← **Notes**

▲ Binary Indexed Tree made Easy

54   Bit   binary-indexed-tree   Fenwick-tree

For the past few days, I have been reading various explanations of the Binary Indexed Tree. For some reason, none of the explanations were doing it for me. All explanations told me the same thing over and over again. I was not able to find the motive behind this data structure, intuition behind this data structure.

Finally, I decided to sit down, check some examples, diagram them out, check stack overflow and understand it. I now understand the beauty of this data structure, and I think, I can explain it. For those who have gone through this and also for those who don't want to go through this phase, I am writing this post..

Let me tell you one thing, this is going to be a longer post. I will try to cover all the things associated with it. I have included examples for understanding. Give it half an hour, you will surely get new thing to learn.

Wasting no time, lets have a well defined problem.

```
We will be given an array. And we are asked to answer few queries
Queries will be of two types:-
1) Update X Y : Increment value at Xth index by Y.
2) Sum L R : Print sum of values at index L to R inclusive.
```

Lets have a look at other approaches in short, before going for BIT (Binary Indexed Tree), so that you will know the need of BIT.

1. We can **update** any value in the array in single step. So, update operation will need $O(1)$ time. Also, for **sum** operation, we can traverse the array and find sum. That operation will take $O(n)$ time in worst case.

2. One more thing we can do. At each index, we will store the cumulative frequency i.e. we will store sum of all elements before it including itself. We can construct this new array in $O(n)$. Lets say this array as CF[]. After that, All the **sum** operation will take $O(1)$ time since we will just subtract CF[L-1] from CF[R] to get the answer for sum L R. But well, we will need to construct CF[] or at least update CF[] every-time update operation is made. The worst case time required for this will be $O(n)$.

Since, the queries are huge in number, we can not always afford `0(n)` time complexity too. So, here comes the BIT for our rescue.

BINARY INDEXED TREE **or** FENWICK TREE

CONSTRUCTION of BIT:

Lets have an example with us. Input array is:

```
[ 5 ] [ 1 ] [ 6 ] [ 4 ] [ 2 ] [ 3 ] [ 3 ]
   1       2       3       4       5       6       7
```

Now, think of what we did in 2nd approach. For each index, we were storing sum of all elements before that element to that index. Right? But because of that, we were needed to change values at all locations for every update.

Now think it this way, what if we store sum of some elements at each index? i.e. Each index will store sum of some elements the number may vary. Similarly, for update operation also, we will need to update only few values, not all. We will see how!

Formally, we will create some benchmarks. Each benchmark will store sum of all elements before that element; but other than those benchmarks, no other point will store sum of all elements, they will store sum of few elements. Okay, if we can do this, what we will need to do to get sum at any point is - intelligently choosing right combination of positions so as to get sum of all elements before that point. And then we will extend it to sum of elements from L to R (for this, the approach will be same as we did in second approach). We will see that afterwards.

Now, having done the base work, lets move ahead.

Before telling HOW will we be doing, I would like to tell you WHAT are we going to do. To remind you, we are going to create BIT[] of given input array.

WHAT:

This is a kind of manual process I am showing.

The benchmarks I was talking about are the powers of 2. Each index, if it is a power of 2, will store the sum of all elements before that. And we will apply this repetitively so as to get what each index will store.

Suppose, we have an array of 16 elements, [1 .. 16].

Powers of 2 are:- 1, 2, 4, 8, 16

These index will store sum of all elements before them.

Fine?

What about others?

Divide this array in two halves:- we get [1..8] and [9 .. 16].

Now think recursively what we did for array of 16, apply same for this, okay?

Seems like little bouncer? Wait, have an example of 8 elements only. Say 8 elements are :

1	2	3	4	5	6	7	8

Ok, powers of 2 are: 1 2 4 8 so, in BIT[] indiced 1 2 4 8 will store 1 = 1, 1 + 2 =3, 1 + 2 + .. + 4 = 10 and 1 + 2 + .. + 8 = 36 respectively. Right? Remember, sum of all elements before that element? Right? Good. So, till now, BIT looks like this:-

```
[ 1 ] [ 3 ] [   ] [ 10 ] [    ] [    ] [   ] [36]
   1      2     3      4      5      6      7      8
```

Now, divide the given array in 2 halves.

Arr1:

1	2	3	4

Arr2:

5	6	7	8

Consider Arr1 first. Powers of 2 are:- 1 2 4 They already have their right values, no need to update.

Now, Consider Arr2: Powers of 2 are: 1 2 4

So, at indices 1, 2 and 4 will store 5 = 5, 5 + 6 = 11, 5 + 6 + 7 + 8 = 26 respectively.

