Data Structures and Algorithms

Lecture 16

Aniket Basu Roy

2026-02-25 Wed

Contents

1 Agenda

1.1 Data Structures

1.2 Heaps

- Array representation
- Max-heap property
- MAX-HEAPIFY
- BUILD-MAX-HEAP

2 Motivation: Priority Queue

- We want a data structure that supports:
  - Insert a new element with a priority
  - Extract the element with the highest priority
- Examples: Job scheduling, Dijkstra's shortest path, event simulation
- A **heap** is the right data structure for this.

3 What is a Heap?

- A heap is a nearly complete binary tree stored compactly as an array.
- The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

```
Array: [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
Index:   1   2   3  4  5  6  7  8  9 10

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

- *A.length*: number of elements in the array
- *A.heap-size*: number of heap elements stored ($A.heap\text{-}size \leq A.length$)

4 Array Representation

For a node at index i:

Relation	Formula
Parent	$\lfloor i/2 \rfloor$
Left child	$2i$
Right child	$2i + 1$

```
PARENT(i)      return floor(i/2)
LEFT(i)        return 2i
RIGHT(i)       return 2i + 1
```

5 Max-Heap Property

5.1 Definition

A binary tree is a **max-heap** if for every node i other than the root:

$$A[\mathrm{PARENT}(i)] \geq A[i]$$

- The largest element is stored at the root.
- The subtree rooted at any node is itself a max-heap.

5.2 Min-Heap

A **min-heap** satisfies the symmetric property: $A[\text{PARENT}(i)] \leq A[i]$.

- Smallest element is at the root.
- Useful for implementing a min-priority queue.

6 Height of a Heap

- A heap of n elements has height $h = \lfloor \lg n \rfloor$.
- There are at most $\lceil n/2^{h+1} \rceil$ nodes of height h.
- In particular, at most $\lceil n/2 \rceil$ leaves (nodes of height 0).

7 MAX-HEAPIFY: Maintaining the Heap Property

7.1 The Problem

Suppose $A[i]$ might be smaller than its children, but the subtrees rooted at LEFT(i) and RIGHT(i) are already valid max-heaps.

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

7.2 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)
```

7.3 Example: MAX-HEAPIFY(A, 2)

```
Before:
             16
           /    \
         [4]     10
        /  \    /  \
       14   7  9    3
      / \  /
     2   8 1

Step 1: largest = LEFT(2) = 4, A[4]=14 > A[2]=4  => largest = 4
Step 2: A[5]=7 < A[4]=14, largest stays 4
Step 3: swap A[2] and A[4]

After swap:
             16
           /    \
         14      10
        /  \    /  \
       [4]  7  9    3
      / \  /
     2   8 1

Recurse: MAX-HEAPIFY(A, 4)
  largest = LEFT(4)=8, A[8]=2 < A[4]=4  => largest=4
  largest = RIGHT(4)=9, A[9]=8 > A[4]=4 => largest=9
  swap A[4] and A[9]

Final:
             16
           /    \
         14      10
        /  \    /  \
       8    7  9    3
      / \  /
     2  [4] 1
```

7.4 Time Complexity of MAX-HEAPIFY

- At each step, we do $O(1)$ work and recurse on a subtree.
- The 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): $T(n) = O(\lg n) = O(h)$

8 BUILD-MAX-HEAP: Building a Heap from Scratch

8.1 Key Observation

Elements $A[\lfloor n/2 \rfloor + 1], \ldots, A[n]$ are all leaves — they are trivially valid max-heaps. So we only need to call MAX-HEAPIFY on the internal nodes, from bottom to top.

8.2 Pseudocode

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

8.3 Example

```
Input array: [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]
Index:         1  2  3  4   5  6   7   8  9 10

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
```

```
i=5: A[5]=16 > children (none visible at bottom) => no change
i=4: A[4]=2, children: A[8]=14, A[9]=8 => swap with 14
i=3: A[3]=3, children: A[6]=9, A[7]=10 => swap with 10
i=2: A[2]=1, children: A[4]=14, A[5]=16 => swap with 16, then recurse
i=1: A[1]=4, children: A[2]=16, A[3]=10 => swap with 16, then recurse

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

8.4 Time Complexity of BUILD-MAX-HEAP

- Naive analysis: n calls to MAX-HEAPIFY each costing $O(\lg n)$ gives $O(n \lg n)$.
- But this is **not tight**. Nodes at greater heights are fewer.

Tighter analysis: nodes at height h cost $O(h)$ and there are at most $\lceil n/2^{h+1} \rceil$ of them.

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

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

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

9 Summary

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

10 Questions

1. In MAX-HEAPIFY, what is the maximum number of comparisons performed?
2. Why do we iterate from $\lfloor n/2 \rfloor$ downto 1 in BUILD-MAX-HEAP and not from 1 to $\lfloor n/2 \rfloor$?
3. Why does the naive analysis of BUILD-MAX-HEAP give $O(n \lg n)$ while the tight analysis gives $O(n)$?