Binary Indexed Tree or Fenwick Tree

Let us consider the following problem to understand Binary Indexed Tree.

We have an array arr[0 . . . n-1]. We would like to  
**1** Compute the sum of the first i elements.  
**2**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](https://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/) 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 parent of BITree[index]. The parent 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.