Created by Hisem

LeetCode: @hiranjoshi

Date 31-12-23
Page

Fenwick Tree / Binary Indexed Tree

- Let's have a well defined problem.

: We will be given an array, and we are asked to answer few queries on it.

The queries will be of two types:

(i) Update query : Replace "idx" index value by "target" value

(ii) Sum query : find sum of a given range L to R, inclusive.

* Let's look at the naive solution, before going for the BIT:

<1> We can update any value in just one step. This operation will take O(1) in the worst cases. Sum operation can be made by traversing the array, this operation will take O(n) linear time in the worst case.

<2> The another approach that we can do is at each index we will store the "cumulative frequency" or "Prefix Sum". We will store sum of all elements before it and including itself. We can build this new array in O(n) time. Let's say this

array is called "prefSum[]." Now in this, all the sum operation will take O(1) Time, Since we will just subtract "prefsum [L-1]" from "prefSum [R]" to get answer for the sum of range L to R. But, we will need to construct "prefSum[]" or at last update "prefsum[]" every time for each update operation. So, for this operation it will take O(n) in the worst case.

- We know that, the number of queries can be very huge, so we cant afford O(n) time complexity too. So, here comes the concept of BIT to do the operations in logarithmic time.:

:→ A BIT (Binary Indexed Tree) or fenwick Tree is a data structure providing efficient methods for calculation and manipulation of the prefix sum of a table of values.

Fenwick Tree calculates prefix sum and modify the table in O(Logn) time.

:→ Let us consider an initial array of size N (A valid size within Integer Range).
We can easily build another array of similiar size N with values which is used to hold the "cumulative values / Prefix Sums".

Created by Hiren

Leet Code : @. hirenjoshi

Date
Page

note:- A "BIT" is represented by an Array. Generally.. that array is called "BIT array or Responsibility array".

:- The underlying working principle of the Fenwick Tree is achieved by the help of "BIT[] array". It's an array of size >= N, where each index holds some specific value / the sum of few elements (called the partial sum). Let us declare an array :- "int BIT[N]".

* Basic Idea / Intuition Behind BIT :-

- We know that each integer can be represented by the sum of powers of 2.

Ex: • $9 = 2^3 + 2^0$
$= (4) + (1) \Rightarrow (9)$

• $15 = 2^3 + 2^2 + 2^1 + 2^0$
$(4) + (4) + (4) + (1) \Rightarrow (15)$

• $37 = 2^5 + 2^2 + 2^0$
$(16) + (16) + (4) + 1 \Rightarrow (37)$

Date
Page

Binary Notation :-

$9 = \boxed{1001}$

$37 = \boxed{00100101} \rightsquigarrow \boxed{100101}$

$15 = \boxed{1001}$

- now see, if i calculate $\boxed{ceil(\log_2(N))}$ so let the result value is "X", then this "X" is actually the total number of digits (bits) in the binary representation of N.

Ex:- • $9 = \boxed{1001}$, total used bits are "4",

$ceil(\log_2(9)) = 4$ (X is "4" here)

• $37 = \boxed{100101}$, total bits are "6",

$ceil(\log_2(37)) = 6$

→ I hope, you understood. so, here we can see that the total number of digits in the binary notation of "N" is always "$\text{Log}_2 N$". This is what exactly the property being made use in BIT.

Date
Page

Similiarly, instead of storing the cumulative frequencies for the entire array, we can store the sum of some few elements (some sub frequencies) in each index. Let's understand with an example:

- let an array of size 7;

arr = | 5 | 1 | 6 | 4 | 2 | 3 | 3 | , N = 7
0 1 2 3 4 5 6

→ now suppose an "idx" be some index (not value) of the array "arr" of size N. (therefore, 0 <= idx <= N)

idx = 4 ------> [0 1 0 0]
(Binary notation of idx "4")

- now let "r" be the index of the first set bit (1) from the right side of the Binary notation.

[0 1 0 0] ---> bits
3 2 1 0 ---> indeces (right to left)
|
"r" ("r" be the index of the first set bit)
[0 <= r <= log2N]

- now we know; idx = 4, r = 2
- now apply this formula to find the "responsibility range/ The range of which's element's sum it's going to store."

- $\boxed{idx = 4}$, $\boxed{r = 2}$
- So, the range of which the sum of elements the "idx" index is going to store can be find out from this formula :-

$$\left[(idx - 2^{r} + 1) \text{ to } idx \text{ (inclusive)} \right]$$

- which means ;

$$= (4 - 2^{2} + 1)$$

$$= (4 - 4 + 1)$$

$$\Rightarrow 1 \quad \text{(let this value is "tmp")}$$

⇒ So, the responsibility range according to the given array "arr" will be :-

$$\boxed{L = tmp - 1}, \quad \boxed{R = idx - 1}$$

⇒ which makes it ≈ (L, R)

(0, 3)

tmp - 1 (1 - 1 = 0)

idx - 1 (4 - 1 = 3)

⇒ So, in BIT[] array, the "idx" → 4, will store the sum of elements within range (0, 3)

~ BIT[4] = arr[0] + arr[1] + arr[2] + arr[3]

~ BIT[4] = 5 + 1 + 6 + 4

~ **BIT[4] = 16**

→ so, this how all the indeces in BIT[] array will store the sum of some few elements from the input array "arr", using the "responsibility range formula".

→ so, after finding the sum for all the indeces in the BIT[] array, The BIT[] array for the corresponding input array "arr" will look something like this:-

BIT[N] =

5	6	6	16	2	5	3
0	1	2	3	4	5	6

~ now, inorder to handle manipulation's carefully and easily, let's reset the BIT[] array by assuming 1-based indexing.

:→ BIT[N+1] =

0	5	6	6	16	2	5	3
0	1	2	3	4	5	6	7

(idx = 4)

The imp:- This our "BIT[N+1]" array, which will be considered for the input array "arr". See, as we talked before [idx = 4], will store 16, and we could see here that's it storing the value 16.

~ Don't worry about [BIT[0] = 0], this value is always be "0" for any input array, it because we are assuming 1-based indexing in the BIT array.

~ now, here we got the BIT[] array using our "range responsibility formula".

