Building Heap from Array - GeeksforGeeks

Source: https://www.geeksforgeeks.org/building-heap-from-array/

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 Building Heap from Array Last Updated : 18 Oct, 2025 Given an integer array arr[] , build a Max Heap from the given array. A Max Heap is a complete binary tree where each parent node is greater than or equal to its children, ensuring the largest element is at the root. Examples: Input: arr[] = [4, 10, 3, 5, 1] Output: Corresponding Max-Heap: Input : arr[] = [1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17] Output : Corresponding Max-Heap: [Approach] Using Recursion - O(n) Time and O(log n) Space To build a Max Heap from an array, treat the array as a complete binary tree and heapify nodes from the last non-leaf node up to the root in reverse level order. Leaf nodes already satisfy the heap property, so we start from the last non-leaf node, and for each subtree, we compare the parent with its children. Whenever a child node is greater than the parent, we swap them and continue heapifying that subtree to ensure the Max Heap property is maintained throughout. Note : Root is at index 0. Left child of node i -> 2*i + 1. Right child of node i -> 2*i + 2. Parent of node i -> (i-1)/2. Last non-leaf node -> parent of last node -> (n/2) - 1. C++ #include <iostream> #include <vector> using namespace std ; // To heapify a subtree void heapify (vector < int >& arr , int n , int i) { // Initialize largest as root int largest = i ; int l = 2 * i + 1 ; 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); } } // Function to build a Max-Heap from the given array void buildHeap (vector < int >& arr) { int n = arr . size (); // Index of last non-leaf node int startIdx = (n / 2) - 1 ; // Perform reverse level order traversal // from last non-leaf node and heapify // each node for (int i = startIdx ; i >= 0 ; i --) { heapify (arr , n , i); } } int main () { // Binary Tree Representation // of input array // 1 // / \ // 3 5 // / \ / \ // 4 6 13 10 // / \ / \ // 9 8 15 17 vector < int > arr = { 1 , 3 , 5 , 4 , 6 , 13 , 10 , 9 , 8 , 15 , 17 }; // Function call buildHeap (arr); for (int i = 0 ; i < arr . size (); ++ i) cout << arr [i] << " " ; cout << " \n " ; // Final Heap: // 17 // / \ // 15 13 // / \ / \ // 9 6 5 10 // / \ / \ // 4 8 3 1 return 0 ; } C #include <stdio.h> #include <stdlib.h> // To heapify a subtree void heapify (int arr [], int n , int i) { // Initialize largest as root int largest = i ; int l = 2 * i + 1 ; 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); } } // Function to build a Max-Heap from the given array void buildHeap (int arr [], int n) { // Index of last non-leaf node int startIdx = (n / 2) - 1 ; // Perform reverse level order traversal // from last non-leaf node and heapify // each node for (int i = startIdx ; i >= 0 ; i --) { heapify (arr , n , i); } } int main () { // Binary Tree Representation // of input array // 1 // / \ // 3 5 // / \ / \ // 4 6 13 10 // / \ / \ // 9 8 15 17 int arr [] = { 1 , 3 , 5 , 4 , 6 , 13 , 10 , 9 , 8 , 15 , 17 }; int n = sizeof (arr) / sizeof (arr [0]); // Function call buildHeap (arr , n); for (int i = 0 ; i < n ; ++ i) printf ("%d " , arr [i]); printf (" \n "); // Final Heap: // 17 // / \ // 15 13 // / \ / \ // 9 6 5 10 // / \ / \ // 4 8 3 1 return 0 ; } Java public class GfG { // To heapify a subtree static void heapify (int arr [] , int n , int i) { // Initialize largest as root int largest = i ; int l = 2 * i + 1 ; 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); } } // Function to build a Max-Heap from the given array