Heap Sort - GeeksforGeeks

Source: https://www.geeksforgeeks.org/heap-sort/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Heap Sort Last Updated : 5 Feb, 2026 Heap Sort is a comparison-based sorting algorithm based on the Binary Heap data structure. It is an optimized version of selection sort. The algorithm repeatedly finds the maximum (or minimum) element and swaps it with the last (or first) element. Using a binary heap allows efficient access to the max (or min) element in O(log n) time instead of O(n). The process is repeated for the remaining elements until the array is sorted. Overall, Heap Sort achieves a time complexity of O(n log n). Heap Sort Algorithm : First convert the array into a max heap using heapify, Please note that this happens in-place. The array elements are re-arranged to follow heap properties. Then one by one delete the root node of the Max-heap and replace it with the last node and heapify. Repeat this process while size of heap is greater than 1. Detailed Working of Heap Sort Step 1: Treat the Array as a Complete Binary Tree We first need to visualize the array as a complete binary tree. For an array of size n, the root is at index 0, the left child of an element at index i is at 2i + 1, and the right child is at 2i + 2. Step 2: Build a Max Heap Step 3: Sort the array by placing largest element at end of unsorted array. In the illustration above, we have shown some steps to sort the array. We need to keep repeating these steps until there's only one element left in the heap. Try it on GfG Practice C++ #include <iostream> #include <vector> using namespace std ; // To heapify a subtree rooted with node i void heapify (vector < int >& arr , int n , int i){ // Initialize largest as root int largest = i ; // left index = 2*i + 1 int l = 2 * i + 1 ; // right index = 2*i + 2 int r = 2 * i + 2 ; // If left child is larger than root if (l < n && arr [l] > arr [largest]) largest = l ; // If right child is larger than largest so far if (r < n && arr [r] > arr [largest]) largest = r ; // If largest is not root if (largest != i) { swap (arr [i], arr [largest]); // Recursively heapify the affected sub-tree heapify (arr , n , largest); } } // Main function to do heap sort void heapSort (vector < int >& arr){ int n = arr . size (); // Build heap (rearrange vector) for (int i = n / 2 - 1 ; i >= 0 ; i --) heapify (arr , n , i); // One by one extract an element from heap for (int i = n - 1 ; i > 0 ; i --) { // Move current root to end swap (arr [0], arr [i]); // Call max heapify on the reduced heap heapify (arr , i , 0); } } int main (){ vector < int > arr = { 9 , 4 , 3 , 8 , 10 , 2 , 5 }; heapSort (arr); for (int i = 0 ; i < arr . size (); ++ i) cout << arr [i] << " " ; } C #include <stdio.h> // To heapify a subtree rooted with node i void heapify (int arr [], int n , int i){ // Initialize largest as root int largest = i ; // left index = 2*i + 1 int l = 2 * i + 1 ; // right index = 2*i + 2 int r = 2 * i + 2 ; // If left child is larger than root if (l < n && arr [l] > arr [largest]) largest = l ; // If right child is larger than largest so far if (r < n && arr [r] > arr [largest]) largest = r ; // If largest is not root if (largest != i) { int temp = arr [i]; arr [i] = arr [largest]; arr [largest] = temp ; // Recursively heapify the affected sub-tree heapify (arr , n , largest); } } // Main function to do heap sort void heapSort (int arr [], int n){ // Build heap (rearrange vector) for (int i = n / 2 - 1 ; i >= 0 ; i --) heapify (arr , n , i); // One by one extract an element from heap for (int i = n - 1 ; i > 0 ; i --) { // Move current root to end int temp = arr [0]; arr [0] = arr [i]; arr [i] = temp ; // Call max heapify on the reduced heap heapify (arr , i , 0); } } int main () { int arr [] = { 9 , 4 , 3 , 8 , 10 , 2 , 5 }; int n = sizeof (arr) / sizeof (arr [0]); heapSort (arr , n); for (int i = 0 ; i < n ; ++ i) printf ("%d " , arr [i]); return 0 ; } Java import java.util.Arrays ; public class GFG { // To heapify a subtree rooted with node i static void heapify (int [] arr , int n , int i) { // Initialize largest as root int largest = i ; // left index = 2*i + 1 int l = 2 * i + 1 ; // right index = 2*i + 2 int r = 2 * i + 2 ; // If left child is larger than root if (l < n && arr [l] > arr [largest]) largest = l ; // If right child is larger than largest so far if (r < n && arr [r] > arr [largest]) largest = r ; // If largest is not root if (largest != i) { int temp = arr [i] ; arr [i] = arr [largest] ; arr [largest] = temp ; // Recursively heapify the affected sub-tree heapify (arr , n , largest); } } // Main function to do heap sort static void heapSort (int [] arr) { int n = arr . length ; // Build heap (rearrange vector) for (int i = n / 2 - 1 ; i >= 0 ; i --) heapify (arr , n , i); // One by one extract an element from heap for (int i = n - 1 ; i > 0 ; i --) { // Move current root to end int temp = arr [0] ; arr [0] = arr [i] ; arr [i] = temp ; // Call max heapify on