~ now, let see "how to" find the sums of the few elements / responsibility range sums. digramatically or using the Tree diagram :-

* Diagrammatic representation / Tree diagram of the input array and its BIT :-

input array :- arr[N] =

5	1	6	4	2	3	3
0	1	2	3	4	5	6

[N = 7]

~> Now, let's create the "binary tree" structure of the "arr" array.

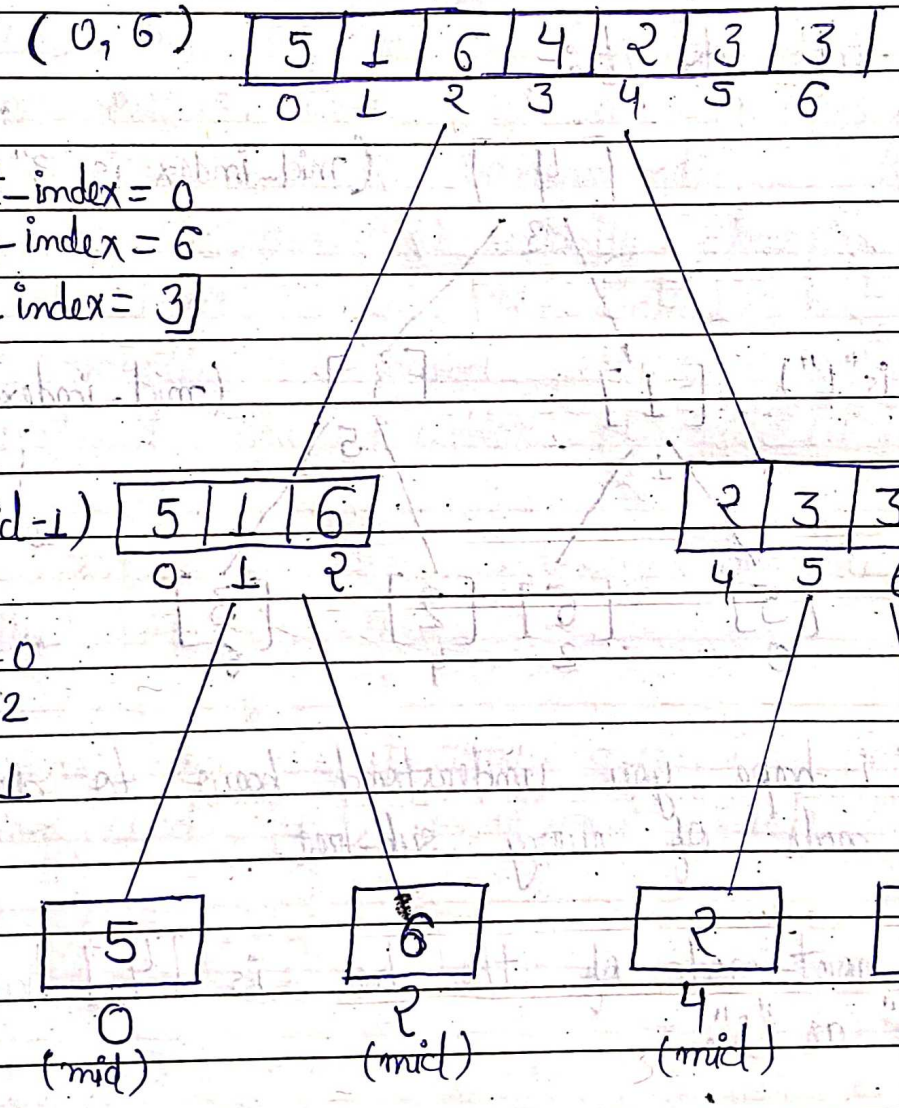
~> To create such structure, we will use the very easy intuition of "Binary search". In Binary search we used to split the search intervals into halves until the "element" gets not found.

<created by Hiren

LeetCode : @hirenjoshi

Note:- Kindly, learn the Binary search concept firstly to understand the tree digram more easily.

imp:- we will splitting the array into halves until only one element gets left. from now on, i will be telling the "index" as "node", as we are dealing with the Binary tree structure.



→ at the end, there is no more elements hence we stop here or we could say, we are splitting the array into halves until there is just one element gets left.

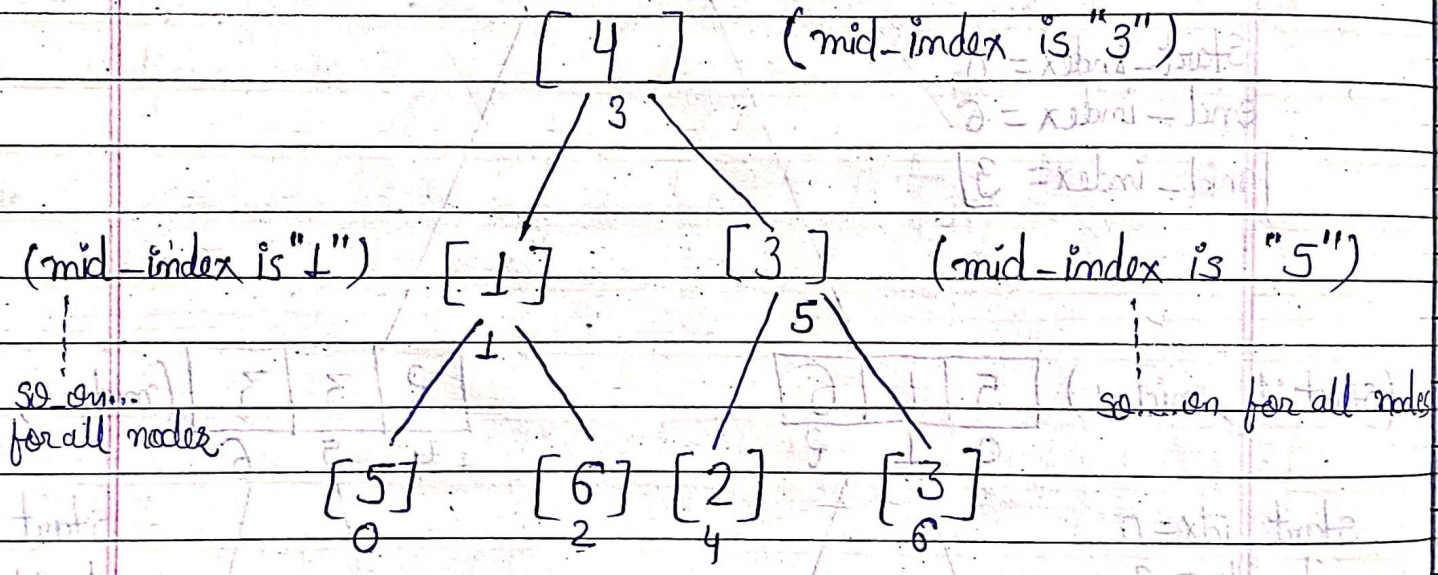
↓
not

→ now, let's create the binary tree by determining the root nodes of each subtree;

imp → which "index element" will become the root node?

