Huffman Coding | Greedy Algo-3 - GeeksforGeeks

Source: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

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 Huffman Coding | Greedy Algo-3 Last Updated : 19 Dec, 2025 Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream. Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be "cccd" or "ccb" or "acd" or "ab". There are mainly two major parts in Huffman Coding Build a Huffman Tree from input characters. Traverse the Huffman Tree and assign codes to characters. Steps to build Huffman Tree This algorithm builds a tree in bottom up manner using a priority queue (or heap) . Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root) Extract two nodes with the minimum frequency from the min heap. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete. Let us understand the algorithm with an example: character Frequency a 5 b 9 c 12 d 13 e 16 f 45 Try it on GfG Practice Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node. Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14. Illustration of step 2 Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements character Frequency c 12 d 13 Internal Node 14 e 16 f 45 Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25 Illustration of step 3 Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes character Frequency Internal Node 14 e 16 Internal Node 25 f 45 Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30 Illustration of step 4 Now min heap contains 3 nodes. character Frequency Internal Node 25 Internal Node 30 f 45 Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55 Illustration of step 5 Now min heap contains 2 nodes. character Frequency f 45 Internal Node 55 Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100 Illustration of step 6 Now min heap contains only one node. character Frequency Internal Node 100 Since the heap contains only one node, the algorithm stops here. Steps to print codes from Huffman Tree: Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered. Steps to print code from HuffmanTree The codes are as follows: character code-word f 0 c 100 d 101 a 1100 b 1101 e 111 C++ // C++ code for the above approach: #include <bits/stdc++.h> using namespace std ; // Class to represent huffman tree class Node { public : int data ; Node * left , * right ; Node (int x) { data = x ; left = nullptr ; right = nullptr ; } }; // Custom min heap for Node class. class Compare { public : bool operator () (Node * a , Node * b) { return a -> data > b -> data ; } }; // Function to traverse tree in preorder // manner and push the huffman representation // of each character. void preOrder (Node * root , vector < string > & ans , string curr) { if (root == nullptr) return ; // Leaf node represents a character. if (root -> left == nullptr && root -> right == nullptr) { ans . push_back (curr); return ; } preOrder (root -> left , ans , curr + '0'); preOrder (root -> right , ans , curr + '1'); } vector < string > huffmanCodes (string s , vector < int > freq) { int n = s . length (); // Min heap for node class. priority_queue < Node * , vector < Node *> , Compare > pq ; for (int i = 0 ; i < n ; i ++) { Node * tmp = new Node (freq [i]); pq . push (tmp); } // Construct huffman tree. while (pq . size () >= 2) { // Left node Node * l = pq . top (); pq . pop (); // Right node Node * r = pq . top (); pq . pop (); Node * newNode = new Node (l -> data + r -> data); newNode -> left = l ; newNode -> right = r ; pq . push (newNode); } Node * root = pq . top (); vector < string > ans ; preOrder (root , ans , ""); return ans ; } int main () { string s = "abcdef" ; vector < int > freq = { 5 , 9 , 12 , 13 , 16 , 45 }; vector < string > ans = huffmanCodes (s , freq); for (int i = 0 ; i < ans . size (); i ++) { cout << ans [i] << " " ; } return 0 ; } Java // Java program for the above approach: import java.util.