    /  \
  (4)  7  9    3
 / \  /
2   8 1
```

$A[9] = 8 > A[4] = 4 \Rightarrow$ swap $A[4]$ and $A[9]$

MAX-HEAPIFY: Example (contd.)

Final:

```
        16
       /  \
     14     10
    /  \   /  \
   8    7 9    3
  / \  /
2 (4) 1
```

MAX-HEAPIFY: Time Complexity

- $O(1)$ work per level, recurse on a subtree.
- Subtree has at most $2n/3$ elements (worst case: bottom level half full).

Recurrence:

$$T(n) \leq T(2n/3) + \Theta(1)$$

By Master Theorem (Case 2):

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

BUILD-MAX-HEAP

Key Observation

Elements $A[\lfloor n/2 \rfloor + 1], \ldots, A[n]$ are leaves — trivially valid max-heaps.
Call MAX-HEAPIFY only on internal nodes, bottom to top.

Pseudocode

```
BUILD-MAX-HEAP(A, n)
1.   A.heap-size = n
2.   for i = floor(n/2) downto 1
3.       MAX-HEAPIFY(A, i)
```

BUILD-MAX-HEAP: Example

Input: $[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$

```
Initial tree:
            4
          /   \
         1     3
        / \   / \
       2  16 9  10
      / \ / \
     14 8 7
```

```
floor(10/2) = 5, call MAX-HEAPIFY on i=5,4,3,2,1
```

BUILD-MAX-HEAP: Example

```
Result:
            16
          /    \
        14      10
       /  \    /  \
      8    7  9    3
     / \  /
    2   4 1
```

BUILD-MAX-HEAP: Time Complexity

- Naive: n calls $\times$ $O(\lg n)$ each $= O(n \lg n)$ — not tight.

Tighter: nodes at height h cost $O(h)$; at most $\lceil n/2^{h+1} \rceil$ such nodes.

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

(Using $\displaystyle\sum_{h=0}^{\infty} hx^h = \frac{x}{(1-x)^2}$ with $x = 1/2$.)

BUILD-MAX-HEAP runs in $\Theta(n)$ time.

Summary

Operation	Time Complexity
MAX-HEAPIFY	$O(\lg n)$
BUILD-MAX-HEAP	$\Theta(n)$

Heapsort: The Idea

- Build a max-heap from the input array. $(\Theta(n))$

Heapsort: The Idea

- Build a max-heap from the input array. $(\Theta(n))$
- The maximum element is at $A[1]$.

Heapsort: The Idea

- Build a max-heap from the input array. $(\Theta(n))$
- The maximum element is at $A[1]$.
- Swap $A[1]$ with $A[n]$, shrink the heap by one, restore with MAX-HEAPIFY.

Heapsort: The Idea

- Build a max-heap from the input array. $(\Theta(n))$
- The maximum element is at $A[1]$.
- Swap $A[1]$ with $A[n]$, shrink the heap by one, restore with MAX-HEAPIFY.
- Repeat until the heap has size 1.

Heapsort: Pseudocode

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

Heapsort: Example Walkthrough

```
Input: [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]

After BUILD-MAX-HEAP:
  Heap:   [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

i=10: swap A[1]=16 <-> A[10]=1, heap-size=9
  Heap:   [14, 8, 10, 4, 7, 9, 3, 2, 1]
  Sorted: [16]

i=9: swap A[1]=14 <-> A[9]=1, heap-size=8
  Heap:   [10, 8, 9, 4, 7, 1, 3, 2]
  Sorted: [14, 16]
  ...
Final: [1, 2, 3, 4, 7, 8, 9, 10, 14, 16]
```

Correctness: Loop Invariant

At the start of each iteration with index i:

- $A[1:i]$ is a max-heap containing the i smallest elements of A.
- $A[i+1:n]$ contains the $n-i$ largest elements in sorted order.

Initialization

After BUILD-MAX-HEAP, $A[1:n]$ is a max-heap; $A[n+1:n]$ is empty. ✓

Maintenance

$A[1]$ is maximum of $A[1:i]$. Swap places it at $A[i]$. MAX-HEAPIFY restores the heap on $A[1:i-1]$. ✓

Termination

$i=1$: $A[2:n]$ sorted, $A[1]$ is the minimum. Array fully sorted. ✓

Time Complexity of Heapsort

Step	Cost	Invocations
BUILD-MAX-HEAP	$\Theta(n)$	1
MAX-HEAPIFY	$O(\lg n)$	$n - 1$
Total	$O(n \lg n)$	

Time Complexity of Heapsort

Step	Cost	Invocations
BUILD-MAX-HEAP	$\Theta(n)$	1
MAX-HEAPIFY	$O(\lg n)$	$n - 1$
Total	$O(n \lg n)$	

Lower Bound

Heapsort is comparison-based $\Rightarrow \Omega(n \lg n)$ worst case.

$\Rightarrow$ Heapsort runs in $\Theta(n \lg n)$ time.

Properties of Heapsort

- In-place: $O(1)$ extra space beyond the input array.
- Not stable: equal elements may not preserve original order.

Properties of Heapsort

- In-place: $O(1)$ extra space beyond the input array.
- Not stable: equal elements may not preserve original order.

Comparison

Algorithm	Time	Space	Stable?
Merge Sort	$\Theta(n \lg n)$	$O(n)$	Yes
Quicksort	$O(n \lg n)$ exp.	$O(\lg n)$	No
Heapsort	$\Theta(n \lg n)$	$O(1)$	No

Priority Queues

Definition

A max-priority queue supports:

Operation	Description
INSERT(S, x, k)	Insert x with key k into S
MAXIMUM(S)	Return element with max key
EXTRACT-MAX(S)	Remove & return element with max key
INCREASE-KEY(S, x, k)	Increase the key of element x to k

HEAP-MAXIMUM

```
HEAP-MAXIMUM(A)
1.  if A.heap-size < 1
2.      error "heap underflow"
3.  return A[1]
```

Time: $\Theta(1)$

HEAP-EXTRACT-MAX

```
HEAP-EXTRACT-MAX(A)
1.  max = HEAP-MAXIMUM(A)
2.  A[1] = A[A.heap-size]
3.  A.heap-size = A.heap-size - 1
4.  MAX-HEAPIFY(A, 1)
5.  return max
```

Time: $O(\lg n)$

HEAP-INCREASE-KEY

Increase key of x to k (require $k \geq x.key$). Bubble x upward.

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

Time: $O(\lg n)$ — at most height-many swaps.

MAX-HEAP-INSERT

```
MAX-HEAP-INSERT(A, key, n)
    if A.heap-size == n
        error "heap overflow"
    A.heap-size = A.heap-size + 1
    x = A[A.heap-size]
    x.key = -infinity
    HEAP-INCREASE-KEY(A, x, key)
```

Time: $O(\lg n)$

Summary: Priority Queue Operations

Operation	Time Complexity
HEAP-MAXIMUM	$\Theta(1)$
HEAP-EXTRACT-MAX	$O(\lg n)$
HEAP-INCREASE-KEY	$O(\lg n)$
MAX-HEAP-INSERT	$O(\lg n)$