:~ All the "mid-index element" will going to be the root node.

:→ now, let's do it:-



→ here, i hope you understood how to determine the root node of every sub-tree.

imp: The root-node of the tree is [4] having "mid-index" as "3".

:→ now everyone as we know, that the BIT array is of size [N+1], where N is the size of the input array "arr". So, inorder to

Created by Hiren

LeetCode: @j hirenjoshi

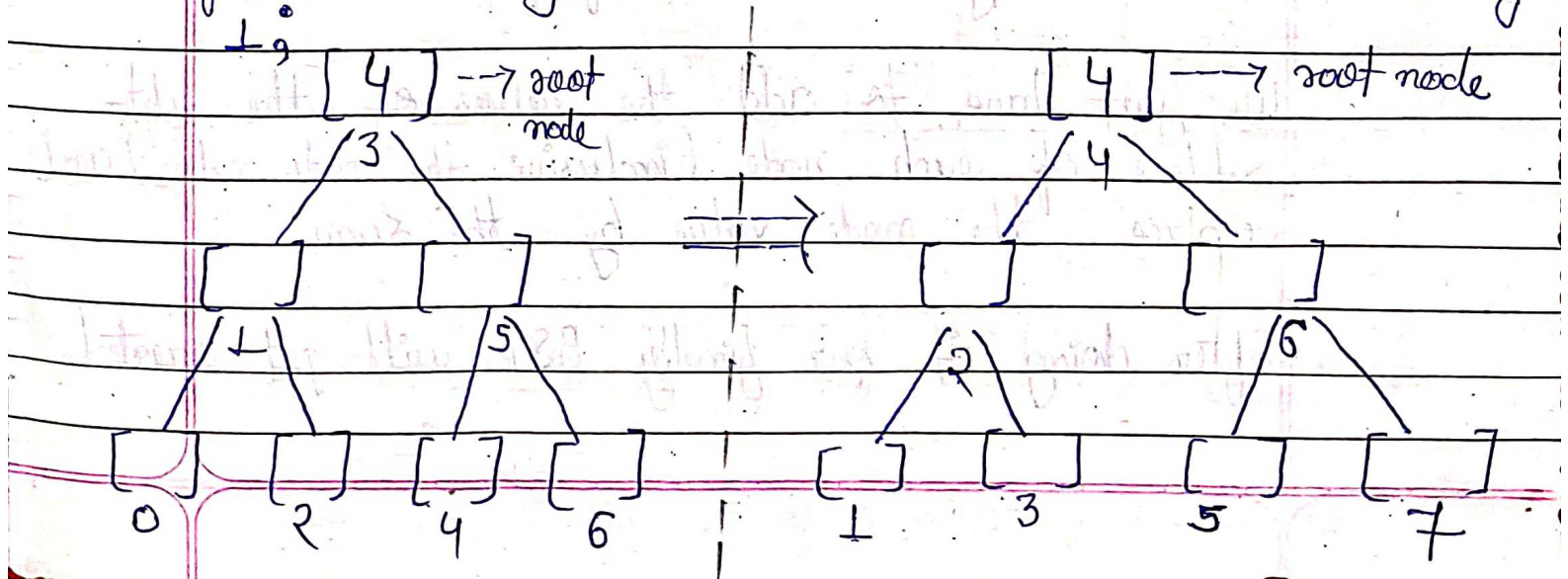
Date
Page

Create the Binary tree structure for the BIT[] array, we are considering 1-based indexing for the index elements. [I have already teached this that why we are having 1-based indexing, so no question for this now!].

~ So, let's convert the Tree-structure of input array "arr" to BIT array's Tree-structure. To do this we have to suppose 1-based indexing for the input array tree-structure. [We are assuming, so there would be no confusion in understanding BIT's Tree diagram, we are not actually changing the indeces of the input array tree-structure, but for now we are assuming 1-based indexing so that we can understand the creation of BIT[] Tree structure.]

:→ To assume 1-based indexing correctly and more easily then do this:

→ Traverse the "input array tree" in inorder traversal fashion and just increment each index value by 1;



⇝ for now, the input array tree-structure (after changing the indices) will look like this:-

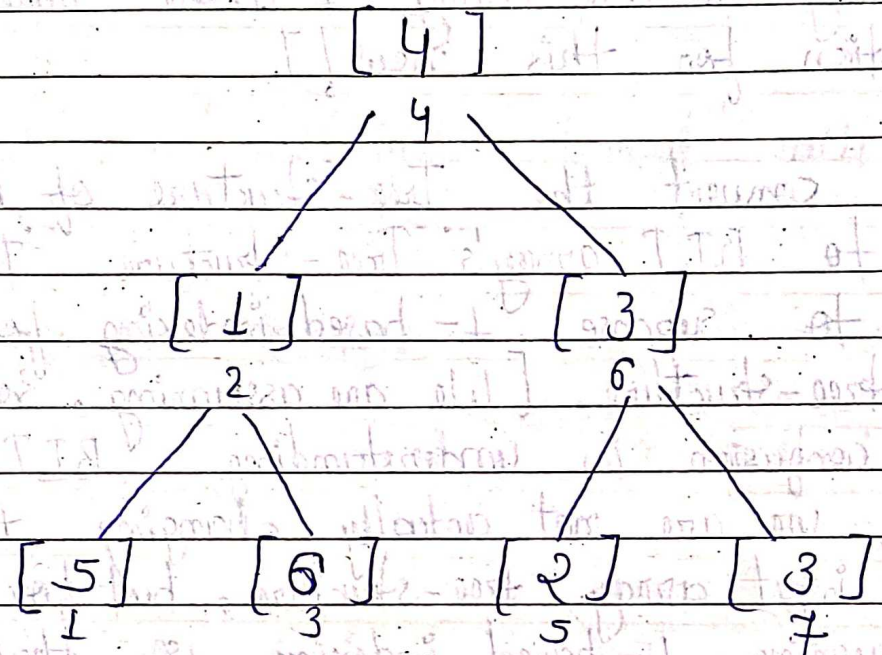


Fig ⇝ Input array's BIT's Tree-structure in process.

