Heap Sort

Varun Kumar

July 5, 2025

Constructing Max Heap: Insertion Method vs Build-Heap Method

1. Key Idea: Insertion Method

- Start with an empty heap.
- Insert one element at a time.
- After each insertion, perform **up-heap (bubble up)** to maintain heap property.

2. Key Idea: Build-Heap Method

- Start with all elements in array form.
- Treat it as a complete binary tree.
- Apply **heapify (down-heap)** from the last non-leaf node up to the root.

Comparisons and Swaps:

1. Insertion Method

- For each insertion, worst-case comparisons = height of tree = $\log i$
- Total comparisons: $O(n \log n)$
- Each insertion may involve several swaps.

2. Build-Heap Method

- Heapify from $\left[\frac{n}{2} - 1\right]$ to 0.
- Comparisons are fewer near the top.
- Total comparisons: $O(n)$
- Much more efficient than repeated insertion.

Optimal Behavior

- **Insertion method:** Intuitive but inefficient for large arrays.
- **Build-heap method:** Optimal and used in Heap Sort.
- For n elements, build-heap runs in $O(n)$ while insertions take $O(n \log n)$.

Pseudocode: Insertion Method

```
insert(heap, value):
    heap.append(value)
    i = len(heap) - 1
    while i > 0:
        parent = (i - 1) // 2
        if heap[i] > heap[parent]:
            swap(heap[i], heap[parent])
            i = parent
        else:
            break
```

Pseudocode: Build-Heap Method

```
buildHeap(arr, n):
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)

heapify(arr, n, i):
    largest = i
    left = 2*i + 1
    right = 2*i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        swap(arr[i], arr[largest])
        heapify(arr, n, largest)
```

Example: Build Max Heap using Insertion Method

Input: Insert elements one-by-one from: $[3, 5, 1, 10, 2, 7, 6, 4]$

1. Insert 3 $\rightarrow$ No parent to compare

$$[3]$$

2. Insert 5

$$[3, 5] \rightarrow 5 \text{ ¿ } 3 \Rightarrow \text{Swap} \Rightarrow [5, 3]$$

3. Insert 1

$$[5, 3, 1] \rightarrow 1 \text{ ¡ } 5 \Rightarrow \text{No change}$$

4. Insert 10

$$[5, 3, 1, 10] \rightarrow 10 \text{ ¿ } 3 \Rightarrow \text{Swap} \Rightarrow [5, 10, 1, 3]$$

$$10 \text{ ¿ } 5 \Rightarrow \text{Swap} \Rightarrow [10, 5, 1, 3]$$

5. Insert 2

$$[10, 5, 1, 3, 2] \rightarrow 2 \text{ ¡ } 5 \Rightarrow \text{No change}$$

6. Insert 7

$$[10, 5, 1, 3, 2, 7] \rightarrow 7 \text{ ¿ } 1 \Rightarrow \text{Swap} \Rightarrow [10, 5, 7, 3, 2, 1]$$

7. Insert 6

$$[10, 5, 7, 3, 2, 1, 6] \rightarrow 6 \text{ ¡ } 7 \Rightarrow \text{No change}$$

8. Insert 4

$$[10, 5, 7, 3, 2, 1, 6, 4] \rightarrow 4 \text{ ¿ } 3 \Rightarrow \text{Swap} \Rightarrow [10, 5, 7, 4, 2, 1, 6, 3]$$

Final Max Heap: $[10, 5, 7, 4, 2, 1, 6, 3]$

Example: Build Max Heap using Heapify

Input: $[3, 5, 1, 10, 2, 7, 6, 4]$, $n = 8$

1. Start heapifying from $i = \lfloor n/2 \rfloor - 1 = 3$
2. **heapify(3):**

   $$\text{Node: 10, Left: 4, Right: None} \Rightarrow \text{No change}$$

3. **heapify(2):**

   $$\text{Node: 1, Left: 7, Right: 6} \Rightarrow \text{7 ¿ 1} \Rightarrow \text{Swap 1 and 7}$$

   New array: $[3, 5, 7, 10, 2, 1, 6, 4]$
4. **heapify(1):**

   $$\text{Node: 5, Left: 10, Right: 2} \Rightarrow \text{10 ¿ 5} \Rightarrow \text{Swap 5 and 10}$$

