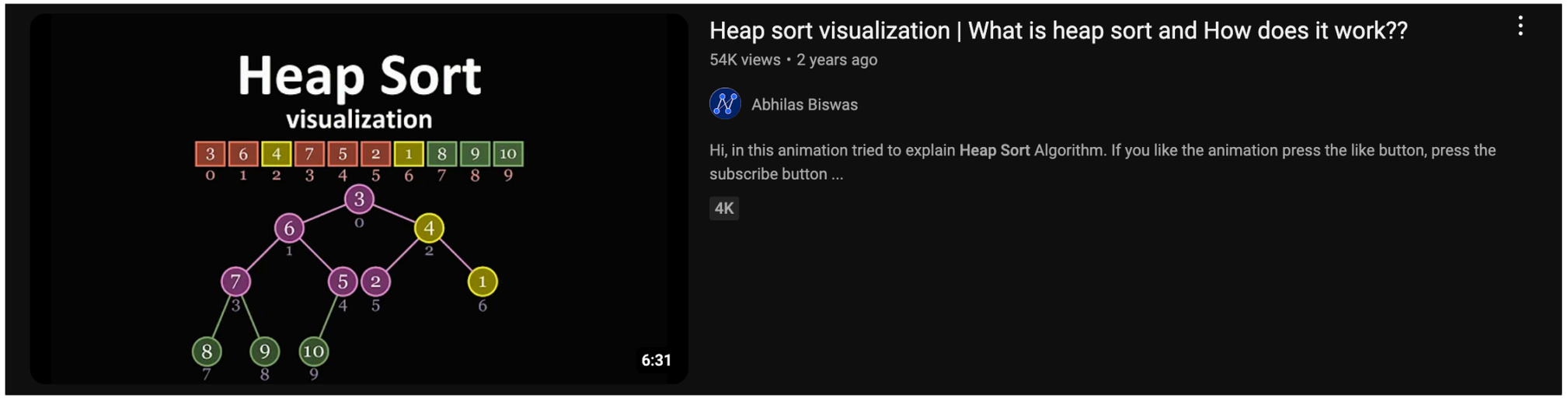
INTRODUCTION TO

HEAP SORT

RAJENDRAN S

HEAP SORT

UNDERSTAND HOW HEAP SORT WORKS!



Click here to redirect

What are we gonna cover here?

Here you go,

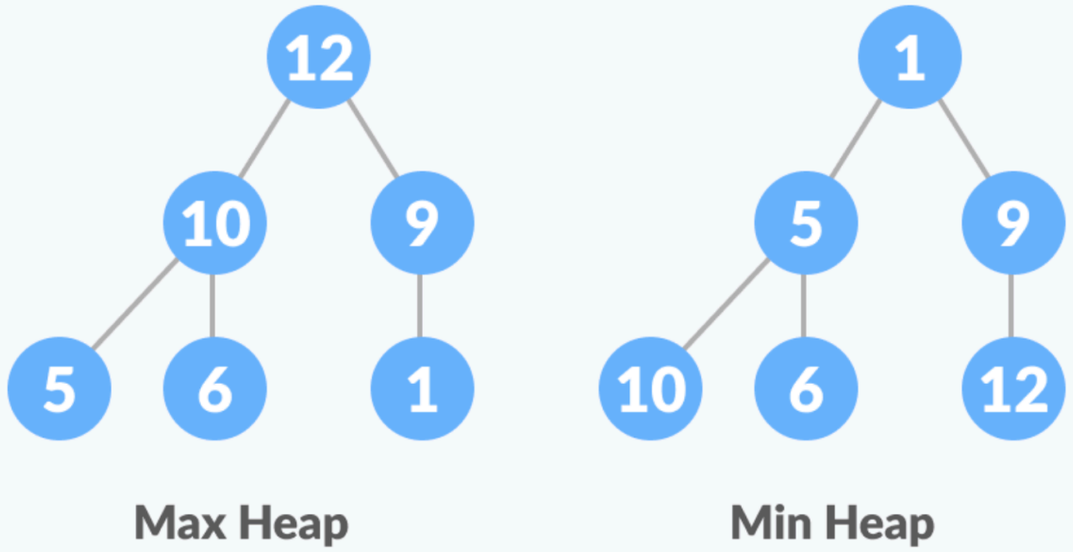
- What is heap DS? Binary Heap DS?
- How does heap sort work?
- What advantages do we have using it?
- Algorithm for HeapSort
- Heap Sort Complexities
- Applications of Heap Sort

What is Heap Data Structure?

Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if

- it is a complete binary tree
- All nodes in the tree follow the property that they are greater than their children i.e. the largest element is at the root and both its children and smaller than the root and so on. Such a heap is called a max-heap. If instead, all nodes are smaller than their children, it is called a min-heap

The following example diagram shows Max-Heap and Min-Heap.

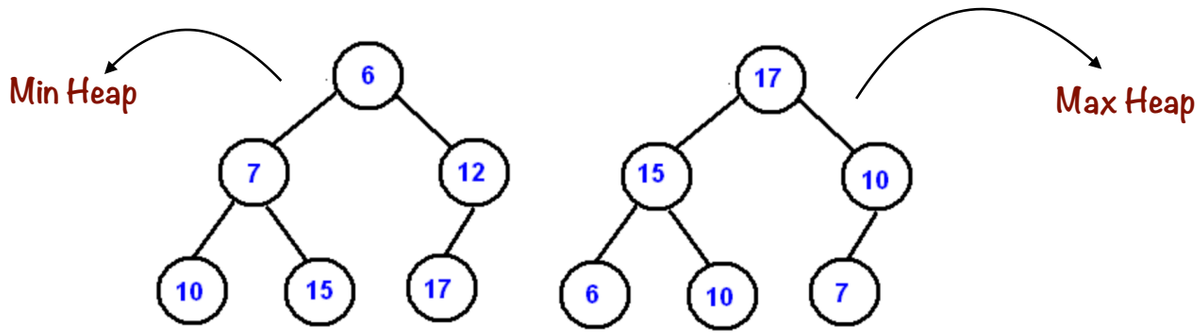


Binary Heaps

Introduction

A binary heap is a complete binary tree which satisfies the heap ordering property. The ordering can be one of two types:

- the *min-heap property*: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.
- the *max-heap property*: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.



In a heap the highest (or lowest) priority element is always stored at the root, hence the name "heap". A heap is not a sorted structure and can be regarded as partially ordered. As you see from the picture, there is no particular relationship among nodes on any given level, even among the siblings.

Since a heap is a complete binary tree, it has a smallest possible height - a heap with N nodes always has O(log N) height.

A heap is useful data structure when you need to remove the object with the highest (or lowest) priority. A common use of a heap is to implement a priority queue.

Overview of Heapsort

- **Data Structure Used**: Heapsort uses a *binary heap*, which is a complete binary tree with the property that each parent node is either greater than or equal to (in a max heap) or less than or equal to (in a min heap) its child nodes.
- **Time Complexity**: $O(n \log n)$ in both the average and worst cases.
- **Space Complexity**: $O(1)$ auxiliary space, as it sorts in place.
- **Stable**: No, heapsort is not a stable sorting algorithm (since equal elements might not preserve their order).

Working of Heap Sort

1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
2. **Swap:** Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
3. **Remove:** Reduce the size of the heap by 1.
4. **Heapify:** Heapify the root element again so that we have the highest element at root.
5. The process is repeated until all the items of the list are sorted.

Complexity Analysis

1. **Time Complexity**:
   - *Heap Construction*: Building the heap takes $O(n)$ time because heapifying each node in a bottom-up manner is efficient.
   - *Extraction and Sorting*: Each extraction and heapify operation takes $O(\log n)$, and there are n extractions, so this phase takes $O(n \log n)$.
   - Overall time complexity: $O(n + n \log n) = O(n \log n)$.
2. **Space Complexity**: $O(1)$, as heapsort sorts the array in place without needing extra storage.