⇝ Now to finally convert the node values to make it containing the sums of few elements, do this:

We just have to add the values of the left-subtree of each node (inclusive the node value) and replace the node value by the sum.

- after doing it our finally BST will get created.

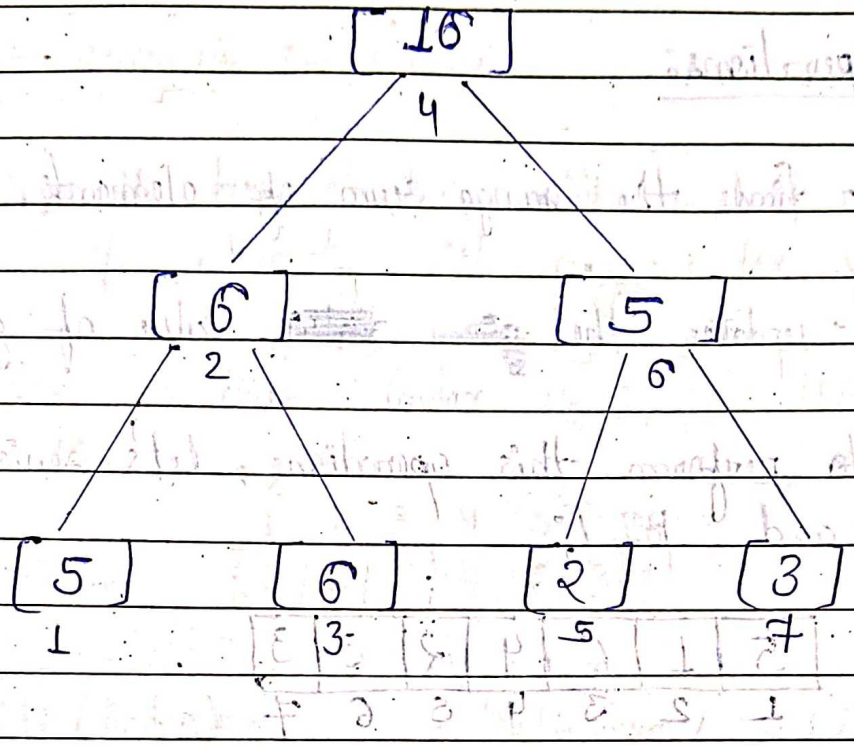


fig ⇒ final "BIT's tree" structure of the input array. (final BIT structure)

⇝ So, here we learn two ways to create the BIT of the input array:-

(i) Using the Responsibility Range formula.

(ii) Using the Tree diagram.

⇒ Input array arr[N] =

0	1	2	3	4	5	6
5	1	6	4	2	3	3

BIT[] of arr;

BIT[N+1] =

0	1	2	3	4	5	6	7
0	5	6	6	16	2	5	3

where N is the size of input array, which is 7.

<created by Hizen

LeetCode : @ hizenjoshi

Date 7-1-24
Page

* BIT Operations:-

(i) Sum - finds the range sum of elements.

(ii) update - updates the value of given index.

- Now to perform this operations, let's revisit our array and BIT;

arr =

5	1	6	4	2	3	3
1	2	3	4	5	6	7

BIT =

0	5	6	6	16	2	5	3
0	1	2	3	4	5	6	7

note:- To understand operations easily, we have to assume 1-based indexing for the input array also, But in our imagination only, I am not actually resetting the indices of the input array. This is just to understand easily, its for easy imagination.

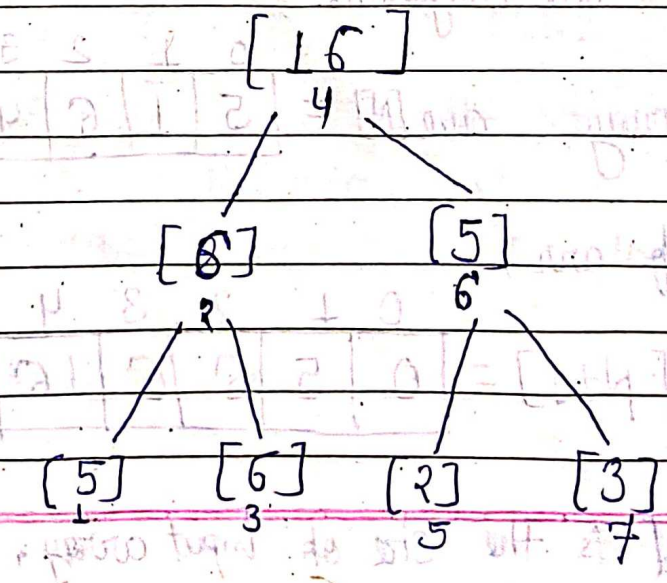


fig → Input array's BIT structure.

Created by Hisen

LeetKode : @hisenposhi

Date
Page

* (i) Sum operation :- (idea)

- Having the tree structure with us, it is helpful to find sum of elements till any index of the input array.

- suppose we have index as 3, for this the sum will be 12.

	1	2	3	4	5	6	7
arr =	5	1	6	4	2	3	3

idx (index) = 3, sum (Prefix Sum) = 12

- now follow this steps to find the sum using BIT structure:

* Create a [ans] variable initialized to 0. This is to store the prefix sum (result sum).

⇒ Start from "root" node and check the "node index" is equal to given "idx" or not :

(i) - if equal then add the node value to "ans" and return the "ans" value.