These are the indices according to this new 4-element array. So, actual indices with respect to original array are 4+1, 4+2, 4+4 i.e. 5, 6, 8. We will not care about index 8 as it is already filled in BIT[]. Hence we will update position 5 and 6 now.

BIT will start looking like this now:-

```
[ 1 ] [ 3 ] [   ] [ 10 ] [ 5  ] [  11  ] [   ] [36]
   1      2     3      4      5       6       7      8
```

I think you guys have got what we are doing. Applying same procedure on Arr1 and Arr2, we will get 4 - two element arrays (2 from Arr1 and 2 from Arr2). Follow the same procedure, don't change the value at any index if it is already filled, you get this BIT finally.

```
[ 1 ] [ 3 ] [ 3 ] [ 10 ] [ 5   ] [ 11 ] [ 7 ] [36]
    1       2       3       4         5         6         7         8
```

Guys, do take an example of 16 element array and convert it to BIT manually to get the gist.

Now see how will we do this in program.

HOW:

We will continue with our previous example.

Now, start thinking of our array as a binary tree, like this:-

```
BEFORE:
[ 5 ] [ 1 ] [ 6 ] [ 4 ] [   2 ] [   3 ] [ 3 ]
    1       2       3       4       5         6         7


NOW:


            4
          [ 4 ]
         /        \
       2            6
     [ 1 ]        [ 3 ]
     / \          / \
    1   3        5   7
   [5] [6]      [2] [3]
```

Now, we will change value at each node by adding the sum of nodes in its **left sub-tree**.

```
UPDATED VERSION:


            4
          [ 16 ]
         /        \
       2            6
```

```
    [ 6 ]      [ 5 ]
     / \        / \
    1   3    5    7
   [5] [6]   [2] [3]
```

I think you have got what we have just done! Take each node, find sum of all nodes in its left sub-tree and add it to value of that node. And this is what we call is BIT.

```
BIT:
[ 5 ] [ 6 ] [ 6 ] [ 16 ] [  2 ] [  5 ] [ 3 ]
   1     2     3     4       5       6      7
```

SUM and UPDATE operations:

Now, we have got the BIT. Lets move ahead and solve our real problem.

Having this tree structure with us, it is to find sum of elements till any index. The idea is to keep a variable `ans` initialized to 0. Follow the path from root to the `index` node. Whenever we need to follow a right link, add the value of current node to `ans`. Once we reach the node, add that value too.

For example, If we want sum of elements upto index 3.

See again,

```
INPUT ARRAY is:
[ 5 ] [ 1 ] [ 6 ] [ 4 ] [  2 ] [  3 ] [ 3 ]
   1     2     3     4      5       6      7
```

(so answer should come out as 5 + 1 + 6 = 12)

```
BIT is:

          4
        [ 16 ]
       /      \
      2        6
    [ 6 ]    [ 5 ]
     / \      / \
    1   3    5   7
   [5] [6]  [2] [3]
```

Following the procedure given above.

```
1> node = root, ans = 0
2> node is 4, index is 3.
3> index < node, go left
4> node = 2, index is 3.
5> index > node, add value(node) to ans and go right
       i.e. ans = ans + value(node 2)
       i.e. ans = 0 + 6
       i.e. ans = 6
       Now, go right
6> node = 3, index = 3
7> node == index, add value of node 3 to ans and return ans
       i.e. ans = ans + 6
       i.e. ans = 12
       return 12
```

In actual implementation, we will be following the reverse path i.e. from node to root.

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

If we want to increment the value at `index` by say `k` .

The idea is very similar to sum operation.

Follow the path from root to the `index` node. Whenever we need to follow a left link, add the value of `k` to current node. Once we reach the node, add `k` to that node too. This is because we will need to update the set of nodes in the tree that include that node in its left subtree, so that it will be consistent with our sum operation, right?

I don't think there is any need of example for this case again.

Moving ahead to the implementation.

For this, we will play a bit with **BITS -- Binary Numbers**. Here comes the fun with bits -- Binary numbers. You will have to do little more work here to figure out the things. I will try my best though.

```
TREE is:
         100
       [ 16 ]
      /      \
    010        110
  [ 6 ]      [ 5 ]
```

```
       / \       / \
     001   011   101   111
    [5]    [6]   [2]    [3]
```

We have just changed the representation of out indices to binary. Ok?

