Binary Indexed Tree or Fenwick Tree - GeeksforGeeks

Source: https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree/

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 Binary Indexed Tree or Fenwick Tree Last Updated : 24 Jan, 2026 Binary Indexed Trees are used for problems where we have following types of multiple operations on a fixed sized. Prefix Operation (Sum, Product, XOR, OR, etc). Note that range operations can also be solved using prefix. For example, range sum from index L to R is prefix sum till R (included minus prefix sum til L-1. Update an array item Time Complexities of both the operations is O(Log n). Note that we need O(n Log n) preprocessing time and O(n) auxiliary space. Example Problem to Understand Binary Index Tree Let us consider the following problem to understand Binary Indexed Tree. We have an array arr[0 . . . n-1]. We would like to Compute the sum of the first i elements. Modify the value of a specified element of the array arr[i] = x where 0 <= i <= n-1. A simple solution is to run a loop from 0 to i-1 and calculate the sum of the elements. To update a value, simply do arr[i] = x. The first operation takes O(n) time and the second operation takes O(1) time. Another simple solution is to create an extra array and store the sum of the first i-th elements at the i-th index in this new array. The sum of a given range can now be calculated in O(1) time, but the update operation takes O(n) time now. This works well if there are a large number of query operations but a very few number of update operations. Could we perform both the query and update operations in O(log n) time? One efficient solution is to use Segment Tree that performs both operations in O(Logn) time. An alternative solution is Binary Indexed Tree, which also achieves O(Logn) time complexity for both operations. Compared with Segment Tree, Binary Indexed Tree requires less space and is easier to implement. . Representation Binary Indexed Tree is represented as an array. Let the array be BITree[]. Each node of the Binary Indexed Tree stores the sum of some elements of the input array. The size of the Binary Indexed Tree is equal to the size of the input array, denoted as n. In the code below, we use a size of n+1 for ease of implementation. Construction We initialize all the values in BITree[] as 0. Then we call update() for all the indexes, the update() operation is discussed below. Operations getSum(x): Returns the sum of the sub-array arr[0,...,x] // Returns the sum of the sub-array arr[0,...,x] using BITree[0..n], which is constructed from arr[0..n-1] 1) Initialize the output sum as 0, the current index as x+1. 2) Do following while the current index is greater than 0. ...a) Add BITree[index] to sum ...b) Go to the parent of BITree[index]. The parent can be obtained by removing the last set bit from the current index, i.e., index = index - (index & (-index)) 3) Return sum. The diagram above provides an example of how getSum() is working. Here are some important observations. BITree[0] is a dummy node. BITree[y] is the parent of BITree[x], if and only if y can be obtained by removing the last set bit from the binary representation of x, that is y = x - (x & (-x)). The child node BITree[x] of the node BITree[y] stores the sum of the elements between y(inclusive) and x(exclusive): arr[y,...,x). update(x, val): Updates the Binary Indexed Tree (BIT) by performing arr[index] += val // Note that the update(x, val) operation will not change arr[]. It only makes changes to BITree[] 1) Initialize the current index as x+1. 2) Do the following while the current index is smaller than or equal to n. ...a) Add the val to BITree[index] ...b) Go to next element of BITree[index]. The next element can be obtained by incrementing the last set bit of the current index, i.e., index = index + (index & (-index)) The update function needs to make sure that all the BITree nodes which contain arr[i] within their ranges being updated. We loop over such nodes in the BITree by repeatedly adding the decimal number corresponding to the last set bit of the current index. How does Binary Indexed Tree work? The

idea is based on the fact that all positive integers can be represented as the sum of powers of 2. For example 19 can be represented as 16 + 2 + 1. Every node of the BITree stores the sum of n elements where n is a power of 2. For example, in the first diagram above (the diagram for getSum()), the sum of the first 12 elements can be obtained by the sum of the last 4 elements (from 9 to 12) plus the sum of 8 elements (from 1 to 8). The number of set bits in the binary representation of a number n is O(Logn). Therefore, we traverse at-most O(Logn) nodes in both getSum() and update() operations. The time complexity of the construction is O(nLogn) as it calls update() for all n elements. C++ // C++ code to demonstrate operations of Binary Index Tree #include <iostream> using namespace std ; /* n --> No. of elements present in input array. BITree[0..n] --> Array that represents Binary Indexed Tree. arr[0..n-1] --> Input array for which prefix sum is evaluated. */ // Returns sum of arr[0..index]. This function assumes // that the array is preprocessed and partial sums of // array elements are stored in BITree[]. int getSum (int BITree [], int index) { int sum = 0 ; // Initialize result // index in BITree[] is 1 more than the index in arr[] index = index + 1 ; // Traverse ancestors of BITree[index] while (index > 0) { // Add current element of BITree to sum sum += BITree [index]; // Move index to parent node in getSum View index -= index & (- index); } return sum ; } // Updates a node in Binary Index Tree (BITree) at given index // in BITree. The given value 'val' is added to BITree[i] and // all of its ancestors in tree. void updateBIT (int BITree [], int n , int index , int val) { // index in BITree[] is 1 more than the index in arr[] index = index + 1 ; // Traverse all ancestors and add 'val' while (index <= n) { // Add 'val' to current node of BI Tree BITree [index] += val ; // Update index to that of parent in update View index += index & (- index); } } // Constructs and returns a Binary Indexed Tree for given // array of size n. int * constructBITree (int arr [], int n) { // Create and initialize BITree[] as 0 int * BITree = new int [n + 1]; for (int i = 1 ; i <= n ; i ++) BITree [i] = 0 ; // Store the actual values in BITree[] using update() for (int i = 0 ; i < n ; i ++) updateBIT (BITree , n , i , arr [i]); // Uncomment below lines to see contents of BITree[] //for (int i=1; i<=n; i++) // cout << BITree[i] << " "; return BITree ; } // Driver program to test above functions int main () { int freq [] = { 2 , 1 , 1 , 3 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 }; int n = sizeof (freq) / sizeof (freq [0]); int * BITree = constructBITree (freq , n); cout << "Sum of elements in arr[0..5] is " << getSum (BITree , 5); // Let use test the update operation freq [3] += 6 ; updateBIT (BITree , n , 3 , 6); //Update BIT for above change in arr[] cout << " \n Sum of elements in arr[0..5] after update is " << getSum (BITree , 5); return 0 ; } Java // Java program to demonstrate lazy // propagation in segment tree import java.util.* ; import java.lang.* ; import java.io.* ; class BinaryIndexedTree { // Max tree size final static int MAX = 1000 ; static int BITree [] = new int [MAX] ; /* n --> No. of elements present in input array. BITree[0..n] --> Array that represents Binary Indexed Tree. arr[0..n-1] --> Input array for which prefix sum is evaluated. */ // Returns sum of arr[0..index]. This function // assumes that the array is preprocessed and // partial sums of array elements are stored // in BITree[]. int getSum (int index) { int sum = 0 ; // Initialize result // index in BITree[] is 1 more than // the index in arr[] index = index + 1 ; // Traverse ancestors of BITree[index] while (index > 0) { // Add current element of BITree // to sum sum += BITree [index] ; // Move index to parent node in // getSum View index -= index & (- index); } return sum ; } // Updates a node in Binary Index Tree (BITree) // at given index in BITree. The given value // 'val' is added to BITree[i] and all of // its ancestors in tree. public static void updateBIT (int n , int index , int val) { // index in BITree[] is 1 more than // the index in arr[] index = index + 1 ; // Traverse all ancestors and add 'val' while (index <= n) { // Add 'val' to current node of BIT Tree BITree [index] += val ; // Update index to that of parent // in update View index += index & (- index); } } /* Function to construct fenwick tree from given array.*/ void constructBITree (int arr [] , int n) { // Initialize BITree[] as 0 for (int i = 1 ; i <= n ; i ++) BITree [i] = 0 ; // Store the actual values in BITree[] // using update() for (int i = 0 ; i < n ; i ++) updateBIT (n , i , arr [i]); } // Main function public static void main (String args []) { int freq [] = { 2 , 1 , 1 , 3 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 }; int n = freq . length ; BinaryIndexedTree tree = new BinaryIndexedTree (); // Build fenwick tree from given array tree . constructBITree (freq , n); System . out . println ("Sum of elements in arr[0..5]" + " is " + tree . getSum (5)); // Let use test the update operation freq [3] += 6 ; // Update BIT for above change in arr[] updateBIT (n , 3 , 6); // Find sum after the value is updated System . out . println ("Sum of elements in arr[0..5]" + " after update is " + tree . getSum (5)); } } // This code is contributed by Ranjan Binwani Python # Python implementation of Binary Indexed Tree # Returns sum of arr[0..index]. This function assumes # that the array is preprocessed and partial sums of # array elements are stored in BITree[]. def getsum (BITTree , i): s = 0 #initialize result # index in BITree[] is 1 more than the index in arr[] i = i + 1 # Traverse ancestors of BITree[index] while i > 0 : # Add current element of BITree to sum s += BITTree [i] # Move index to parent node in getSum View i -= i & (- i) return s # Updates a node in Binary Index Tree (BITree) at given index # in BITree. The given value 'val' is added to BITree[i] and # all of its ancestors in tree. def