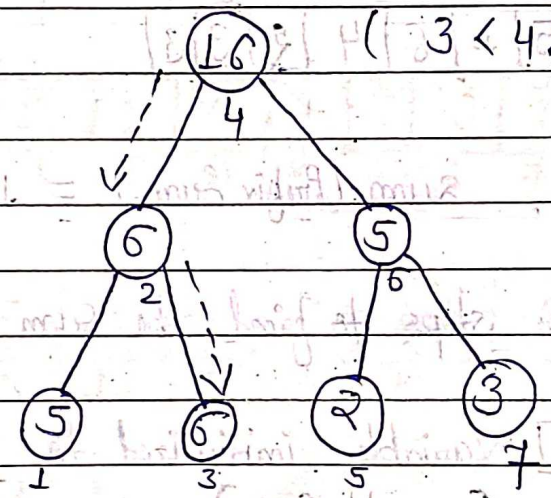
(ii) - if "root index" < "idx", then add the node value to "ans" and move to the right side of the node.

(iii) - if "root index" > "idx", then dont add the node value to "ans" and move to left side of the node.

→ Keep doing these steps until you find the node whose index is equal to the given "idx", but at last don't forget to add this node value also, i mean don't forget to do the "first step mentioned there".

→ so, lets do it:- idx = 3 , output = 12

→ ans = 0



(3 < 4, move to left)

(3 > 2, add value ans += 6, Now move right)

fig - BIT of input array

(3 == 3, add value ans += 6 so ans becoms 12 so, ans = 12, now return the "ans")

→ here we can see we just visited 3 nodes in the tree and we got our output value which is 12.

note: → So, The above explanation is just to give an idea to understand "range sum" more better, later on.

→ In "Actual implementation", we will be following the reverse path from node to root.

We will go in actual implementation too. Just have a look at update operation as well.

*(iii) Update operation :- (idea)

Suppose we want to replace the "idx" value by "K". Then the idea is very similar to sum operation.

- Follow the path from "root" to the "idx" index node, whenever we will be moving to the "left node" or "left link" then before moving add the value of K to the current node.

- Do this until we reach the "idx" node. Once we reach the node, add K to the node value too.

- We are doing updation of all the nodes which are in the path to the "idx" node, it because when we update a value in the input array at any index then of course the cumulative sum for that index will change based on the replaced value. So that's why we are updating all the nodes along in the path.

Ex → Let's understand with example:-

Created by Hisem

LeetCode: @hisemjoshi

Date
Page

input array; arr[] =

1	2	3	4	5	6	7
5	1	6	4	2	3	3

- we are choosing index (idx) as "3" to understand:

* → Before updation:- (case 1)

arr[] =

1	2	3	4	5	6	7
5	1	6	4	2	3	3

BIT[] =

0	1	2	3	4	5	6	7
0	5	6	6	16	2	5	3

→ idx = 3, sum (Prefix Sum) = 12

- So, we got the sum as 12 before updation, now let's see what happens after updation;

* → After updation:- (case 2)

→ idx = 3, k = 10

now at idx = 3, we're replacing the value from 6 to 10, now the input array becomes:-

arr[] =

1	2	3	4	5	6	7
5	1	10	4	2	3	3

- according to this "arr", now [sum (Prefix Sum) = ~~22~~].
- stick with the same old "BIT[] array" for now,

Now we are going to traverse that BIT tree to do updation;

$idx = 3$; $K = 10$

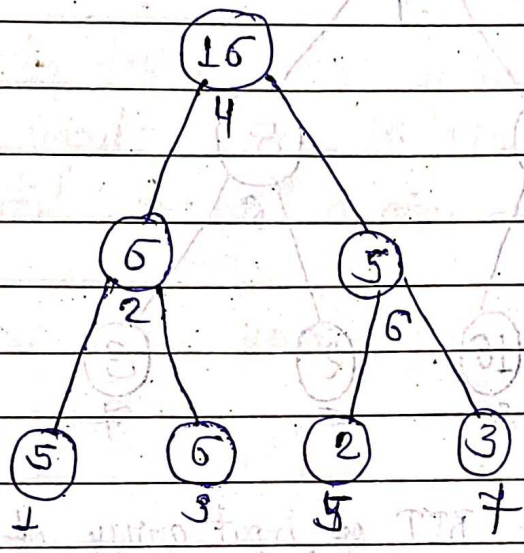


fig = BIT of input array of before updation.

→ do the below steps for updation:- (Start from root)

(i) if "node index" is equal to "idx", then add the value of K to the current node, now stop the process.

(ii) if "node index" > "idx", then add the value of K to the current node and move to left node.

(iii) if "node index" < "idx", then dont add but simply move to the right node.

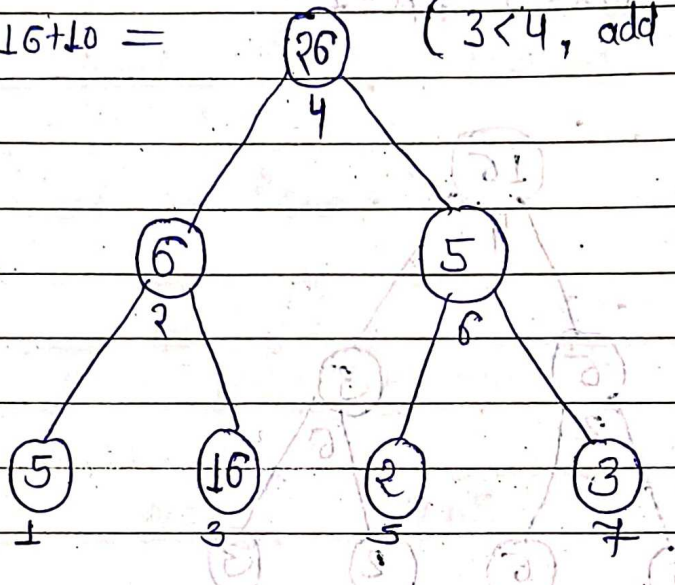
→ Keep doing the steps until you do it for "idx" node.
↓
not do it

Created by Hisen

LeetKode : @ hisenjoshi

→ lets do it; idx = 3, K = 10

16+10 =



(3<4, add K to node and move left)

(3>2, just move to right side)

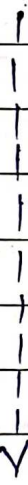


Fig:- BIT of input array of after updation.

(3=3, add K to node, 6+10 = 16. So, the new value after updation becomes 16.)

