# Data Structures and Algorithms

Lecture 17

Aniket Basu Roy

2026-02-27 Fri

## Contents

# 1 Agenda

## 1.1 Data Structures

## 1.2 Heapsort

## 1.3 Priority Queues

# 2 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 $i$.
- BUILD-MAX-HEAP runs in $\Theta(n)$ time.
- MAX-HEAPIFY runs in $O(\lg n)$ time.

# 3 Heapsort

## 3.1 The Idea

1. Build a max-heap from the input array. ($\Theta(n)$)
2. The maximum element is now at $A[1]$.
3. Swap $A[1]$ with $A[n]$, shrink the heap by one, and restore the heap property with MAX-HEAPIFY.
4. Repeat from step 2 until the heap has size 1.

### 3.2 Pseudocode

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

### 3.3 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]
  Sorted: []

Iteration i=10: swap A[1]=16 with A[10]=1, heap-size=9
  Heap:   [14, 8, 10, 4, 7, 9, 3, 2, 1]   (after MAX-HEAPIFY)
  Sorted: [16]

Iteration i=9: swap A[1]=14 with A[9]=1, heap-size=8
  Heap:   [10, 8, 9, 4, 7, 1, 3, 2]        (after MAX-HEAPIFY)
  Sorted: [14, 16]

...continuing...

Final sorted array: [1, 2, 3, 4, 7, 8, 9, 10, 14, 16]
```

## 4 Correctness of Heapsort

### 4.1 Loop Invariant

At the start of each iteration of the for-loop (line 2), the subarray $A[1:i]$ is a max-heap containing the $i$ smallest elements of $A$, and $A[i+1:n]$ contains the $n-i$ largest elements of $A$ in sorted order.

### 4.2 Initialization

After BUILD-MAX-HEAP, $A[1:n]$ is a max-heap. The subarray $A[n+1:n]$ is empty, so the invariant holds trivially.

### 4.3 Maintenance

At the start of an iteration with index $i$:

- $A[1]$ is the maximum of $A[1:i]$ (max-heap property).
- We swap $A[1]$ with $A[i]$, placing the \$i\$-th largest element at $A[i]$.
- Decreasing heap-size and calling MAX-HEAPIFY restores the max-heap on $A[1:i-1]$.
- The invariant holds for $i-1$.

### 4.4 Termination

The loop ends with $i = 1$. The invariant tells us $A[2{:}n]$ is sorted and contains the $n-1$ largest elements. Hence $A[1:n]$ is fully sorted.

## 5 Time Complexity of Heapsort

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

### 5.1 Lower Bound

Heapsort is comparison-based, so it requires $\Omega(n \lg n)$ comparisons in the worst case (by the decision tree lower bound).

### 5.2 $\Rightarrow$ Heapsort runs in $\Theta(n \lg n)$ time.

### 5.3 Properties

- **In-place**: uses only $O(1)$ extra space beyond the input array.
- **Not stable**: equal elements may not preserve their original order.
- Compare with Merge Sort ($\Theta(n \lg n)$, stable, $O(n)$ extra space) and Quicksort ($O(n \lg n)$ expected, in-place, not stable).

# 6 Priority Queues

## 6.1 Definition

A **max-priority queue** is an abstract data type that supports:

| Operation | Description |
|---|---|
| INSERT(S, x, k) | Insert element $x$ with key $k$ into set $S$ |
| MAXIMUM(S) | Return the element with the largest key |
| EXTRACT-MAX(S) | Remove and return the element with largest key |
| INCREASE-KEY(S, x, k) | Increase the key of element $x$ to $k$ |

## 6.2 HEAP-MAXIMUM

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

Time: $\Theta(1)$

## 6.3 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: $O(\lg n)$

## 6.4 HEAP-INCREASE-KEY

Increase the key of element $x$ to $k$ (we require $k \geq x.key$). After increasing the key, bubble $x$ upward until the max-heap property is restored.

```
HEAP-INCREASE-KEY(A, x, k)
1.  if k < x.key
2.      error "new key is smaller than current key"
3.  x.key = k
```

```
4.  while x != A[1] and x.key > x.parent.key
5.      exchange x with x.parent
6.      x = x.parent
```

Time: $O(\lg n)$ — at most the height of the heap swaps.

### 6.5 MAX-HEAP-INSERT

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

Time: $O(\lg n)$

### 6.6 Summary: Priority Queue Operations on a Max-Heap

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

## 7 Comparison of Sorting Algorithms

| Algorithm | Worst Case | Best Case | Average | Space | Stable? |
|---|---|---|---|---|---|
| Insertion Sort | $\Theta(n^2)$ | $\Theta(n)$ | $\Theta(n^2)$ | $O(1)$ | Yes |
| Merge Sort | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ | $O(n)$ | Yes |
| Quicksort | $\Theta(n^2)$ | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ | $O(\lg n)$ | No |
| Heapsort | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ | $O(1)$ | No |

## 8 Questions

1. Can Heapsort be made stable? What would be the cost?
2. Show that HEAP-EXTRACT-MAX maintains the max-heap property.
3. In HEAP-INCREASE-KEY, why do we start by setting the key to $-\infty$ in MAX-HEAP-INSERT?

4. A **d-ary heap** generalizes binary heaps: each node has $d$ children. What are the time complexities of MAX-HEAPIFY, BUILD-MAX-HEAP, and HEAP-EXTRACT-MAX for a d-ary heap?