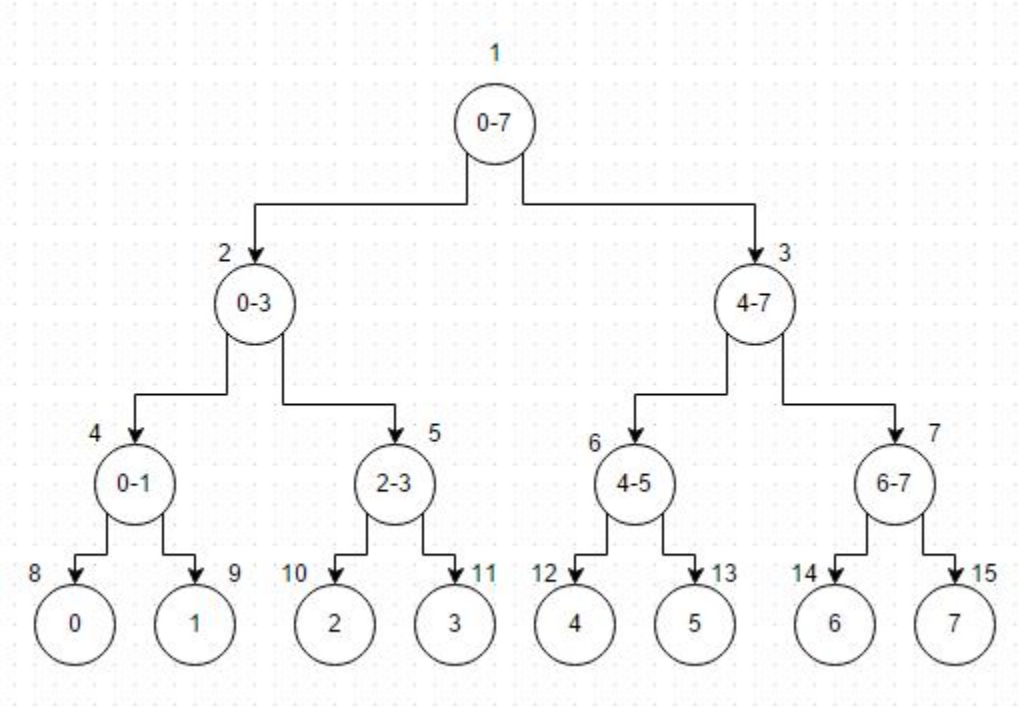
Segment trees as some of you might have heard of, is a cool data structure, primarily used to answer range type queries efficiently. It is a balanced binary tree, nodes of which correspond to various intervals. So, Segment Tree is a Tree data structure for storing intervals, or segments. A segment trees has only three operations: build_tree, update_tree, query_tree.

Building tree: *To build the tree segments or intervals values*
Update tree: *To update value of an interval or segment*
Query tree: *To retrieve the value of an interval or segment*
Due to the recursive nature of operations on the seg-tree, it is incredibly easy to think about and code.

Structure of a segment tree:

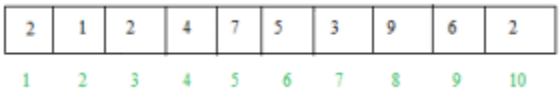
Each node in a segment tree stores statistics for some range/segment of an array. The leaf nodes stores statistics for individual array elements. If the input array had 2^n elements (i.e., the number of elements were a power of 2), then the segment tree over it would look something like this:



Each node here shows the segment of the input array. The number outside a node indicates its index in the segment tree array (1-indexed). Clearly, if the array size N were a power of 2, then the segment tree would have 2*N-1 nodes. Given a node with label *i*, its left and right children are *2i* and *2i+1* respectively. Node 8 would correspond to the interval [0,0] (element at index-0) in the array and node 9 would be [1,1](element at index-1) and so on. As we go up the seg-tree, the interval corresponding to each node in the segment tree is found by merging the intervals of its two children. That way, node 4 will correspond to interval [0,1] and node 3 to interval [4,7] and so on.