→ I hope you understood, so the input array and its BIT array of updation will look like this :-

(updated arr) arr[ ] =

5	1	16	4	2	3	3
1	2	3	4	5	6	7

(updated BIT) BIT[ ] =

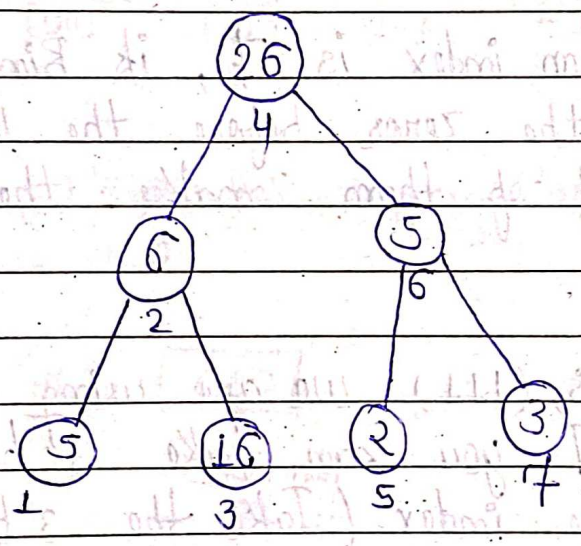
0	5	6	16	26	2	5	3
0	1	2	3	4	5	6	7

* Moving to the implementation:-

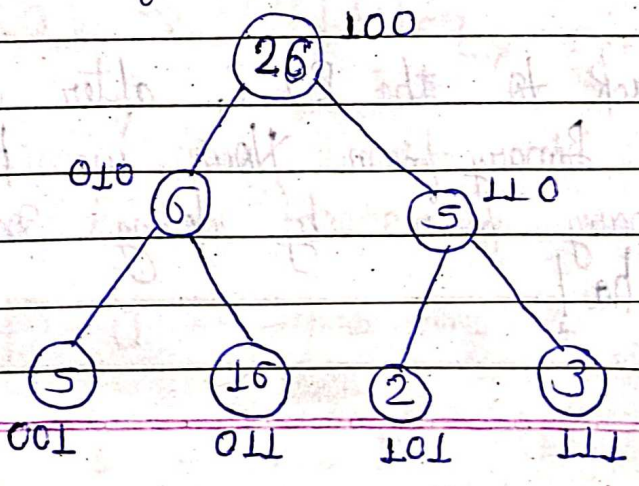
→ For this, we will play with "Binary Numbers" [BITS]
You will get it, its easy.

note:- We will be taking the updated BIT[] of the updated input array. So that i dont have to write the initial array again. OK!

~ let's draw the updated BIT[] of updated arr[]:-



→ Now everyone! change the "indeces" to their "Binary representation" for each tree node;



note:- If you are in confusion that how many bits to determine for any index's binary representation, i mean;

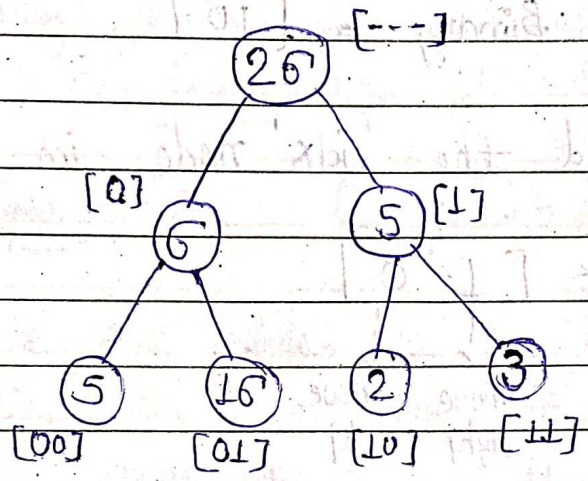
Ex - idx = 1, Binary (0001)

So, how to determine to take (0001) or (01) or (1) or (001) ?

- so, for that just find out the maximum index residing in BIT.
- The maximum index is 7, its Binary (0111), now ignore all the zeros before the leftmost set bit (1)
- ignoring all of them makes the Binary of 7 as (111).
- Now in this, (111) we are using 3 bits only. So in the BIT you can take 3 bits for all the rest of the index. (Take the 3 bits from the right side of their binary representation). Hope you got how many bits to determine for any index.

:- moving back to the BIT, after changing indices to their Binary form. Now, we have to change this Binary forms by doing something. Seems interesting ha!

- now, for each index, find the rightmost set bit, i.e. '1' and drop all the zeroes along with that '1'. now we got the BIT indeces as :-



Ex:- Did not get? see this, seppose taking the idx = 2, Binary = 010

- now in this [010].
  ↑ rightmost set bit

- remove all the 0's along with that 1, Now the Binary form becomes = [0].

Ex:- ⇒ (010) → (010) → (0) ✓
  ↑ remove part

:- So everyone! Here is the thing to be observed. In this Binary form, if we treat;

[0 — representing the left link.]

[1 — representing right link.]

Created by Hisem

LeetCode : @hisemjoshi

Date
Page

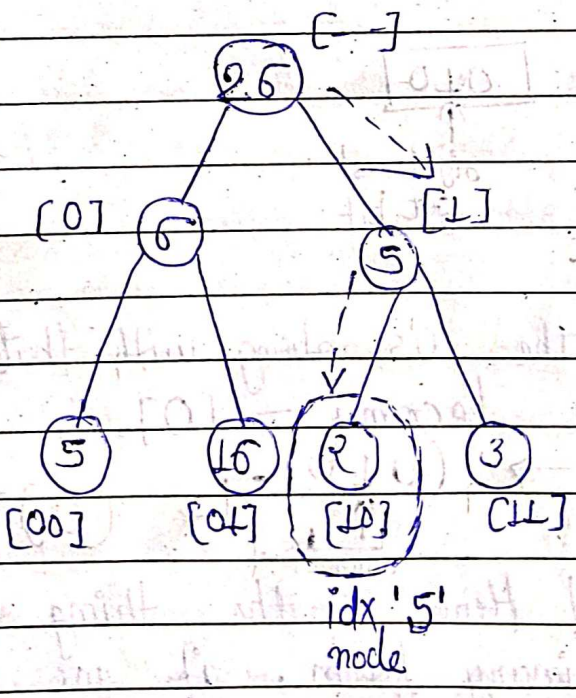
- If we observe in this way then each node tells you the path to be followed from root to reach that node.