6 Steps of a Heap Sort Algorithm

1. Transform the array into a binary tree by inserting each element as a node in a breadth-first manner.
2. Convert the binary tree into a max heap, ensuring that all parent nodes are greater than or equal to their child nodes.
3. Swap the root node — the largest element — with the last element in the heap.
4. Call the `heapify()` function to restore the max heap property.
5. Repeat steps 3 and 4 until the heap is sorted, and exclude the last element from the heap on each iteration.
6. After each swap and `heapify()` call, ensure that the max heap property is satisfied.

```c

// Function to maintain the max heap property for the heap rooted at index i
void heapify(int array[], int n, int i) {
    int largest = i;            // Initialize largest as root
    int left = 2 * i + 1;       // Left child index
    int right = 2 * i + 2;      // Right child index

    // If left child is larger than root
    if (left < n && array[left] > array[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < n && array[right] > array[largest])
        largest = right;

    // If largest is not root
    if (largest != i) {
        swap(&array[i], &array[largest]);   // Swap root with largest
        heapify(array, n, largest);         // Recursively heapify the affected subtree
    }
}

// Main heapsort function
void heapsort(int array[], int n) {
    // Step 1: Build a max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(array, n, i);

    // Step 2: Extract elements one by one from the heap
    for (int i = n - 1; i >= 0; i--) {
        // Move current root to end (max element to sorted position)
        swap(&array[0], &array[i]);

        // Call heapify on the reduced heap
        heapify(array, i, 0);
    }
}
```

Advantages of Heap Sort

- **Efficient Time Complexity:** Heap Sort has a time complexity of O(n log n) in all cases. This makes it efficient for sorting large datasets. The **log n** factor comes from the height of the binary heap, and it ensures that the algorithm maintains good performance even with a large number of elements.
- **Memory Usage:** Memory usage can be minimal (by writing an iterative heapify() instead of a recursive one). So apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work
- **Simplicity:** It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion.

Disadvantages of Heap Sort

- **Costly**: Heap sort is costly as the constants are higher compared to merge sort even if the time complexity is O(n Log n) for both.
- **Unstable**: Heap sort is unstable. It might rearrange the relative order.
- **Inefficient:** Heap Sort is not very efficient because of the high constants in the time complexity.

Heap Sort Complexity

Time Complexity	
Best	O(nlog n)
Worst	O(nlog n)
Average	O(nlog n)
Space Complexity	O(1)
Stability	No

Heap Sort Applications

Systems concerned with security and embedded systems such as Linux Kernel use Heap Sort because of the `O(n log n)` upper bound on Heapsort's running time and constant `O(1)` upper bound on its auxiliary storage.

Although Heap Sort has `O(n log n)` time complexity even for the worst case, it doesn't have more applications (compared to other sorting algorithms like Quick Sort, Merge Sort). However, its underlying data structure, heap, can be efficiently used if we want to extract the smallest (or largest) from the list of items without the overhead of keeping the remaining items in the sorted order. For e.g Priority Queues.

Heap Sort has `O(nlog n)` time complexities for all the cases (best case, average case, and worst case).

Let us understand the reason why. The height of a complete binary tree containing n elements is `log n`

As we have seen earlier, to fully heapify an element whose subtrees are already max-heaps, we need to keep comparing the element with its left and right children and pushing it downwards until it reaches a point where both its children are smaller than it.

In the worst case scenario, we will need to move an element from the root to the leaf node making a multiple of `log(n)` comparisons and swaps.

During the build_max_heap stage, we do that for `n/2` elements so the worst case complexity of the build_heap step is `n/2*log n ~ nlog n`.

During the sorting step, we exchange the root element with the last element and heapify the root element. For each element, this again takes `log n` worst time because we might have to bring the element all the way from the root to the leaf. Since we repeat this `n` times, the heap_sort step is also `nlog n`.

Also since the `build_max_heap` and `heap_sort` steps are executed one after another, the algorithmic complexity is not multiplied and it remains in the order of `nlog n`.

Also it performs sorting in `O(1)` space complexity. Compared with Quick Sort, it has a better worst case `( O(nlog n) )`. Quick Sort has complexity `O(n^2)` for worst case. But in other cases, Quick Sort is fast. Introsort is an alternative to heapsort that combines quicksort and heapsort to retain advantages of both: worst case speed of heapsort and average speed of quicksort.

“Read this blog to fall in ❤️ with heap sort”

Click here to read

And then go to this, Click here to read