See it again
And understand

Fenwick Tree

or

Binary Indexed Tree

Both Same.

{ With only notes
it is difficult to understand
the working Logic }

(#) Fenwick Tree or Binary Index Tree

(Ex) Let us consider a problem

We have an array [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.

⇒ So a very very better soln is Binary Indexed Tree, which achieves O(logn) time complexity for both operations.

⇒ Binary Indexed Tree is represented as an array. Let array be BITree[].

⇒ Number of functions do be implemented

(1) constructBITree(int arr[], int n).

(2) updateBIT(int n, int index, int val).

(3) getSum(int index) → from index 0 to some index 'index'.

⇒ Also if you want to range sum i.e. from index l to index r then we use as

5 stars: rangeSum(l,r) = getSum(r) - getSum(l-1) ← like in segment tree.

⇒ This can be also solved by using segment tree.

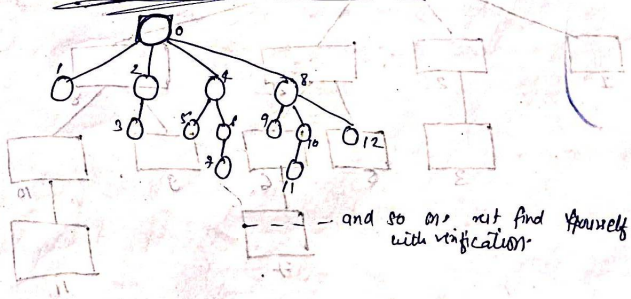
⇒ But since Binary Index Tree require less space and is easier to implement than segment tree.

⇒ So Time Complexity to build Binary Index tree is O(nlogn).

⇒ Time to getSum = O(logn)

⇒ Time to update query = O(logn).

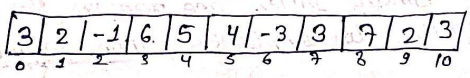
⇒ However Binary Index tree sequence goes as :-



- and so on, wit find yourself sequence with verification.

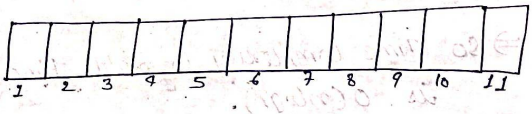
(5) Implementation of Fenwick Tree :-

e.g. given input array

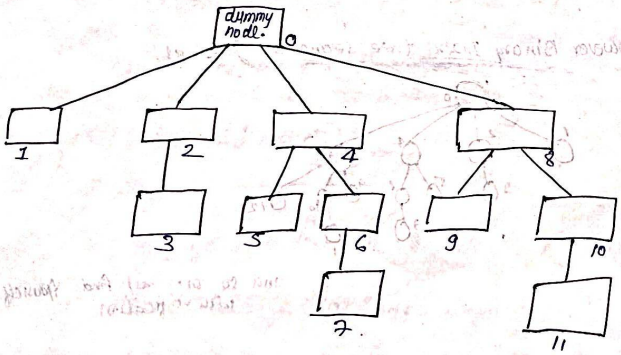


← input array.

BITree[] ⇒



Our conceptual fenwick tree will look like.



⇒ '0' is dummy node, it does not store any information. From 1 to 11 will stored the prefix sum.

⇒ Now why our BITree is like this. i.e. why '0' is parent of '2'.

Parent so

so take '2' (10) $\xrightarrow[\text{set bit}]{\text{flip rightmost}}$ 00 ⟶ '0'

for '8' (1000) $\xrightarrow[\text{set bit}]{\text{flip rightmost}}$ 0000 ⟶ '0'

for '10' (1010) $\xrightarrow[\text{set bit}]{\text{flip rightmost}}$ 1000 ⟶ '8'

for '11' (1011) $\xrightarrow[\text{set bit}]{\text{flip rightmost}}$ 1010 ⟶ '10'.

50 ★★★

⇒ so now it's clear why our tree looks like this.

So Now How To Get Parent (In coding) :-

Get Parent :-

1. 2's complement
2. AND it with original number
3. Subtract from original number.

code as

```
index -= index & (-index);
```

(so this line is doing 2's comp.)

```
index = index - (index & (-index));
```

e.g. to get parent of index 7.