Ex:- idx = 5 , Binary = [1 0]

- now to find the 'idx' node in the BIT, do this:

Binary = [1 0]
move right, move left

- start from root node and move to the nodes according to this observation, you will reach the 'idx' node.



'idx' = 5
Binary = [1 0]
(right, left)
(Begin from root node and move to right node.)
(now from this node move to left node, now reached the 'idx' node)

- This how, you can find any 'idx' node or can track

the path from root to that 'idx' node. The reason why this is important to us, because our Sum and Update operations depends on this path.

→ The "Binary indexed tree" does all of this super efficiently by just using the Bits in the index.

* (i) Sum Operation :- (Approach)

:- lets revise some points; To do the Sum operation we are adding the node values to "ans." from where we will be moving to the right node. We are doing this to read the cumulative sum, right!

:- in short, we need the nodes from where we will be taking the right link. So, how to get these nodes for in the approach.

:-

arr =

5	1	16	4	2	3	3
1	2	3	4	5	6	7

BIT =

0	5	6	16	26	2	5	3
0	1	2	3	4	5	6	7

let say ; idx = 5, prefixSum = 28

~ let's understand how to get the "nodes of right link" of the BIT :-

Created by Hiren

LeetCode : @hirenjoshi

Date
Page

note:- Inorder to read the "cumulative sum / Prefix sum", we just need to remove the "rightmost set bit" each time from the 'idx' index till the 'idx' value remains greater than 0;

Ex:- idx = 5, Orig. Binary form = (101)

do this;

$$101 \xrightarrow{\text{remove the rightmost 1}} 100 \xrightarrow{\text{remove the rightmost one}} 000$$

~ actually 101 and 100 are the "nodes of right links". (index of)

~ which means, 101 ---> idx 5 of BIT
100 ---> idx 4 of BIT

~ so in BIT, nodes with idx 5 and idx 4 will be the nodes whose value we are going to add in our "ans". Or we can say these index nodes are the "nodes of right links".

~ in BIT[]; idx 5 stores "2"
idx 4 is storing "26"

~ so, adding these "2+26" gives "28". See, we got the prefix sum we wanted for index 5.

- here we just visited 2 nodes only, to get the cumulative sum, isn't it really efficient!

another ex:- idx = 3, Binary form = (011)
prefix Sum = 22

do this;

011 ——→ 010 ——→ 000
(optional part)

- in BIT; (011) is node with index "3"
(010) is node with index "2"
(000) is node with index "0"

- add the values of the nodes:-

PrefixSum (idx = 3) ⇒ BIT[3] + BIT[2] + BIT[0]
(optional part, as it's storing value 0, hence adding 0 does not make difference)

PrefixSum (idx = 3) ⇒ BIT[3] + BIT[2] + BIT[0]
⇒ 16 + 6 + 0

PrefixSum ⇒ 22 or ans = 22

Created by Hiren

LeetCode : @hirenjoshi

Date
Page

:- Hope you have understood that how to perform sum operation in the approach, so this is the approach we will be using to read cumulative sum for any index of the input array.

:- So Algorithm is :-

```
int getSum ( int idx) {
    int ans = 0;
    while ( idx > 0) {
        ans += BIT[idx];
        idx = Reset_Rightmost_SetBit (idx);
    }
    return ans;
}
```

imp:- Now, you guys are wondering that how to "Reset Rightmost Set bit"? This is very easy, whenever we subtract one from any number say 'n', then the part before a right most set bit remain same while part after rightmost set bit gets inverted.

So just ANDing this will solve our problem.

```
int Reset_Rightmost_SetBIT (int n) {
    return n & (n-1);
}
```

Ex:- let [n = 3], then; [n-1 = 2]

⇒ n & (n-1)

⇒ 3 & (2)

⇒ 2

```
  0 1 1   ---> 3
& 0 1 0   ---> 2
-------
  0 1 0   ---> 2
```

```
  0 1 | 1 |
  0 1 | 0 |   -> Inverted part
```

↳ The part before rightmost set bit (1) remains same, so AND operation is solving the inverted part.

→ So, i hope you understood that how we are resetting the rightmost set bit each time or in each iteration.

→ So, this is all about the Sum Operation. Dont worry the time and space complexity. It will be taken down after the Update operation.

Vimp:- I know that you all are wondering that here we are perfectly calculating cumulative sum of

any index. But how to find the Range Sum.

- So to find the "Range Sum", we will be using the same operation twice, see here;

$$\text{rangeSum}(L, r) = \text{getSum}(r) - \text{getSum}(l-1)$$

- so this how you can calculate "range Sum queries" on an array using B.I.T. fine!

* (iii) **Update Operation :-** (Approach)

note :- Before moving to update operation, I want to tell you this, as we discussed that in Sum operation that we can "Reset the right-most set bit" by doing this ---->

$$n = n \,\&\, (n-1)$$

- so, the another alternative way to do the same, is this ---->

$$n = n - (n \,\&\, (-n))$$

- so you can use either of them :- let n as 'index';

$$\text{index} = \text{index} \,\&\, (\text{index}-1)$$

or

$$\text{index} = \text{index} - (\text{index} \,\&\, (-\text{index}))$$

:- Moving back to the update operation. Let's revisit the point, we know that for update operation, we are adding the value 'K' to the node whenever we are moving to the left node. which means we want to find the "nodes of left links" inorder to do the update operation.

→ So, inorder to get the "nodes of left link" of BIT, we have to change the "index" in this way;

$$index = index + (index \ \& \ (-index))$$

→ The above statement will give the "index of the node of left link" in each iteration.

:- So algorithm is :-

```
void update(int index, int K){
    while(index <= N){
        BIT[index] += K;
        index = index + (index & (-index));
    }
}
// Condition is while(index <= N)
```

- [while (index <= N)] ~ here N is the size of the input array.
- so as we discussed, in each iteration we will getting the "index of node of left link" of BIT.
- we are doing this until we update "all the nodes of left link of "BIT."