Now, For each index, find the right most SET-bit i.e. '1' and drop the all zeros along with that '1'. We get,

```
            ( - - )
            [ 16 ]
          /         \
       0             1
     [ 6 ]         [ 5 ]
      / \           / \
    00    01      10    11
   [5]   [6]     [2]   [3]
```

Here is the thing to be observed. If we treat 0 as LEFT and 1 as RIGHT, each node tells you the path to be followed from root to reach that node. Really? Have example, say node 5, which has 10 there, i.e. RIGHT and LEFT. This is the path we need to follow from root to 5. Cool thing, right?

The reason why this is important to us is, our **sum** and **update** operations depends on the this path. Are they not? You remember, Left link, right link, right? During a **sum**, we just care about the left links we follow. During an **update**, we just care about the right links we follow. This binary indexed tree does all of this super efficiently by just using the bits in the index.

The key thing behind the efficiency of BIT is:

> Given any index n, the next node on the path from root to that index where we go right is directly calculated by RESETing i.e. '0' the last (right most) SET-bit from the binary representation of `index` . Apply this until we reach the root.
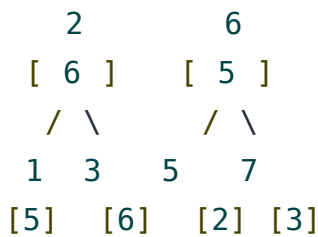
Lets have examples:

```
TREE is:
            4
        [ 16 ]
      /         \
```

```
    2        6
  [ 6 ]    [ 5 ]
   / \      / \
  1  3    5   7
 [5]  [6]   [2] [3]
```

Say index is 5. The path from 4 to 5 is [4 -> RIGHT -> 6 -> LEFT -> 5] i.e. we take RIGHT at 4. Binary Representation of 5 is 101. RESET right-most SET-bit. 101 -> 100 4 is the one node from where we will go right STOP here. We have reached the root.

Say index is 7. The path from 4 to 7 is [4 -> RIGHT -> 6 -> RIGHT -> 7] i.e. we take RIGHT at 4 and 6. Binary Representation of 7 is 111. RESET right-most SET-bit. 111 -> 110 6 is the node from where we will go right RESET right-most SET-bit. 110 -> 100 4 is the node from where we will go right STOP here. We have reached the root.

We will use this information in our implementation.

Implementation:

Now we know, how to go from any index to the root and find what all right-links come in our path.

I will repeat some part of what we have looked.

> For SUM: The idea is to keep a variable `ans` initialized to 0. Follow the path from root to the `index` node. Whenever we need to follow a right link, add the value of current node to `ans` . Once we reach the node, add that value too.
>
> For UPDATE: Follow the path from root to the `index` node. Whenever we need to follow a left link, add the value of `k` to current node. Once we reach the node, add `k` to that node too.

Now you have got the complete picture I guess. Everything of What we saw, Why we saw?

For SUM, We need to follow RIGHT-links no matter from root to index or reverse. And we also know how to do that. Right?

So algorithm is:

```
SUM(index):
    ans = 0
    while(index != 0):
```

```
        ans += BIT[index]
        index = Reset_Rightmost_SET_bit(index)
    return ans
```

Now, the thing remain unanswered is: How to Reset rightmost SETbit? This is a very simple task which I have already covered in my this note. By some observations, we can arrive at a conclusion that, whenever we subtract one from any number say n, the part before a right-most set bit remain same while part after right-most set bit gets inverted. So, just ANDing these can solve our problem.

```
Reset_Rightmost_SET_bit(n):
    return n&(n-1)
```

Please be focused and try to understand this.

We need all left links but we can only know right links with the technique we studied earlier.

We know that, dropping the right-most SET bit and part after that gives us the path from root to node.

So, zeros which come after the right-most one are not useful to us at all.

We will use both these fact and try to find a way.

You must have observed, what happens when we add a 1 to right-most SET bit of a number? [Consider scan from right to left]

1. The first zero from right (which will come after i.e. left to, right-most ONE of number) turns into one

2. Part after (i.e. left to) that ZERO remain unchanged and Part before that get inverted.

Is this not the exact reverse procedure of what used to happen in SUM operation.

This is all what we wanted!

And this is the value i.e. index from which we needed to take the left link to reach to our node from root.

We have successfully found the left link too.

Adding one to right most one is nothing but adding place value of right-most ONE to the number.

Hence our Update operation is as simple as:

```
UPDATE(index, addition):
    while(index < length_of_array):
        BIT[index] += addition
        index = index + (index & -index)
```

Try to check the similarity and difference, and you will never forget again.

Here I will stop. I guess you have everything what you need to know about Binary Indexed Tree as a data structure. Now I advice you to implement it yourself and see if you can do it.