Step 1: '7' → 111 → 1's complement (Just flip the bits) → 000 → 2's complement (Just Add 1) → 001

So 2's complement = 000 + 1 = 001.

Step 2: Now AND with original 111.

∴
```
  1 1 1
  0 0 1
-------
  0 0 1
```

Step 3: Now subtract this result from original

So :-
```
  1 1 1
- 0 0 1
-------
  1 1 0
```
→ '6'

∴ 7's parent is '6'.

for '10' → 1010

2's complement = 0101 + 1 = 0110

AND :-
```
  1 0 1 0
  0 1 1 0
-----------
  0 0 1 0
```

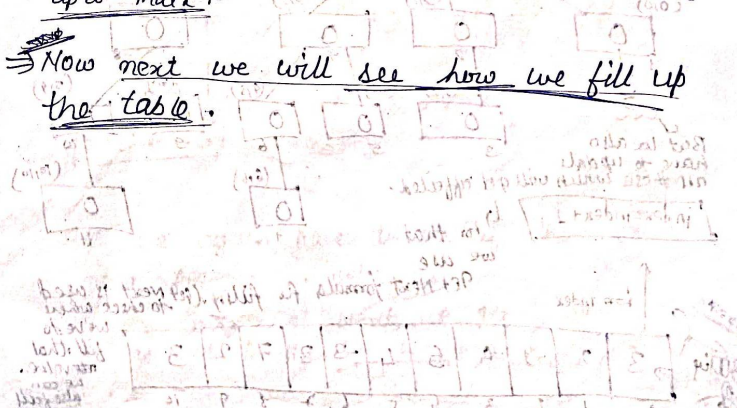
Subtract =
```
  1 0 1 0
- 0 0 1 0
-----------
  1 0 0 0
```
→ '8'

∴ 10's parent is 8.

⇒ So to get parent we've to do this three steps which is O(1) operation.

⇒ So this formula of Get Parent is used in GetSum() function. It adds from leaf to parent to getsum upto 'index i'.

⇒ Now next we will see how we fill up the table.



⑤ Now how to fill Binary Index tree from Array in efficient time :-

⇒ Get Next (this is used for filling array values in the tree, we can fill each value in multiple nodes in tree so for that we use GetNext function)

1. 2's complement
2. AND with original number
3. Add to original number.

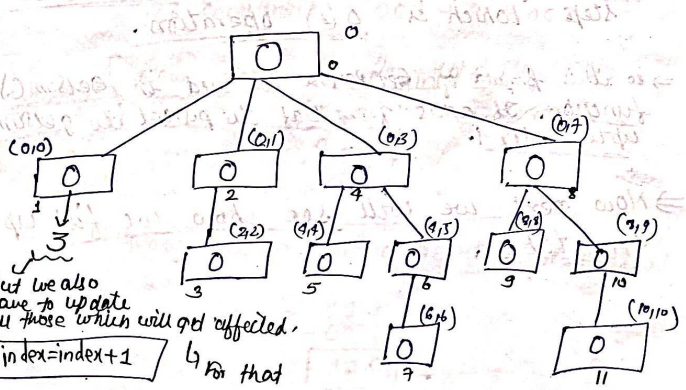
In code as

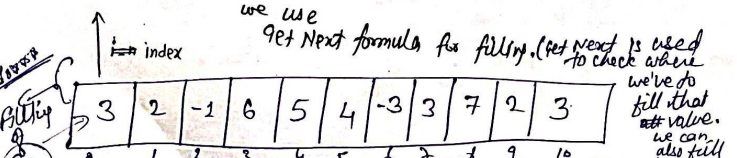
```
Index += index & (-index);
index = index + (index & (-index));
```

element

⇒ GetNext() function is used to fill arr, to the index & there parent tree form.

⇒ originally our BITree will be all storing '0' values so now we've to fill our array in it.