Example:- arr[] = | 5 | 1 | 16 | 4 | 2 | 3 | 3 |, N=7
1 2 3 4 5 6 7

BIT[] = | 0 | 5 | 6 | 16 | 26 | 2 | 5 | 3 |
0 1 2 3 4 5 6 7

(note:- assuming 1-based indexing for input array "arr". so to understand the update operation for the BIT, more easily!)

Let's do:- performing the operation;

index = 3, K = 10

- updating the value of arr[3] by adding the value of K to it; arr[3] += K, now the array becomes:-

arr[] = | 5 | 1 | 26 | 4 | 2 | 3 | 3 |
1 2 3 4 5 6 7

→ now let's update the BIT[], ah! I forgot, let's see the prefix Sum of index 3;

[index = 3, prefixSum = 32]

↳ (5+1+26)

:- now let's update BIT[]:-

index = 3, N = 7

K = 10

:- according to algorithm, we know [3<=7] so we will update BIT[3] first, add the value of K to BIT[3].

(BIT after 1st updation) ⟶ BIT[] =

0	1	2	3	4	5	6	7
0	5	6	26	26	2	5	3

:- now let's, find the "index of node of left link":-

[index = index + (index & (-index))]

⇒ index = 3 + (3 & (-3))

note:- To evaluate -3, find its 2's complement and you're done with -3.

Date
Page

→ 2's complement of +3 ⇒ (i) Write binary of 3
(ii) get 1's complement of it
(iii) add 1 to it, to get the 2's complement.

⇒ 3 ---> 0 1 1
1 0 0 (1's complement of 011)
+ 1
1 0 1 (2's complement of +3)

⇒ so, the binary of [-3 is 101].

⇒ now, index = 3 + (3 & (-3))

= 3 + (011 & 101)

⇒ so, 011
& 101
001 ---> AND gives $(001)_2 = (1)_{10}$

⇒ now, index = 3 + 1 ⇒ 4

index = 4

⇒ so, now for the next iteration, index becomes 4 which is the index of the node of left link of B.T.

⇒ now, [4<=7], so [BIT[4] += K]

⇒ so, adding K to BIT[4], now BIT becomes;

(BIT after 2nd updation) ⟹ BIT[] =

0	5	6	26	36	2	5	3
0	1	2	3	4	5	6	7

⇒ after this, the index becomes 8 and at next iteration [8<=7], the condition is not true that's why the loop breaks!

⇒ so, see we just visited 2 nodes in the BIT[] for updation. For confirmation, now let's find out the prefix sum of index 3 using the updated BIT[].

⇒ we already learn, how to find the sum in BIT. So do it on your own and after that you will see that we are getting the prefix sum as 32 for index 3.

⇒ Hope, you understood the update operation!

∴ Now, I know that you are wondering that how to build the BIT, for that we will use the Update operation. Have a look at the whole code below and you will get it for sure.

Created by Hisem

LeetCode : @_hisemjoshi

Date
Page

* Code :-

// Program to implement the Fenwick Tree along with the operations ~ coded by Hisem

```
#include <iostream>
# include <vector>
using namespace std;

//Time: O(LogN) | [redacted] Space: O(N)

void update (vector<int>& BIT, int N, int idx, int val)
{
    while (idx <=N) {

        BIT [idx] += val;
        index = index + (index & (-index));

    } // The name 'index' in the statement is
      mistakenly written. It's actually 'idx'.
}
```

Created by Hiren

LeetCode : @ hirenjoshi

```
// Time : O(LogN) | [redacted] space : O(N)
int getSum (vector<int> &BIT, int N, int idx) {
    int ans = 0;
    while (idx > 0) {
        ans += BIT[idx];
        index = index - (index & (-index));
    } // 'index' is actually 'idx'.
    return ans;
}

// Driver code
int main () {
    int N;
    cout << "Enter the size of the array : ";
    cin >> N;

    vector <int> inputArray (N+1, 0);

    cout << "Enter the array elements : ";
    // Update array elements.
    for (int i=1; i<=N; ++i) {
        cin >> inputArray[i];
    }
```

Created by Hiren

LeetCode : @hirenjoshi

```
// "BIT" array to represent the "binary indexed tree".
vector<int> BIT (N+1, 0);

// Constaction of BIT | Time: O(NLogN) | space: O(N)
for(int i=1; i<=N; ++i){

        update(BIT, N, i, inputArray[i]);

}

// section to hit the prefix sum query.
int idx = 4;

if (idx >= 1 && idx <= N){

        int prefixSum = getSum (BIT, N, idx);

    cout<<"\n The prefix sum of index "<< idx
        <<" is : "<< prefixSum;

"}

// section to hit the range sum query.
int L=1, R=4;
if ( L>=1 && R<=N){
    int rangeSum = getSum(BIT, N, R) - getSum(BIT, N, L-1);
    cout<<"\n The sum of range is : "<< rangeSum;
}
return 0;}
```

Note:- "Do urn the code yourself to understand very better." (DRY Run it.)

* Time and Space complexity :-

:- In the worst case, the getSum() and update() method can take $Log_2 N$ Time.

:- $O(Log_2 N)$ because, if you observe the number of iterations made in both the methods can at most equal to the value of $Log_2 N$.

:- Hence both the operations will take $O(Log_2 N)$ time.

:- The overall space here is $O(N+1) = O(N)$, which is taken by the "BIT" array.

* Applications of BIT :-

(i) BIT's are used to implement the arithmetic coding algorithm.

(ii) BIT's can be used to count inversions in an array in $O(N Log N)$ time.

(iii) Practially, BIT's are used very much in competitive programming.

Created by Hiren.

LeetCode: @hirenjoshi

Date
Page

note:- last but not the least! Haha, i forgot this to tell you is that;

- "Peter M. Fenwick" first described the "Binary Indexed Tree" in 1994 in a paper titled: "A new data structure for cumulative frequency tables."

- The "BIT" is also known as "Fenwick Tree".

Must Read:- I tried my best to cover "Fenwick Tree" for you. My notes include everything on the "Fenwick Tree".

If you want to appreciate my hard work then you can share this notes with your friends! Thank you :)

--> connect with me on LinkedIn : hiren joshi 1630

--> visit me on Leet Code : hirenjoshi

:) Happy Coding :)