Segment Trees

17 December 2021 02:21 PM

WHY DO WE NEED SEGMENT TREES?
==============================

[**DISCLAIMER** : I AM USING SUM AS THE PRIMARY OPERATION IN THIS TUTORIAL , WE CAN MAKE THE SEGMENT TREE FOR DIFFERENT RANGE QUERIES AS WE WANT AS PER OUR NEED FOR EX. min , sum , max, etc] .

For ex.
We have an array : [1, 2, 3, 4, 5]
We need to find the sum of all the values between index (2,3) and many more .

BRUTE FORCE / NAIVE METHOD :

We can do this operation in **O(N)** , but when we have Q number of queries i.e. we need to find the sum of different parts of the array Q times , it will take us **O(N * Q)** time .
Even if we make something like a prefix-sum array , if we update the array again and again , we will have to update the prefix sum array again and again and it will still be a **quadratic** solution!

What is the solution to this madness???

BETTER / EFFICIENT METHOD :

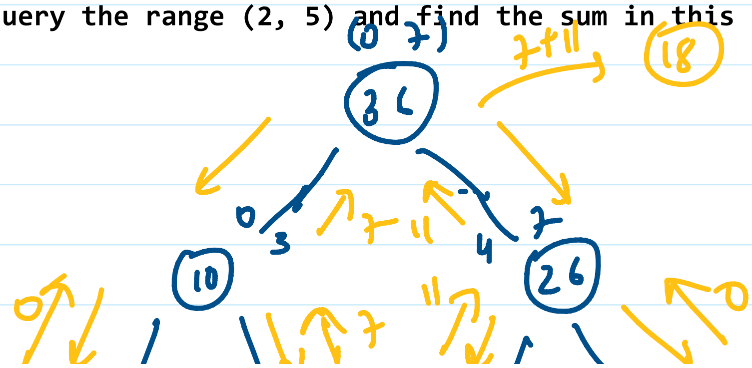
Segment tree is the solution to this . We can make a tree type structure which can perform all these queries in O(log(N)) .

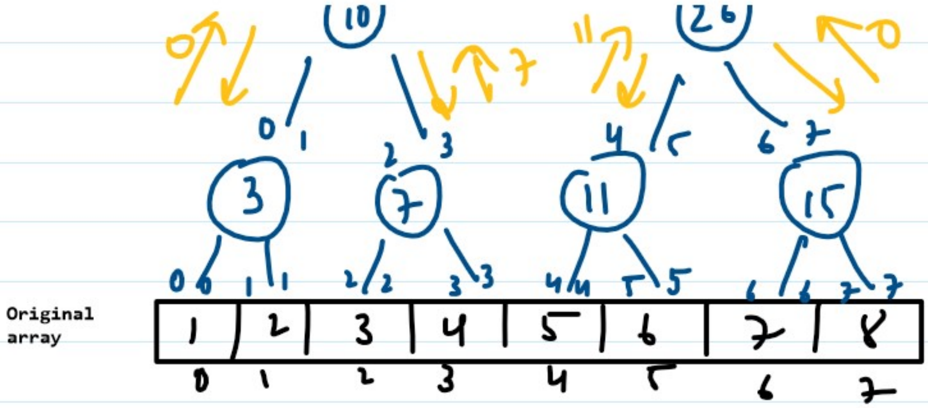
- Segment trees are used for range queries .
- Range queries refer to :
  - Updating a range
  - Querying a range
  - And doing all of that in an efficient manner .

For ex.
If our array is [1, 2, 3, 4, 5, 6, 7, 8]
We will make a segment tree which will look like this .

