It’s a bit different implimentation of Segment tree. There is two operation-

1. **operation “a” :** Just we have to update the tree at the end of the existing tree, that is we know, at present the existing limit is n, then since the “a” operation add new element at the end, that mean we will update the index “n+1” .
2. **operation “c” :**Here the value at the given index will have to be printed and then deleted. That mean the input array will be compressed. Now, how we can make the tree to be compressed since we are working on tree !!!

The solution is, we will not compress the tree, rather we will make the “vis” value of the tree as “0” . Observe that , the “vis” value here is indicating that the value at that index is either existing or removed. That’s why the **sum**value of **root node** ( tree[1].sum ) of the tree always indicates the total number of elements present in the tree. Using this property now we can easily find the element at given index.

Observe that here the most tricky logic is – changing the value of **index** in the **query**function. Since, when we are entering the query function, we are decrementing the **sum** value of each node in the path of the required index by 1. Since we know, ultimately we have to delete an element at the given index.

So actually the path will experience a decrement by 1, that means the total number of element will be decreases by 1. That’s why in the main function, when we get a “c” operation, at first we are checking that whether the sum value of  root node ( tree[1].sum ) is less than the given index. That is whether I am asked an index which does not exist in my present tree.

If not, then we are sure that the index exists and lets find out the element at that index and let’s expel that element from the tree !!