the reduced heap heapify (arr , i , 0); } } public static void main (String [] args) { int [] arr = { 9 , 4 , 3 , 8 , 10 , 2 , 5 }; heapSort (arr); for (int i = 0 ; i < arr . length ; ++ i) System . out . print (arr [i] + " "); } } Python # To heapify a subtree rooted with node i def heapify (arr , n , i): # Initialize largest as root largest = i # left index = 2*i + 1 l = 2 * i + 1 # right index = 2*i + 2 r = 2 * i + 2 # If left child is larger than root if l < n and arr [l] > arr [largest]: largest = l # If right child is larger than largest so far if r < n and arr [r] > arr [largest]: largest = r # If largest is not root if largest != i : arr [i], arr [largest] = arr [largest], arr [i] # Recursively heapify the affected sub-tree heapify (arr , n , largest) # Main function to do heap sort def heapSort (arr): n = len (arr) # Build heap (rearrange vector) for i in range (n // 2 - 1 , - 1 , - 1): heapify (arr , n , i) # One by one extract an element from heap for i in range (n - 1 , 0 , - 1): # Move current root to end arr [0], arr [i] = arr [i], arr [0] # Call max heapify on the reduced heap heapify (arr , i , 0) if __name__ == "__main__" : arr = [9 , 4 , 3 , 8 , 10 , 2 , 5] heapSort (arr) for i in range (len (arr)): print (arr [i], end = " ") C# using System ; using System.Collections.Generic ; class GFG { // To heapify a subtree rooted with node i static void heapify (List < int > arr , int n , int i) { // Initialize largest as root int largest = i ; // left index = 2*i + 1 int l = 2 * i + 1 ; // right index = 2*i + 2 int r = 2 * i + 2 ; // If left child is larger than root if (l < n && arr [l] > arr [largest]) largest = l ; // If right child is larger than largest so far if (r < n && arr [r] > arr [largest]) largest = r ; // If largest is not root if (largest != i) { int temp = arr [i]; arr [i] = arr [largest]; arr [largest] = temp ; // Recursively heapify the affected sub-tree heapify (arr , n , largest); } } // Main function to do heap sort static void heapSort (List < int > arr) { int n = arr . Count ; // Build heap (rearrange vector) for (int i = n / 2 - 1 ; i >= 0 ; i --) heapify (arr , n , i); // One by one extract an element from heap for (int i = n - 1 ; i > 0 ; i --) { // Move current root to end int temp = arr [0]; arr [0] = arr [i]; arr [i] = temp ; // Call max heapify on the reduced heap heapify (arr , i , 0); } } static void Main () { List < int > arr = new List < int > { 9 , 4 , 3 , 8 , 10 , 2 , 5 }; heapSort (arr); foreach (int x in arr) Console . Write (x + " "); } } JavaScript // To heapify a subtree rooted with node i function heapify (arr , n , i) { // Initialize largest as root let largest = i ; // left index = 2*i + 1 let l = 2 * i + 1 ; // right index = 2*i + 2 let r = 2 * i + 2 ; // If left child is larger than root if (l < n && arr [l] > arr [largest]) largest = l ; // If right child is larger than largest so far if (r < n && arr [r] > arr [largest]) largest = r ; // If largest is not root if (largest != i) { [arr [i], arr [largest]] = [arr [largest], arr [i]]; // Recursively heapify the affected sub-tree heapify (arr , n , largest); } } // Main function to do heap sort function heapSort (arr) { let n = arr . length ; // Build heap (rearrange vector) for (let i = Math . floor (n / 2) - 1 ; i >= 0 ; i --) heapify (arr , n , i); // One by one extract an element from heap for (let i = n - 1 ; i > 0 ; i --) { // Move current root to end [arr [0], arr [i]] = [arr [i], arr [0]]; // Call max heapify on the reduced heap heapify (arr , i , 0); } } //Driver Code let arr = [9 , 4 , 3 , 8 , 10 , 2 , 5]; heapSort (arr); console . log (arr . join (" ")); Output Sorted array is 2 3 4 5 8 9 10 Time Complexity: O(n log n) Auxiliary Space: O(log n), due to the recursive call stack. However, auxiliary space can be O(1) for iterative implementation. Important points about Heap Sort An in-place algorithm. Its typical implementation is not stable but can be made stable (See this) Typically 2-3 times slower than well-implemented QuickSort . The reason for slowness is a lack of locality of reference. Advantages of Heap Sort Efficient Time Complexity: Heap Sort has a guaranteed time complexity of O(n log n) in all cases, making it suitable for large datasets. The log n factor comes from the height of the binary heap, ensuring consistent performance. Minimal Memory Usage : Heap Sort can work in-place with minimal extra memory. Using an iterative heapify() avoids additional stack space, so apart from storing the array itself, no extra memory is required. Simplicity: Heap Sort is relatively easy to understand and implement compared to other efficient sorting algorithms, as it relies on the straightforward binary heap structure without advanced concepts like recursion (if iterative heapify is used). 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. Comment Article Tags: Article Tags: Sorting Heap DSA Amazon Oracle Samsung Belzabar Intuit SAP Labs Visa 24*7 Innovation Labs Heap Sort + 8 More