static void buildHeap (int arr []) { int n = arr . length ; // Index of last non-leaf node int startIdx = (n / 2) - 1 ; // Perform reverse level order traversal // from last non-leaf node and heapify // each node for (int i = startIdx ; i >= 0 ; i --) { heapify (arr , n , i); } } public static void main (String [] args) { // Binary Tree Representation // of input array // 1 // / \ // 3 5 // / \ / \ // 4 6 13 10 // / \ / \ // 9 8 15 17 int arr [] = { 1 , 3 , 5 , 4 , 6 , 13 , 10 , 9 , 8 , 15 , 17 }; int n = arr . length ; // Function call buildHeap (arr); for (int i = 0 ; i < n ; ++ i) System . out . print (arr [i] + " "); System . out . println (); // Final Heap: // 17 // / \ // 15 13 // / \ / \ // 9 6 5 10 // / \ / \ // 4 8 3 1 } } Python # To heapify a subtree def heapify (arr , n , i): # Initialize largest as root largest = i l = 2 * i + 1 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) # Function to build a Max-Heap from the given array def buildHeap (arr): n = len (arr) # Index of last non-leaf node startIdx = (n // 2) - 1 # Perform reverse level order traversal # from last non-leaf node and heapify # each node for i in range (startIdx , - 1 , - 1): heapify (arr , n , i) if __name__ == "__main__" : # Binary Tree Representation # of input array # 1 # / \ # 3 5 # / \ / \ # 4 6 13 10 # / \ / \ # 9 8 15 17 arr = [1 , 3 , 5 , 4 , 6 , 13 , 10 , 9 , 8 , 15 , 17] n = len (arr) # Function call buildHeap (arr) for i in range (n): print (arr [i], end = " ") print () # Final Heap: # 17 # / \ # 15 13 # / \ / \ # 9 6 5 10 # / \ / \ # 4 8 3 1 C# using System ; class GFG { // To heapify a subtree static void heapify (int [] arr , int n , int i) { // Initialize largest as root int largest = i ; int l = 2 * i + 1 ; 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); } } // Function to build a Max-Heap from the given array static void buildHeap (int [] arr) { int n = arr . Length ; // Index of last non-leaf node int startIdx = (n / 2) - 1 ; // Perform reverse level order traversal // from last non-leaf node and heapify // each node for (int i = startIdx ; i >= 0 ; i --) { heapify (arr , n , i); } } static void Main () { // Binary Tree Representation // of input array // 1 // / \ // 3 5 // / \ / \ // 4 6 13 10 // / \ / \ // 9 8 15 17 int [] arr = { 1 , 3 , 5 , 4 , 6 , 13 , 10 , 9 , 8 , 15 , 17 }; int n = arr . Length ; // Function call buildHeap (arr); for (int i = 0 ; i < n ; i ++) Console . Write (arr [i] + " "); Console . WriteLine (); // Final Heap: // 17 // / \ // 15 13 // / \ / \ // 9 6 5 10 // / \ / \ // 4 8 3 1 } } JavaScript // To heapify a subtree rooted function heapify (arr , n , i) { let largest = i ; let l = 2 * i + 1 ; 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); } } // Function to build a Max-Heap from the given array function buildHeap (arr) { const n = arr . length ; // Index of last non-leaf node let startIdx = Math . floor (n / 2) - 1 ; // Perform reverse level order traversal // from last non-leaf node and heapify // each node for (let i = startIdx ; i >= 0 ; i --) { heapify (arr , n , i); } } // Driver Code // Binary Tree Representation of input array // 1 // / \ // 3 5 // / \ / \ // 4 6 13 10 // / \ / \ // 9 8 15 17 const arr = [1 , 3 , 5 , 4 , 6 , 13 , 10 , 9 , 8 , 15 , 17]; const n = arr . length ; // Build Max Heap buildHeap (arr); for (let i = 0 ; i < n ; i ++) process . stdout . write (arr [i] + " "); console . log ("\n"); // Final Heap Representation // 17 // / \ // 15 13 // / \ / \ // 9 6 5 10 // / \ / \ // 4 8 3 1 Output 17 15 13 9 6 5 10 4 8 3 1 Comment Article Tags: Article Tags: Misc Heap DSA Arrays