* ; // Class to represent huffman tree class Node { int data ; Node left , right ; Node (int x) { data = x ; left = null ; right = null ; } } class GfG { // Function to traverse tree in preorder // manner and push the huffman representation // of each character. static void preOrder (Node root , ArrayList < String > ans , String curr) { if (root == null) return ; // Leaf node represents a character. if (root . left == null && root . right == null) { ans . add (curr); return ; } preOrder (root . left , ans , curr + '0'); preOrder (root . right , ans , curr + '1'); } static ArrayList < String > huffmanCodes (String s , int [] freq) { int n = s . length (); // Min heap for node class. PriorityQueue < Node > pq = new PriorityQueue <> ((a , b) -> { if (a . data < b . data) return - 1 ; return 1 ; }); for (int i = 0 ; i < n ; i ++) { Node tmp = new Node (freq [i]); pq . add (tmp); } // Construct huffman tree. while (pq . size () >= 2) { // Left node Node l = pq . poll (); // Right node Node r = pq . poll (); Node newNode = new Node (l . data + r . data); newNode . left = l ; newNode . right = r ; pq . add (newNode); } Node root = pq . poll (); ArrayList < String > ans = new ArrayList <> (); preOrder (root , ans , ""); return ans ; } public static void main (String [] args) { String s = "abcdef" ; int [] freq = { 5 , 9 , 12 , 13 , 16 , 45 }; ArrayList < String > ans = huffmanCodes (s , freq); for (int i = 0 ; i < ans . size (); i ++) { System . out . print (ans . get (i) + " "); } } } Python # Python program for the above approach: import heapq # Class to represent huffman tree class Node : def __init__ (self , x): self . data = x self . left = None self . right = None def __lt__ (self , other): return self . data < other . data # Function to traverse tree in preorder # manner and push the huffman representation # of each character. def preOrder (root , ans , curr): if root is None : return # Leaf node represents a character. if root . left is None and root . right is None : ans . append (curr) return preOrder (root . left , ans , curr + '0') preOrder (root . right , ans , curr + '1') def huffmanCodes (s , freq): # Code here n = len (s) # Min heap for node class. pq = [] for i in range (n): tmp = Node (freq [i]) heapq . heappush (pq , tmp) # Construct huffman tree. while len (pq) >= 2 : # Left node l = heapq . heappop (pq) # Right node r = heapq . heappop (pq) newNode = Node (l . data + r . data) newNode . left = l newNode . right = r heapq . heappush (pq , newNode) root = heapq . heappop (pq) ans = [] preOrder (root , ans , "") return ans if __name__ == "__main__" : s = "abcdef" freq = [5 , 9 , 12 , 13 , 16 , 45] ans = huffmanCodes (s , freq) for code in ans : print (code , end = " ") C# // C# program for the above approach: using System ; using System.Collections.Generic ; // Class to represent huffman tree class Node { public int data ; public Node left , right ; public Node (int x) { data = x ; left = null ; right = null ; } } class GfG { // Function to traverse tree in preorder // manner and push the huffman representation // of each character. static void preOrder (Node root , List < string > ans , string curr) { if (root == null) return ; // Leaf node represents a character. if (root . left == null && root . right == null) { ans . Add (curr); return ; } preOrder (root . left , ans , curr + "0"); preOrder (root . right , ans , curr + "1"); } static List < string > huffmanCodes (string s , int [] freq) { int n = s . Length ; // Min heap for node class. PriorityQueue < Node > pq = new PriorityQueue < Node > (new Comparer ()); for (int i = 0 ; i < n ; i ++) { Node tmp = new Node (freq [i]); pq . Enqueue (tmp); } // Construct huffman tree. while (pq . Count >= 2) { // Left node Node l = pq . Dequeue (); // Right node Node r = pq . Dequeue (); Node newNode = new Node (l . data + r . data); newNode . left = l ; newNode . right = r ; pq . Enqueue (newNode); } Node root = pq . Dequeue (); List < string > ans = new List < string > (); preOrder (root , ans , ""); return ans ; } static void Main (string [] args) { string s = "abcdef" ; int [] freq = { 5 , 9 , 12 , 13 , 16 , 45 }; List < string > ans = huffmanCodes (s , freq); for (int i = 0 ; i < ans . Count ; i ++) { Console . Write (ans [i] + " "); } } } // Custom comparator class for min heap class Comparer : IComparer < Node > { public int Compare (Node a , Node b) { if (a . data > b . data) return 1 ; else if (a . data < b . data) return - 1 ; return 0 ; } } // Custom Priority queue class PriorityQueue < T > { private List < T > heap ; private IComparer < T > comparer ; public