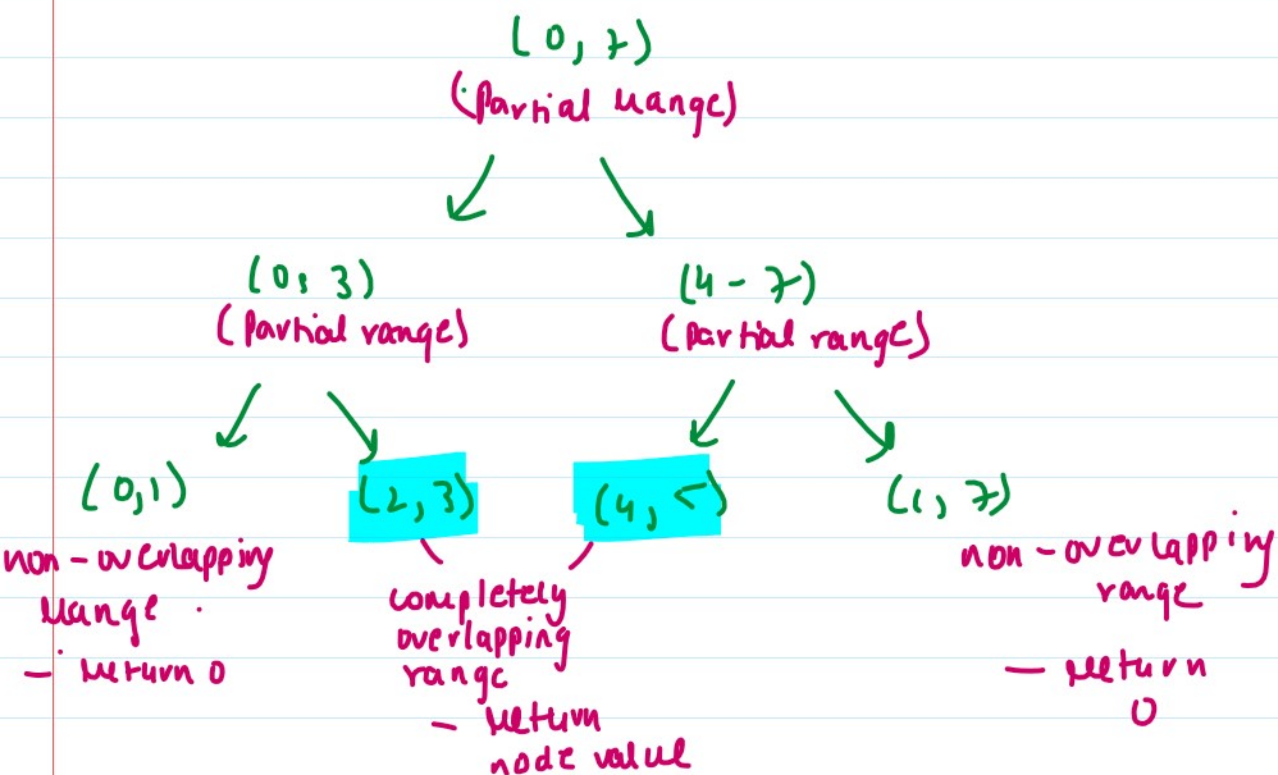
QUERY :

Let's query the range (2, 5) and find the sum in this range :





Here we see that we traverse the range (2, 5).
   The traversal looks as follows :



- Types of ranges :
  - Completely overlapping range - We will add this completely to our answer , no need to explore the children .
  - Partially overlapping range - We will keep this with us as we need a part of it , here we will explore the left and right child .
  - Non overlapping range - we will discard it since we don't need it at all , no need to explore the children .

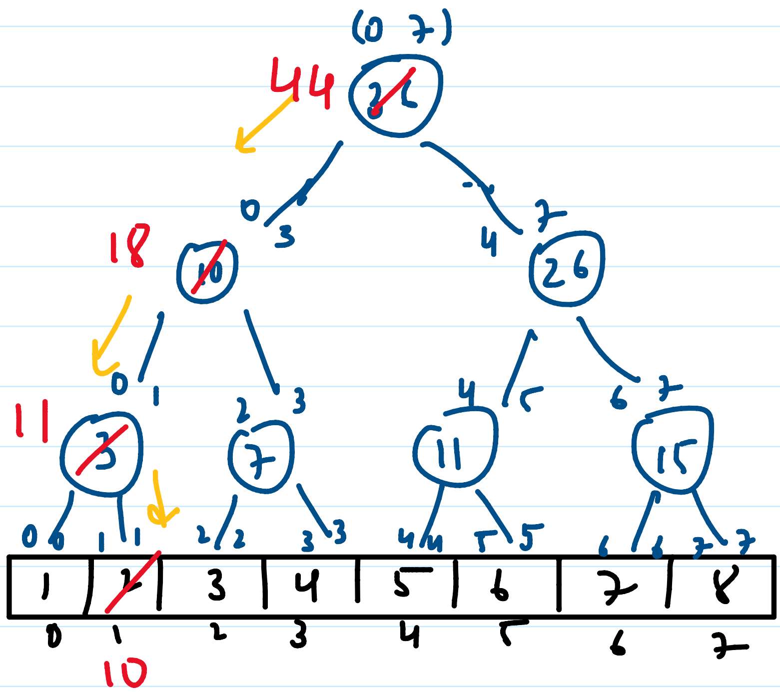
Here we saw that if we want to query a particular range , we can do that very easily in a time complexity of : **O(log(N))** , N - size of array .