   New array: $[3, 10, 7, 5, 2, 1, 6, 4]$
5. **heapify(3):**

   $$\text{Node: 5, Left: 4, Right: None} \Rightarrow \text{No change}$$

6. **heapify(0):**

   $$\text{Node: 3, Left: 10, Right: 7} \Rightarrow \text{10 ¿ 3} \Rightarrow \text{Swap 3 and 10}$$

   New array: $[10, 3, 7, 5, 2, 1, 6, 4]$
7. **heapify(1):**

   $$\text{Node: 3, Left: 5, Right: 2} \Rightarrow \text{5 ¿ 3} \Rightarrow \text{Swap 3 and 5}$$

   New array: $[10, 5, 7, 3, 2, 1, 6, 4]$
8. **heapify(3):**

   $$\text{Node: 3, Left: 4, Right: None} \Rightarrow \text{4 ¿ 3} \Rightarrow \text{Swap}$$

   Final array: $[10, 5, 7, 4, 2, 1, 6, 3]$

   Final Max Heap: $[10, 5, 7, 4, 2, 1, 6, 3]$

Python code for Max-Heap Construction using Heapify from Middle to Root

```
def heapify(arr, n, i):
    largest = i           # Assume current index is largest
    left = 2 * i + 1      # Left child index
    right = 2 * i + 2     # Right child index

    # Check if left child exists and is greater than current largest
    if left < n and arr[left] > arr[largest]:
        largest = left

    # Check if right child exists and is greater than current largest
    if right < n and arr[right] > arr[largest]:
        largest = right

    # If largest is not the current index, swap and continue heapifying
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def build_max_heap(arr):
    n = len(arr)
    # Start from last non-leaf node and move up to root
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)

# Input array
arr = [3, 5, 1, 10, 2, 7, 6, 4]
print("Before-Build-Heap:", arr)
build_max_heap(arr)
print("After␣Build-Heap:", arr)
```

Is Build-Heap Method Stable? If No, then why?

Answer: No, the Build-Heap Method is **not stable**.

- It uses `heapify()` which swaps elements without checking their original position.
- On equal values, it may move the element that appeared earlier in the input to a lower position in the heap.
- This violates the principle of stability: *"equal elements retain their original order"*.

Example of Instability in Build-Heap

Input with tagged equal elements:

```
[(4a), (3), (4b)]
```

- Initial positions: `4a` before `4b`
- When heapify is called at index 0:
  - Children: `4b` and `3`
  - Heapify may choose `4b` (right child) over `4a` (root)
  - After swap: `[(4b), (3), (4a)]`
- Now `4b` appears before `4a` — original order broken.

Hence, Build-Heap is not stable.

Why Heap Sort is Unstable

Heap Sort is **not stable** because it may change the relative order of equal elements.

Unstable Operation: `swap()` in `heapify()`

Root Cause of Instability

- During the `heapify()` process in `buildHeap()`, nodes are compared and swapped.
- If two elements have the **same value**, their **original order can be reversed** by swapping.
- This violates the definition of a stable sort.

Example: Loss of Stability

Consider the input array with values and tags:

$$[(4, \mathrm{A}),\ (3, \mathrm{B}),\ (4, \mathrm{C})]$$

- Both elements A and C have value 4.
- After applying heapify, the element `(4, C)` may be moved above `(4, A)`.
- This reverses their original order and makes the sort unstable.

How to Make Build-Heap Stable? Stable Heap Suggestion

To preserve stability:

- During comparisons, compare tuples: `(value, original_index)`
- If two elements have the same value, prefer the one with the **smaller original index**.
- This avoids swapping equal elements out of order.

Thus, standard `buildHeap()` is unstable due to arbitrary swaps of equal values. To fix this, we must track and respect original positions.

Modified Comparison Example

```
[(4, 0), (3, 1), (4, 2)]
```

- Compare `(4, 0)` and `(4, 2)`:
  - Values equal: $4 = 4$
  - Use index: $0 < 2 \rightarrow$ keep `(4, 0)` above
- Thus, original order is preserved.

Note: This requires more memory (to store index) and slightly slower comparisons, but gives **stability**.