PriorityQueue (IComparer < T > comparer = null) { this . heap = new List < T > (); this . comparer = comparer ?? Comparer < T > . Default ; } public int Count => heap . Count ; // Enqueue operation public void Enqueue (T item) { heap . Add (item); int i = heap . Count - 1 ; while (i > 0) { int parent = (i - 1) / 2 ; if (comparer . Compare (heap [parent], heap [i]) <= 0) break ; Swap (parent , i); i = parent ; } } // Dequeue operation public T Dequeue () { if (heap . Count == 0) throw new InvalidOperationException ("Priority queue is empty."); T result = heap [0]; int last = heap . Count - 1 ; heap [0] = heap [last]; heap . RemoveAt (last); last -- ; int i = 0 ; while (true) { int left = 2 * i + 1 ; if (left > last) break ; int right = left + 1 ; int minChild = left ; if (right <= last && comparer . Compare (heap [right], heap [left]) < 0) minChild = right ; if (comparer . Compare (heap [i], heap [minChild]) <= 0) break ; Swap (i , minChild); i = minChild ; } return result ; } // Swap two elements in the heap private void Swap (int i , int j) { T temp = heap [i]; heap [i] = heap [j]; heap [j] = temp ; } } JavaScript // JavaScript program for the above approach: class PriorityQueue { constructor (compare) { this . heap = []; this . compare = compare ; } enqueue (value) { this . heap . push (value); this . bubbleUp (); } bubbleUp () { let index = this . heap . length - 1 ; while (index > 0) { let element = this . heap [index], parentIndex = Math . floor ((index - 1) / 2), parent = this . heap [parentIndex]; if (this . compare (element , parent) < 0) break ; this . heap [index] = parent ; this . heap [parentIndex] = element ; index = parentIndex ; } } dequeue () { let max = this . heap [0]; let end = this . heap . pop (); if (this . heap . length > 0) { this . heap [0] = end ; this . sinkDown (0); } return max ; } sinkDown (index) { let left = 2 * index + 1 , right = 2 * index + 2 , largest = index ; if (left < this . heap . length && this . compare (this . heap [left], this . heap [largest]) > 0) { largest = left ; } if (right < this . heap . length && this . compare (this . heap [right], this . heap [largest]) > 0) { largest = right ; } if (largest !== index) { [this . heap [largest], this . heap [index]] = [this . heap [index], this . heap [largest],]; this . sinkDown (largest); } } isEmpty () { return this . heap . length === 0 ; } } // Class to represent huffman tree class Node { constructor (x) { this . data = x ; this . left = null ; this . right = null ; } } // Function to traverse tree in preorder // manner and push the huffman representation // of each character. function preOrder (root , ans , curr) { if (root === null) return ; // Leaf node represents a character. if (root . left === null && root . right === null) { ans . push (curr); return ; } preOrder (root . left , ans , curr + '0'); preOrder (root . right , ans , curr + '1'); } function huffmanCodes (s , freq) { let n = s . length ; // Min heap for node class. let pq = new PriorityQueue ((a , b) => { if (a . data <= b . data) return 1 ; return - 1 ; }); for (let i = 0 ; i < n ; i ++) { let tmp = new Node (freq [i]); pq . enqueue (tmp); } // Construct huffman tree. while (pq . heap . length >= 2) { // Left node let l = pq . dequeue (); // Right node let r = pq . dequeue (); let newNode = new Node (l . data + r . data); newNode . left = l ; newNode . right = r ; pq . enqueue (newNode); } let root = pq . dequeue (); let ans = []; preOrder (root , ans , ""); return ans ; } let s = "abcdef" ; let freq = [5 , 9 , 12 , 13 , 16 , 45]; let ans = huffmanCodes (s , freq); console . log (ans . join (" ")); Output 0 100 101 1100 1101 111 Time complexity: O(nlogn) where n is the number of unique characters Space complexity :- O(n) Applications of Huffman Coding: They are used for transmitting fax and text. They are used by conventional compression formats like PKZIP, GZIP, etc. Multimedia codecs like JPEG, PNG, and MP3 use Huffman encoding(to be more precise the prefix codes). It is useful in cases where there is a series of frequently occurring characters. Comment Article Tags: Article Tags: Greedy Heap DSA Amazon Huffman Coding Morgan Stanley Samsung United Health Group encoding-decoding priority-queue + 6 More