UPDATE :

This is the easiest operation among all . Here we just recursively traverse the range in the tree and update the ancestors .

For ex .

Let's update the node at index 1 from 2 -> 10:



Original array

Here we go into the range as :

$(0,7) \rightarrow (0,3) \rightarrow (2,3) \rightarrow (2,2)$

36 10 3 2

Then we come back updating ONLY the ancestors of the node with the sum of the left and right node .

$(0,7) \rightarrow (0,3) \rightarrow (2,3) \rightarrow (2,2)$

$36 \rightarrow 44$ $10 \rightarrow 18$ $3 \rightarrow 11$ $2 \rightarrow 10$

- **So we understand that :**

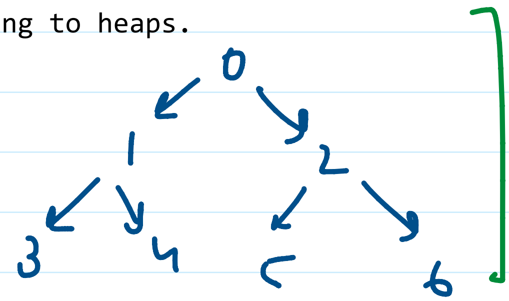
  - Segment tree is used when we need to do multiple range updates on a large range efficiently .

  - The operations in segment tree include :
    - Query
    - Update

  - Both of these operations are done in **log(N)** time complexity .

We understand the structure of segment tree in the form of an array due to its indexing :

0, 1, 2, 3 . . . . . so on .
This is a similar thing to heaps.



These are indexes

Here we can see that the :
If the parent node has index : index
The child nodes have index :
- Left : 2 * index + 1
- Right : 2 * index + 2

LET'S NOW UNDERSTAND THE CODE :

Structure of the segment tree :

```
struct segmenttree {
    int n;
    vector<int> st;

    segmenttree(int n) {
        this->n = n;
        st.resize(4 * n, 0);
    }
};
```

size of the tree

st : segment tree vector .

Build :

```cpp
void build(int start, int end, int node, vector<int> v) {
    if (start == end) {
        st[node] = v[start];
        return;
    }

    int mid = (start + end) / 2;
    build(start, mid, 2 * node + 1, v);
    build(mid + 1, end, 2 * node + 2, v);

    st[node] = st[2 * node + 1] + st[2 * node + 2];
}
```

base case : leaf node .

going to the leaf node .

adding sum of child nodes

Query :

```cpp
int query(int l, int r, int start, int end, int node) {
    // no overlap
    if (start > r or end < l) {
        return 0;
    }

    // complete overlap
    if (start >= l and end <= r) {
        return st[node];
    }

    // partial overlap
    int mid = (start + end) / 2;
    int q1 = query(l, r, start, mid, 2 * node + 1);
    int q2 = query(l, r, mid + 1, end, 2 * node + 2);

    return (q1 + q2);
}
```

No overlap

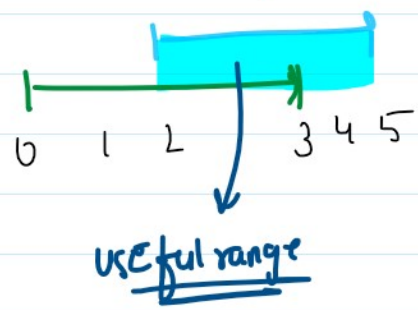
complete overlap

Partial overlap

for ex.

our range : [2,5]

total range : [0,3]



Update :

```cpp
void update(int start, int end, int index, int value, int node) {
    if (start == end) {
        st[node] = value;
        return;
    }

    int mid = (start + end) / 2;
    if (index <= mid) {
        update(start, mid, index, value, 2 * node + 1);
    } else {
        update(mid + 1, end, index, value, 2 * node + 2);
    }
    st[node] = st[2 * node + 1] + st[2 * node + 2];
}
```

updating ancestors