Python code for Max-Heap Construction using Insertion Method

```
def insert_max_heap(heap, value):
    heap.append(value)  # Add new value at the end
    i = len(heap) - 1   # Index of inserted value

    # Bubble up (up-heap) to maintain max-heap property
    while i > 0:
        parent = (i - 1) // 2
        if heap[i] > heap[parent]:
            # Swap if child is greater than parent
            heap[i], heap[parent] = heap[parent], heap[i]
            i = parent
        else:
            break

# Input array
arr = [3, 5, 1, 10, 2, 7, 6, 4]
heap = []

print("Step-by-step␣insertion␣into␣Max-Heap:")
for val in arr:
    insert_max_heap(heap, val)
    print(heap)
```

Is Insertion based Heap Sort Stable? If no, then why?

Answer: No, Heap Sort is **not stable**.

- **Heapify** and **swap** operations reorder elements based on value only.
- They do **not preserve the original order** of equal elements.
- This violates the condition of stability.

Example Demonstrating Instability

Assume elements have labels to distinguish duplicates:

- Input array: `[(5a), 4, (5b), 3]`
- Note: `5a` and `5b` have equal values but different initial positions.

Step: Build Max Heap

- Heapify at index 1: no change
- Heapify at index 0:
  - Compares `5a` (index 0) and `5b` (index 2)
  - May pick `5b` as root (due to implementation order)

Resulting Heap: `[(5b), 4, (5a), 3]`
Conclusion: Relative order of equal elements `5a`, `5b` is changed.

Which Operation Causes Instability?

heapify():

- Selects the largest among parent, left, right — no regard for original position.
- On tie (equal values), any child may be chosen.
- This leads to non-stable reordering.

Can Heap Sort be Made Stable?

- Yes, by storing a tuple `(value, original_index)`.
- Modify comparisons to break ties using index.
- But this is not standard Heap Sort anymore.

Time Complexity of Heap Sort

- **Best Case:** $O(n \log n)$
  - Even in the best scenario, Heap Sort does not benefit from partial ordering.
  - Every element still needs to be heapified and extracted.
- **Average Case:** $O(n \log n)$
  - On average, heap construction takes $O(n)$ and each of the n extractions takes $O(\log n)$.
- **Worst Case:** $O(n \log n)$
  - In the worst case, all heapify operations go to the bottom of the tree.
  - Each delete-max operation takes $O(\log n)$.

Space Complexity

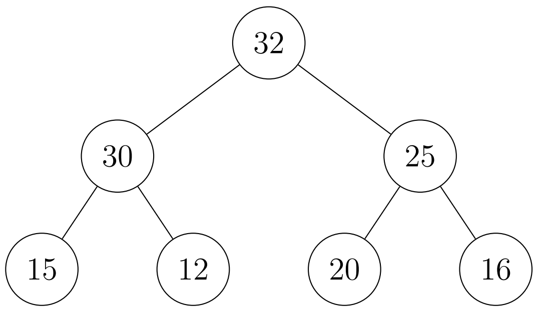
- **Auxiliary Space:** $O(1)$ (in-place sorting)

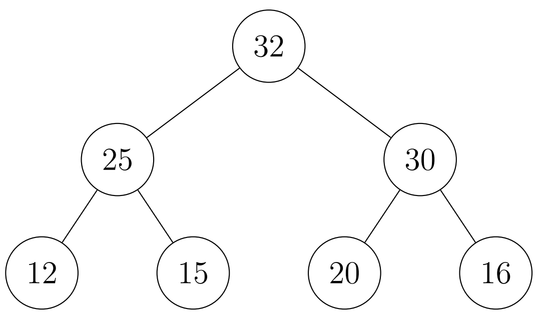
Heap Sort Summary Table

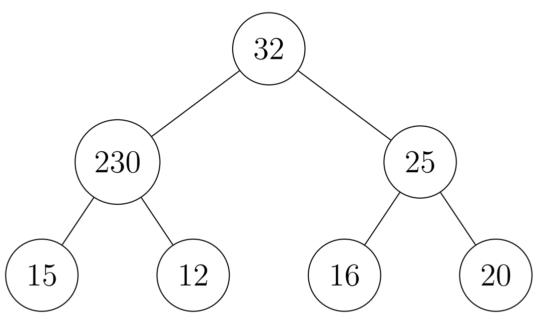
Case	**Comparisons**	**Swaps**	**Time**	**Adaptive**	**Stable**
Best Case	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No	No
Average Case	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No	No
Worst Case	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No	No

1 GATE CSE 2004

The elements $32, 15, 20, 30, 12, 25, 16$ are inserted one by one in the given order into a maxHeap. The resultant maxheap is

(a)


(b)


(c)


(d)