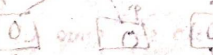


⑤ so index = 1 $\xrightarrow{\text{binary rep.}}$ 00000001

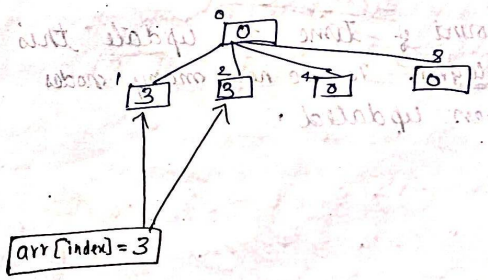
2's complement → 11111110 + 1 → 11111111

AND with original ⇒
```
  00000001
& 11111111
----------
  00000001
```

ADD to original ⇒ 00000001 + 00000001 ⇒ 00000010 ⇒ "2"

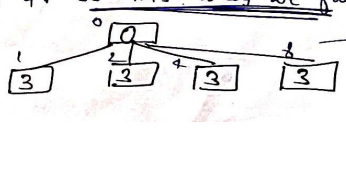


So, we've to also add index ~~=index~~ from arr i.e [3] → not only in 1st but also in 2nd position also

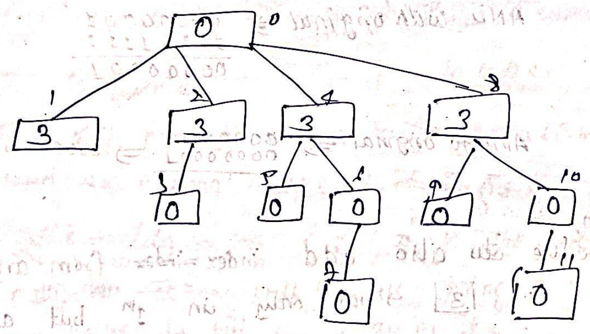


Also we again have to see if there exist next for 2 or not.

so using Get Next we get 4 as well as 8 as next for 4. so this way we fill.



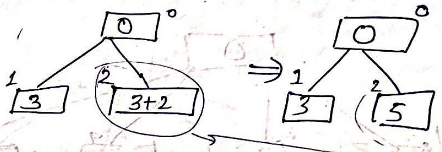
⇒ So we have added arr[0] in our Binary Index tree.



⇒ So amount of time to update this is O(logn). To see how many nodes has been updated.

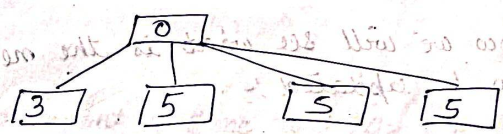
⇒ So now come to arr[1] = 2
index = index + 1
so BIT index=2.

So



So now we also have to update next of '2'.

next of '2' is '4'
next of '4' is '8'. so we get as:-

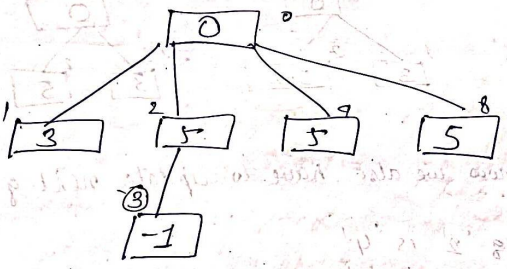


If there is already element present then just add current array value in that node.

⇒ so now come to arr[2] = -1

index = index + 1

index = 3 in BIT.



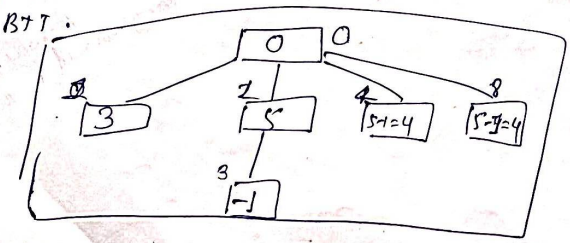
now we will see what is the next number to be updated :-

So next of 8 is → '4'

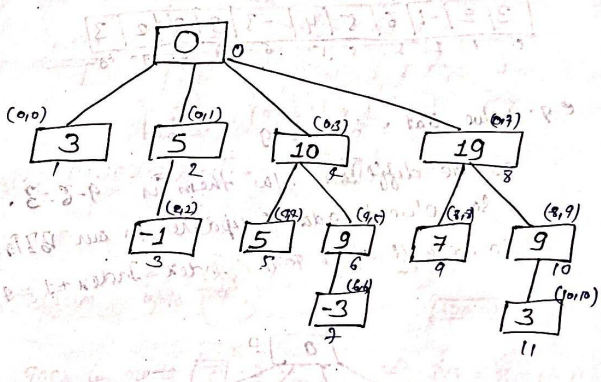
& 4's next is → '8'

so we've to add -1 in node '4' & '8' also. altho since '8' next is outside range so we stop.