Each node, has 0 or two children. Left and right. If a node's interval is [*l*, *r*) and $l + 1 \neq r$, the interval of its children will be [*l*, *mid*) and [*mid*, *r*) in order where mid=(l+r)/2 , so the height of this tree will be $O(log(n))$ (where n is number of elements of array).

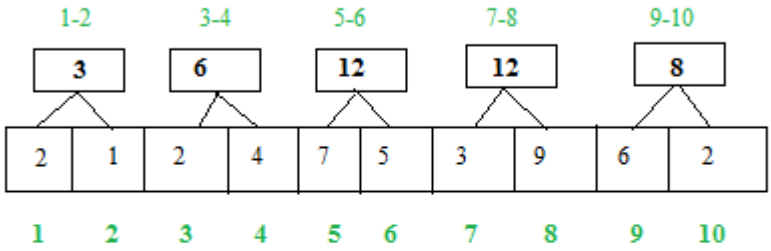
To understand the concept of segment trees let's start with the simple problem.



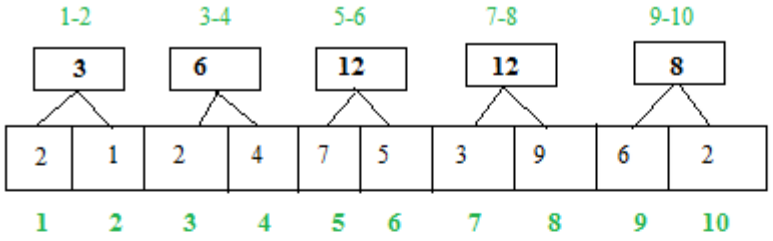
Given above is an array and we can perform two operations on this array:

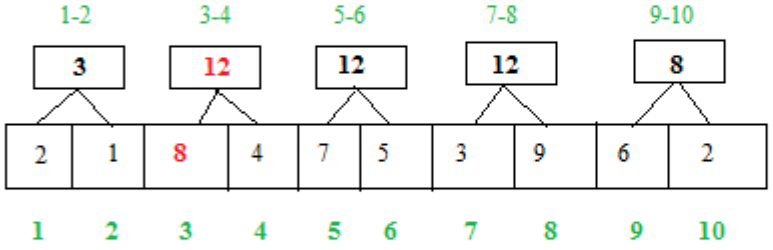
1. **Sum:** *Calculate the sum of elements between two given indexes. eg. Sum of elements from index 3 to index 7 shall be 2 + 4 + 7 + 5 + 3 = 21. In other words this operation can also be expressed as "find sum of elements in a given range".*
2. **Update:** *Perform an update operation on the array by updating the value on a certain index. Eg. A typical update operation shall look like "update element at index 3 to 8". This would change the element at index 3, i.e. 2 to 8.*

We see that Sum operation is an O(n) operation and update operation is a O(1) operation. So let's try to ease it a little by storing the sum of two indexes at a time as shown below.



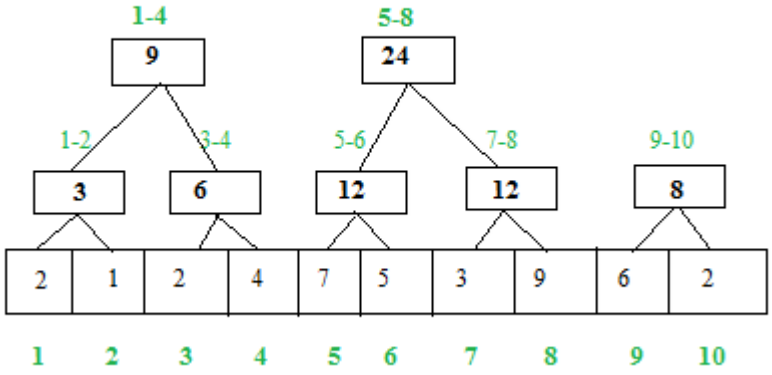
Now let's say we need to calculate the sum of elements from 2 to 9 we need to add sum of 2 + (sum of 3 and 4) + (sum of 5 and 6) + (sum of 7 and 8) + sum of 9. Since the sum of elements 3-4, 5-6 and 7-8 is already pre-calculated our calculation reduces down to 1 + 6 + 12 + 12 + 6 = 37. This reduces the number of operations involved in finding the sum to nearly half. If we look at the update operation, we just need to update two array indexes as shown below:



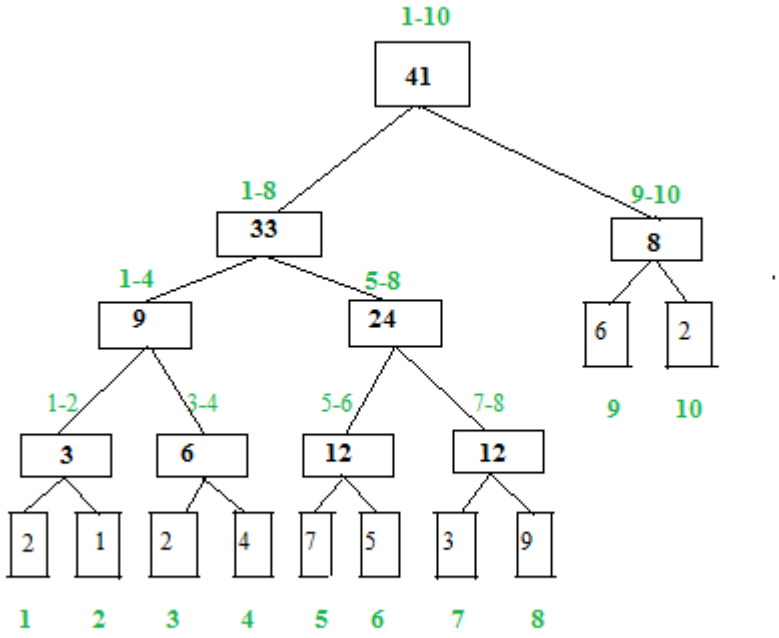


A change in index 3 i.e. updating 2 to 8 results in only two changes.

So we have effectively reduced the time complexity of sum operation. We can further optimize this by going one level up and storing sum of sum of two indexes as illustrated below.

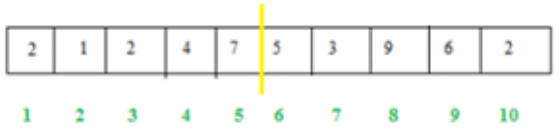


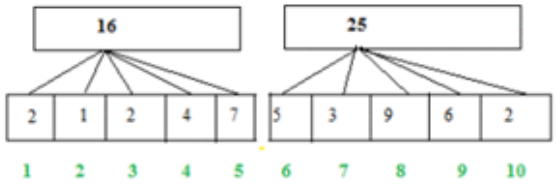
And then progressively keep going level up to arrive at final tree as illustrated in the figures below.



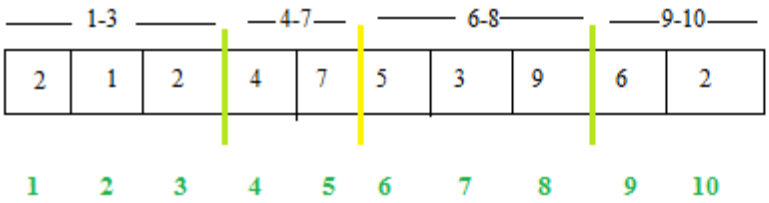
Above tree unbalanced binary tree. To make balanced binary tree (balanced segment tree), the steps are shown below:

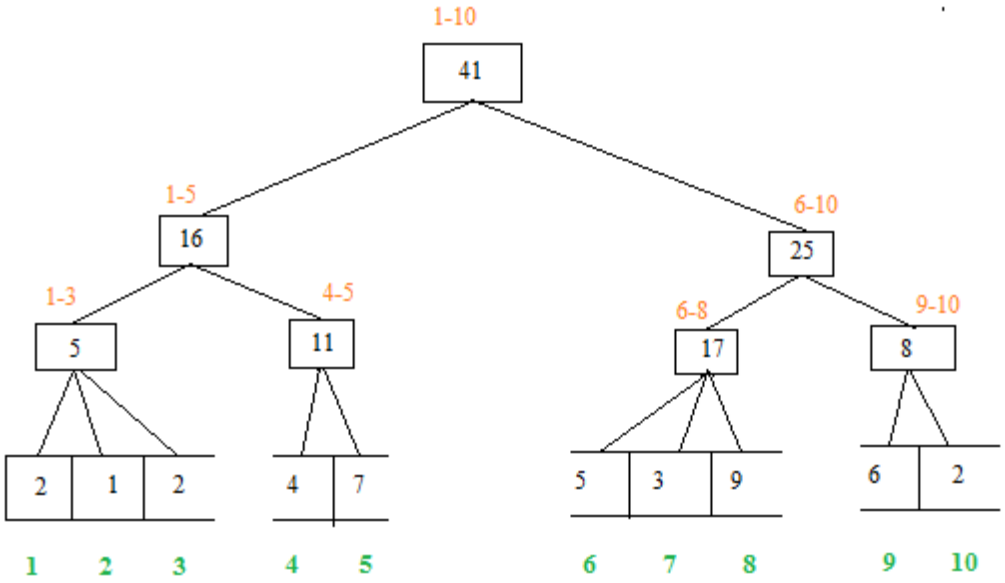
1) *We start by dividing the array from the middle into two ranges 1-5 and 6-10.*



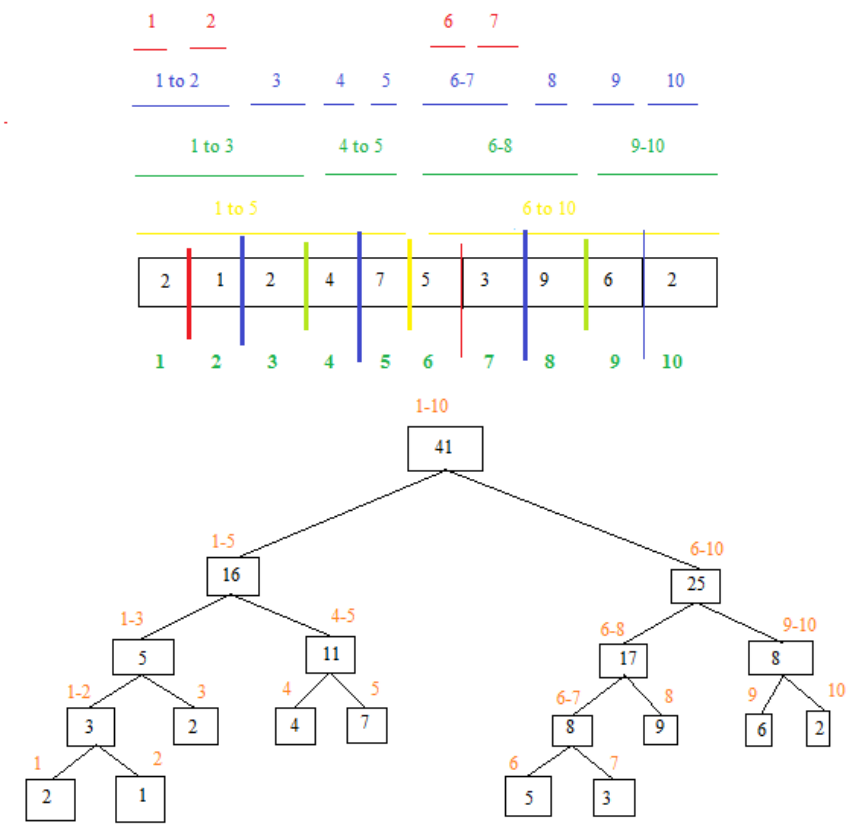


2) *Next we further partition the range 1-5 to 1-3 and 4-5 (remember that since 5 is not perfectly divisible by 2 we round off 5/2 = 2.5 to 3 and use it to partition into two ranges i.e. 1-3 and 4-5) , similarly we partition range 6-10 in the sub ranges 6-8 and 9-10 as shown below.*