updatebit (BITTree , n , i , v): # index in BITree[] is 1 more than the index in arr[] i += 1 # Traverse all ancestors and add 'val' while i <= n : # Add 'val' to current node of BI Tree BITTree [i] += v # Update index to that of parent in update View i += i & (- i) # Constructs and returns a Binary Indexed Tree for given # array of size n. def construct (arr , n): # Create and initialize BITree[] as 0 BITTree = [0] * (n + 1) # Store the actual values in BITree[] using update() for i in range (n): updatebit (BITTree , n , i , arr [i]) # Uncomment below lines to see contents of BITree[] #for i in range(1,n+1): # print BITTree[i], return BITTree # Driver code to test above methods freq = [2 , 1 , 1 , 3 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9] BITTree = construct (freq , len (freq)) print ("Sum of elements in arr[0..5] is " + str (getsum (BITTree , 5))) freq [3] += 6 updatebit (BITTree , len (freq), 3 , 6) print ("Sum of elements in arr[0..5]" + " after update is " + str (getsum (BITTree , 5))) # This code is contributed by Raju Varshney C# // C# program to demonstrate lazy // propagation in segment tree using System ; public class BinaryIndexedTree { // Max tree size readonly static int MAX = 1000 ; static int [] BITree = new int [MAX]; /* n --> No. of elements present in input array. BITree[0..n] --> Array that represents Binary Indexed Tree. arr[0..n-1] --> Input array for which prefix sum is evaluated. */ // Returns sum of arr[0..index]. This function // assumes that the array is preprocessed and // partial sums of array elements are stored // in BITree[]. int getSum (int index) { int sum = 0 ; // Initialize result // index in BITree[] is 1 more than // the index in arr[] index = index + 1 ; // Traverse ancestors of BITree[index] while (index > 0) { // Add current element of BITree // to sum sum += BITree [index]; // Move index to parent node in // getSum View index -= index & (- index); } return sum ; } // Updates a node in Binary Index Tree (BITree) // at given index in BITree. The given value // 'val' is added to BITree[i] and all of // its ancestors in tree. public static void updateBIT (int n , int index , int val) { // index in BITree[] is 1 more than // the index in arr[] index = index + 1 ; // Traverse all ancestors and add 'val' while (index <= n) { // Add 'val' to current node of BIT Tree BITree [index] += val ; // Update index to that of parent // in update View index += index & (- index); } } /* Function to construct fenwick tree from given array.*/ void constructBITree (int [] arr , int n) { // Initialize BITree[] as 0 for (int i = 1 ; i <= n ; i ++) BITree [i] = 0 ; // Store the actual values in BITree[] // using update() for (int i = 0 ; i < n ; i ++) updateBIT (n , i , arr [i]); } // Driver code public static void Main (String [] args) { int [] freq = { 2 , 1 , 1 , 3 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 }; int n = freq . Length ; BinaryIndexedTree tree = new BinaryIndexedTree (); // Build fenwick tree from given array tree . constructBITree (freq , n); Console . WriteLine ("Sum of elements in arr[0..5]" + " is " + tree . getSum (5)); // Let use test the update operation freq [3] += 6 ; // Update BIT for above change in arr[] updateBIT (n , 3 , 6); // Find sum after the value is updated Console . WriteLine ("Sum of elements in arr[0..5]" + " after update is " + tree . getSum (5)); } } // This code is contributed by PrinciRaj1992 JavaScript /* JavaScript implementation of Binary Indexed Tree */ // Returns sum of arr[0..index]. This function assumes // that the array is preprocessed and partial sums of // array elements are stored in BITree[]. function getsum (BITTree , i) { let s = 0 ; // initialize result // index in BITree[] is 1 more than the index in arr[] i = i + 1 ; // Traverse ancestors of BITree[index] while (i > 0) { // Add current element of BITree to sum s += BITTree [i]; // Move index to parent node in getSum View i -= i & (- i); } return s ; } // Updates a node in Binary Index Tree (BITree) at given index // in BITree. The given value 'val' is added to BITree[i] and // all of its ancestors in tree. function updatebit (BITTree , n , i , v) { // index in BITree[] is 1 more than the index in arr[] i += 1 ; // Traverse all ancestors and add 'val' while (i <= n) { // Add 'val' to current node of BI Tree BITTree [i] += v ; // Update index to that of parent in update View i += i & (- i); } } // Constructs and returns a Binary Indexed Tree for given // array of size n. function construct (arr , n) { // Create and initialize BITree[] as 0 let BITTree = new Array (n + 1). fill (0); // Store the actual values in BITree[] using update() for (let i = 0 ; i < n ; i ++) { updatebit (BITTree , n , i , arr [i]); } // Uncomment below lines to see contents of BITree[] //for (let i = 1; i <= n; i++) { // console.log(BITTree[i]); //} return BITTree ; } // Driver code to test above methods let freq = [2 , 1 , 1 , 3 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9]; let BITTree = construct (freq , freq . length); console . log ("Sum of elements in arr[0..5] is " + getsum (BITTree , 5)); freq [3] += 6 ; updatebit (BITTree , freq . length , 3 , 6); console . log ("Sum of elements in arr[0..5] after update is " + getsum (BITTree , 5)); Output Sum of elements in arr[0..5] is 12 Sum of elements in arr[0..5] after update is 18 Time Complexity: O(N Log N) Auxiliary Space: O(N) Can we extend the Binary Indexed Tree to computing the sum of a range in O(Logn) time? Yes. rangeSum(l, r) = getSum(r) - getSum(l-1). Applications: The implementation of the arithmetic coding algorithm. The development of the Binary Indexed Tree was primarily motivated by its application in this case. Example Problems: Count inversions in an array | Set 3 (Using BIT) Two Dimensional Binary Indexed Tree or Fenwick Tree Counting Triangles in a Rectangular space using BIT Comment Article Tags: Article Tags: Tree Advanced Data Structure DSA array-range-queries Binary Indexed Tree Tutorials + 2 More