Why Building a Binary Heap Takes Linear Time

Manish Acharya

1 Introduction

Binary heaps are among the most fundamental data structures in algorithms, forming the basis of priority queues and heap sort. A binary heap can be constructed from an unordered array using a standard bottom-up procedure often referred to as *build-heap*.

At first glance, this procedure appears to require $O(n \log n)$ time, since it repeatedly applies a subroutine that may traverse a path of logarithmic length. Surprisingly, this intuition is incorrect. In this note, we show that building a binary heap from an unordered array runs in *linear time*.

Our goal is to explain this result using a precise structural analysis of binary heaps and a careful accounting of the work performed at nodes of different heights.

2 Binary Heaps and the Heap Property

A binary heap is a complete binary tree whose nodes satisfy the *heap-order property*: each node's key is ordered with respect to the keys of its children (either greater than or less than, depending on convention).

The shape of the tree is completely determined by the number of elements. All levels except possibly the last are full, and the last level is filled from left to right. This structural property plays a crucial role in the running-time analysis.

3 Heapify-Down Operation

We consider the following primitive operation:

> HEAPIFY-DOWN(v): Given a node v whose children already satisfy the heap-order property, restore the heap-order property in the subtree rooted at v by repeatedly swapping the key at v with the appropriate child until the property is restored.

This operation is the core subroutine used in heap construction. Its behavior depends on how far the key at v must move downward to reach a valid position.

4 The Build-Heap Procedure

The build-heap algorithm constructs a heap by applying HEAPIFY-DOWN to nodes in reverse level order:

- All leaves trivially satisfy the heap-order property.
- Starting from the last internal node and moving upward to the root, apply HEAPIFY-DOWN to each node.

By the time HEAPIFY-DOWN is applied to a node, both of its subtrees already satisfy the heap property.

5 Why a Uniform $O(\log n)$ Bound Is Not Tight

A natural first estimate assigns a uniform $O(\log n)$ cost to each application of HEAPIFY-DOWN, based on the fact that the height of a binary heap with n elements is $O(\log n)$. Since the build-heap procedure applies this operation to $\Theta(n)$ nodes, this reasoning yields an $O(n \log n)$ upper bound.

While this bound is correct as a worst-case estimate, it fails to capture the actual distribution of work performed by the algorithm. The cost of HEAPIFY-DOWN depends on the height of the node on which it is invoked, not on the height of the heap as a whole. Most nodes in a binary heap lie close to the leaves and therefore have very small height. As a result, assigning a logarithmic cost uniformly to all nodes significantly overestimates the total running time.

6 Height-Based Cost Analysis

6.1 Height of a Node

The *height* of a node is defined as the number of edges on the longest downward path from that node to a leaf. Leaves therefore have height 0. Since a binary heap with n elements forms a complete binary tree, the height of the root is $\lfloor \log n \rfloor$, and no node has height exceeding this value.

6.2 Cost of Heapify-Down

The running time of HEAPIFY-DOWN is governed by how far a key may need to move downward in the tree. When applied to a node of height h, the operation compares the key at that node with its children and, if necessary, swaps it with one of them. This process may repeat as the key moves downward toward a leaf.

Because the key can move down at most one level per iteration, and cannot move upward, it traverses at most h edges. Each level involves a constant amount of work, consisting of a fixed number of comparisons and at most one swap. Consequently, the total running time of HEAPIFY-DOWN on a node of height h is $O(h)$.

7 Counting Nodes by Height

To establish the linear-time complexity of building a heap, we now bound the distribution of nodes relative to their height.

Lemma 1. *In a binary heap containing n nodes, the number of nodes at height exactly h is at most* $\left\lceil \frac{n}{2^{h+1}} \right\rceil$.

Proof. We utilize the standard array-based representation where nodes are indexed $1, 2, \ldots, n$. For any node at index i, its children (if they exist) are located at indices $2i$ and $2i + 1$.

A node at index i has height at least h only if it possesses a descendant at least h levels below it. The index of the leftmost descendant h levels below node i is $2^h i$. Therefore, for a node to have height at least h, it must satisfy the condition

$$2^h i \leq n \quad \Longrightarrow \quad i \leq \frac{n}{2^h}.$$

The number of such indices i is exactly $\lfloor n/2^h \rfloor$.

To find the number of nodes with height *exactly* h, we subtract the number of nodes with height at least $h + 1$ from the number of nodes with height at least h:

$$N(h) = \left\lfloor \frac{n}{2^h} \right\rfloor - \left\lfloor \frac{n}{2^{h+1}} \right\rfloor.$$

Using the inequality $\lfloor 2x \rfloor - \lfloor x \rfloor \leq \lceil x \rceil$ for all real x, and substituting $x = n/2^{h+1}$, we obtain

$$N(h) \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil.$$

This confirms that the number of nodes at height h decreases exponentially as the height increases, which is the foundational observation for the $O(n)$ running-time bound. □

8 Aggregate Running Time

We now compute the total running time of the build-heap procedure by aggregating the cost of HEAPIFY-DOWN over all nodes, grouped according to their height. Let $T(n)$ denote the total running time on a heap with n nodes.

From the height-based analysis, applying HEAPIFY-DOWN to a node of height h costs $O(h)$. Since nodes of the same height incur comparable costs, it is natural to sum the work by height:

$$T(n) \leq \sum_{h=0}^{\lfloor \log n \rfloor} (\text{number of nodes of height } h) \cdot O(h).$$

Using the bound from the previous lemma on the number of nodes at each height, we obtain

$$T(n) \leq \sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^{h+1}} \cdot O(h).$$

Factoring out the common factor of n, this expression simplifies to

$$T(n) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right).$$

The summation $\sum_{h=0}^{\infty} \frac{h}{2^h}$ converges to a constant, since the exponential decay of 2^{-h} dominates the linear growth of h. Truncating the sum at $\lfloor \log n \rfloor$ therefore does not affect the asymptotic bound.

We conclude that

$$T(n) = O(n).$$

9 Intuition Behind the Linear Bound

Although HEAPIFY-DOWN can be expensive when applied near the root, only a small number of nodes have large height. Most nodes lie close to the leaves and therefore incur little or no cost. The total running time is thus dominated by the large number of inexpensive operations near the bottom of the heap, rather than by the few expensive operations near the top.

This phenomenon illustrates a general principle in algorithm analysis: understanding how computational costs are distributed across an input can lead to significantly tighter bounds than uniform worst-case reasoning.

10 Conclusion

We have shown that building a binary heap from an unordered array takes linear time. The proof relies on analyzing the cost of heapify-down as a function of node height and aggregating these costs across the heap.

This result holds regardless of whether the heap-order property is defined using maximum or minimum keys, since the analysis depends solely on the structural properties of complete binary trees.

References

[CLRS09] T. H. Cormen et al., *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.

[KT06] J. Kleinberg and É. Tardos, *Algorithm Design*, Pearson, 2006.

[SW11] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.