∴ our BIT :



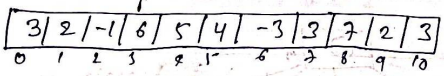
and so in this way our BIT tree is updated in its array and our final BIT tree will be as :-



⇒ so the amount of time to create this Binary Index tree is O(nlogn).

⇒ So now we will see how to update a value.

For e.g. 9

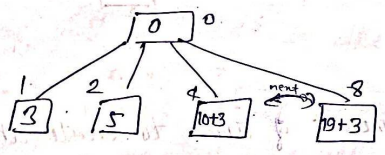


e.g. value 6 at 5 becomes 9

So the difference b/w them is 9-6=3.

So we've to add 3 update in our BITree.

So we will start from Index = Index+1 = 4.



So the amount of time it will take to update is O(log n).

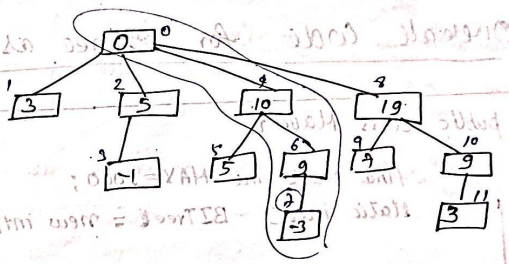
9's next '8' to 8 also add +3.

8's next not 8 range so stop.

So this way update works.

(3) Now our way to getSum :-

our BITree as :-



for getSum upto '7'. sum=0

goes to node 7 ⇒ -3 ⇒ sum = sum+(-3)

↓ its parent (get parent) i=i-i&(-i)

6 ⇒ 9 ⇒ sum = sum+(-3)+(9)

↓ its parent (get parent)

4 ⇒ 10 ⇒ sum = sum+(-3)+(9)+(10)

↓ parent

0 ⇒ 0 ⇒ sum = sum+(-3)+(9)+(10)+0

∴ Ans = 19-3 = 16.

∴ sum upto 7

So.

+3/5 6+5=11 4-4/0

⇒ 16

⇒ so now we also know how get sum works.

(#) Overall code for BITree as :-

```
public class Main {
    final static int MAX = 1000;
    static int[] BITree = new int[MAX];

    static int getSum(int index) {          // first n numbers sum
        int sum = 0;
        index = index + 1;
        while (index > 0) {
            sum += BITree[index];
            index -= index & (-index);      // get parent formula.
        }
        return sum;
    }
```

(GetNext) Function

```
    static void updateBIT(int n, int index, int val) {
        index = index + 1;
        while (index <= n) {
            BITree[index] += val;           // put in that node (add if already present value)
            index += index & (-index);      // To get next node to be updated.
        }
    }
```

(Means (index node) (add p) // BITree[index] = BITree[index] + val)

Find next node value we need to put that arr value.

```
    static void constructBITree(int arr[], int n) {
        // Initialize BITree as 0
        for (int i = 1; i <= n; i++) {
            BITree[i] = 0;
        }
        for (int i = 0; i < n; i++) {
            updateBIT(n, i, arr[i]);
        }
    }
```

(Looks like getNext() 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;
        // build fenwick tree from given array :
        constructBITree(freq, n);
        System.out.println("Sum of elements in arr[0..5] is " + getSum(5));
        // Let use test the update operation
        updateBIT(freq.length, 3, 2);
        System.out.println("Sum of elements in arr[0..5] after update is " + getSum(5));
    }
}
```

I study in depth also from this.

⇒ Output :- 9 → before update query sum upto 4

11 → after update query sum upto 4.

Code for range sum will be as :-

```
static int rangeSum (int l, int r) {
    return (getSum(r) - getSum(l-1));
}
```

e.g.

```
int[] arr = {1,2,3,4,5,6,7,8,9};
(it's always inclusive)
Sysо( getSum(2)) // output = 1+2+3 = 6   (upto index=2)

(for getting index 2 to 5) //3+4+5+6.
int ans = getSum(5) - getSum(2-1);
Sysо(ans) // output = 3+4+5+6 = 18.
```

18th Sep 2022

Ok now understood completely how BIT tree or Fenwick tree is build and how we are updating and how we are getting range sum.