**Fenwick Tree or Binary Indexed Tree(BIT)**

**Concept:**

Consider a Array [ 0…..N ] and we have to process the query to get the sum in the given range.

Lets have a query to find the sum in range [0,K].

This problem can be solved in the following ways.  
**Brute Force** : Running from i=0 to K and finding the Sum+=Arr[i].

**Dp Array :** We can maintain a Dp Prefix sum and get the Sum[K].

**Segment Tree :** We can always process range queries using Segment Tree.

**Update and Query Times for above mentioned**:

**Brute Force** Update takes O(1) and Query takes O(N) .

**Dp Array** Update takes O(N) to update entire array, and Query takes O(1).

**Segment Tree** Query time O(logN) , Update is very complex with Lazy Propagation.

Hence in This Case, **Fenwick Tree** or **BIT** is efficient which takes

**O ( N\*Log(N) ) -** To Create the Tree.

**O ( Log(N) ) -** To Update the Tree.

**O ( Log(N) ) -** For A Query.

**NOTE :-** This is because the No. of set bits in a BIT is atmost Log(N).

**Conceptual Representation :**

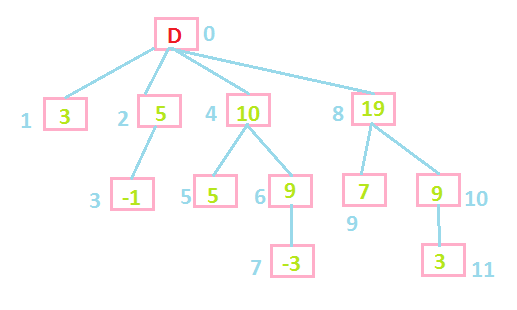
Consider an Array,

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 2 | -1 | 6 | 5 | 4 | -3 | 3 | 7 | 2 | 3 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

**Elements** =

**Index**  =

**The Tree Will Look Like**



**NOTE :** This is just a theoritical representation, we will actually use Array to store the Tree.

Here D = Dummy, Which does not Contribute.   
The Parent-Child Relationship is as follows.

**To Get Parent of Node** :

To get the parent, we have to set the Right most set bit to 0.

For Example,

7 = 111 , setting Right most set bit to 0 becomes 110=> 6 (Parent).

5 = 101 , setting Right most set bit to 0 becomes 100=>4 (Parent).

8= 1000, setting Right most set bit to 0 becomes 0000=>0 (Parent).

4= 100, setting Right most set bit to 0 becomes 000=>0 (Parent).

We can efficiently set the right most bit to 0 by following operations.

1. Find 2’s Complement of that Number.
2. Make AND Operation with Original Number.
3. Subtract the result from the original Number.

**Filling Up The Tree : O( N\*Log(N) )**

Here we traverse the entire array, for each index i, we select Idx = i+1 first.  
Fill the Tree[Idx]=Arr[i].  
And also update the remaining tree which is effected by updating the current node by getting the next using the formula

**GetNext :**

1. 2’s Compliment of current Node.
2. Perform AND operation with original Number.
3. Add the Result to the Original Number.

There will be Log( Idx ) GetNexts for a selected Idx, which makes the algorithm to run in N\*Log(N) time for N elements in the worst case.

**Updating the Tree :**

Updating the tree is same as filling the tree, we use the same GetNext formula , we update the current Node and the GetNexts with the difference of Original Arr[i] and Updated Arr[i].

**Query :**

We can Ask Query to get the sum of the elements in the range [0,X] , how ever if we ever wanted to get a range query for example in the range [X1,X2] , then we can call the query two times namely [0,X2] and [0,X1-1] and subtracting those two will give the answer of the range [X1,X2].

To get the Sum in Range [0,X] , we choose X+1th Node  
Sum = Tree[X+1] + parent(X+1) + parent( parent(X+1) ) + …. Untill we reach parent 0.

**NOTE : GetNext Should be called Untill The max node is reached <= N+1**