Next we again partition the sub-ranges to even smaller sub-ranges till the time we get to the single block as shown in figure below.



Sum: Let's say we have to calculate the sum from index 2 to index 9 above, we shall follow following steps.

1) *Our problem range is 2-9 in addition to that we start with a reference range which will be the full range initially i.e. 1-10 and starting node as root node and starting sum as zero.*

2) *Now we check whether reference range lies inside the problem range i.e whether 1-10 lies inside 2-9, this step is like checking whether the sum of numbers from 2-9 contains the sum of numbers from 1-10. If the answer is no then we partition the reference range in two reference ranges and keep repeating the process until we find the reference range inside the problem range, when that happens we update the sum as sum= sum + (sum of elements in current reference range).*

Update: *The update operation is done by updating leaf node and also updating all the nodes above it in hierarchical order. As can be seen in the figure below where the element at index 2 is 1 and when it's changed to 3 i.e. increased by 2 all the nodes above it in the hierarchy need to be updated.*

The number of levels of binary tree is equal to $\log_2 n$ (number of levels of a given tree can also be said as height of tree). Hence the worst case time complexity of the sum operation shall be $O(\log_2 n)$.

For every update operation we need to traverse the tree from the root to leaf hence the time complexity of the update operation shall be $O(\log_2 n)$ consistently.

Example :

We have an array $[a_1, a_2, ..., a_n]$ and q queries. There are 2 types of queries.

1. ***S l r, Print $a_l, a_{l+1}, ..., a_r$***
2. ***M p x, Modify a_p to x, it means $a_p = x$.***

First of all we need to build the segment tree, for each node we keep the sum of its interval, for node i we call it $s[i]$, so we should **build** the initial segment tree.

```
build(int id ,int left ,int right ){
      if(right - left < 2){
            s[id] = a[left];
            return;
      }
      int mid = (left + right)/2;
      build(id * 2, left, mid);
      build(id * 2 + 1, mid, right);
      s[id] = s[id * 2] + s[id * 2 + 1];
}
```

So, before reading the queries, we should call *build*(1,0,n) .

Modify function :

```
modify(int index_to_modify,int new_val,int id ,int left, int right){
      s[id] += new_val - a[index_to_modify];
      if(r - l < 2){
            a[index] = new_val;
            return;
      }
      int mid = (left + right)/2;
      if(index_to_modify < mid)
            modify(index_to_modify, new_val, id * 2, left, mid);
      else
            modify(index_to_modify, new_val, id * 2 + 1, mid, right);
}
```

(We should call *modify*(*index_to_modify*, *new_val,1,0,n*))

Sum function :

```
sum(int query_left,int query_right,int id,int left,int right){
      if(query_left >= right or left >= query_right)
            return 0;
      if(query_left <= l && r <= query_right)
            return s[id];
      int mid = (l+r)/2;
      return sum(query_left, query_right, id * 2, left, mid) +
             sum(query_left, query_right, id * 2 + 1, mid, right);
}
```

(We should call *sum*(*l*, *r,1,0,n*)).

Usage:

In the range query type problems, in which there are updates also, segment tree is used. Eg: Segment Trees can be used to solve the Range Minimum Query Problem, they can also be used to solve the Lowest Common Ancestor problem.

PROBLEMS:

GSS3 - Can you answer these queries III
QTREE3 - Query on a tree again!
KGSS - Maximum Sum