You can always refer to the code which I am providing you.

```python
# About using the 2 functions:-
# For update, pass index of location to be updated, input array, b
# i.e. new value - original value
# For getting sum of elements in range l to r,
# Getsum returns sum of elements from beginning to index
# Pass index, input array & BIT to function
# getsum of l to r = getsum of r - getsum of (l-1)

def update(index, a, tree, value):
# index is index to be updated, a is input array / list, tree is
# number at index location
    add = value
    n = len(a)
    while index<n:
        tree[index] += add
        index = index + (index & (-index))

def getsum(index, a, tree):
# index is location upto which you want the sum of elements from b
# tree is BIT[], a is input array / list
    n = len(a)
    ans  = 0
    while(index>0):
        ans += tree[index]
        index = index - (index & (-index))
    return ans


#Get the user input
n = int(raw_input("Number of Elements in array: "))
```

```
inputArray =
inputArray.in                                              /: ").split(
                                                        node to ha
#Initialise B                                       t input arra
BIT = []
for i in range(
    BIT.append(

# Now we will c
# The 4th parame                                    h is to be
# since we have                                     hile initial
# value
for i in range(
    update(i, inputArray, BIT, inputArray[i])
```

If you like this, Let me know :) Like, Share, Upvote[at the top]!!

Thank you for reading and also thanks for your patience.

P.S.- The same problem can be solved using segment tree data structure as well. I will be covering this in my next post.

HAve look at my other notes here.

Tweet   Tweet   G+1   4

COMMENTS   ⟳ Refresh

Join Discussion...

Cancel   Post

gaurav bajpai Mar 16 '2015 at 11:48PM

in one word....its exceptional!!!

Reply ● Message ● Permalink

- ○

Bhavesh Munot ⚡ Author Mar 17 '2015 at 12:37PM

Thank you so much for your precious response!

Reply ● Message ● Permalink

-

Karan Dev Mar 19 '2015 at 02:47PM

I could not understand how binary trees are made using the array ? Which approach you used here ?

Reply ● Message ● Permalink

-

Neeraj Gupta Mar 30 '2015 at 05:21PM

OH NOW I GOT THE BASIC CONCEPT OF BINARY INDEX TREE!!! THANK YOU SO MUCH!!! and keep editing :)

Reply ● Message ● Permalink

-

ankur malik Apr 08 '2015 at 06:11PM

that is a great note... just a small doubt @Bhavesh Munot , am a little confuse.. I mean considering
'a binary tree with each node containing sum of its children nodes and leaf nodes as the actual elements of the array'
I cant't figure out the difference between this and the BIT (concept wise, am not too good with implementations)..

Reply ● Message ● Permalink

-

Siddharth Rajan May 20 '2015 at 09:28PM

Really a nice one !!! Keep up the good work!!

Reply ● Message ● Permalink

🪄 AUTHOR

Bhavesh Munot

💼 System Software Engineer ...
📍 Pune, Maharashtra, India
📄 9 notes

Write Note

My Notes

Drafts

TRENDING NOTES

Trie, Suffix Tree, Suffix Array
written by pkacprzak

Technique to play online Bot games
written by Catalin Stefan Tiseanu

Number Theory - III
written by Boris Sokolov

Exact String Matching Algorithms
written by Alei Reyes

Binary Indexed Tree or Fenwick Tree
written by Chandan Mittal

more ...

ABOUT US

Blog

Engineering Blog

Updates & Releases

Team

Careers

In the Press

HACKEREARTH

API

Chrome Extension

CodeTable

HackerEarth Academy

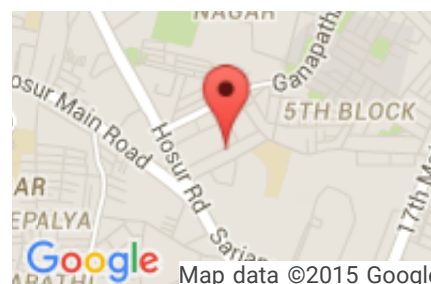Developer Profile

Resume

Campus Ambassadors

Get Me Hired

Privacy

Terms of Service

DEVELOPERS

AMA

Code Monk

Judge Environment

Solution Guide

Problem Setter Guide

Practice Problems

HackerEarth Challenges

College Challenges

RECRUIT

Developer Sourcing

Lateral Hiring

Campus Hiring

FAQs

Customers

Annual Report

REACH US

Map data ©2015 Google

IIIrd Floor, Salarpuria Business Center,

4th B Cross Road, 5th A Block,

Koramangala Industrial Layout,

Bangalore, Karnataka 560095, India.