Segment Trees

Created	@April 24, 2022 3:59 PM
Class	Segment trees
Type	
Materials	
Reviewed	☐

Motivation Problem:

Given an array of length N, containing all integers. You'll get Q queries where in each query you'll get L, R two numbers denoting two valid indexes of the array. For each query calculate the sum of elements from index L to R in the array. N <= 10^5 and Q <= 10^5

Brute Force

We could have processed all the queries from start to end. During the processing of each query, we can start from index L to R and sum up the elements. O(QN) -> TLE

```
// for any current query
for(int i = L; i <= R; i++) { // O(n)
  sum += arr[i];
}
```

Can we do some pre-processing ? ? ?

Hints:

- Can we pre process the data and do some manipulation so that we are able to directly derive the sum ?

- Let’s try to solve a simpler problem. Assume for every query we have L = 0. So if L is always equals to 0 the problem reduces to finding the sum of some prefix of the given array. And prefixes can be precomputed.

Logic for prefix sum: Assume we have a function f(i) which calculates sum of elements from index 0 till the index i, i.e. gives the prefix sum till i, then `f(i) = f(i-1) + arr[i]`

```
std::vector<int> prefix(arr.length, 0);
prefix[0] = arr[0];
for(int i = 1; i < arr.length; i++) {
  prefix[i] = prefix[i-1] + arr[i]; // follows the above relation
}
```

Now if we have these prefixes pre computed then, we can use another property of prefix sums i.e.

> sum(L, R) = f(R) - F(L-1)

> sum(L, R) = F(R) - F(L) + arr[L]

So for every query we can use the above formulae and this will help us to answer the queries in O(1) time. So the time complexity comes out as O(Q + N) , O(Q) for processing each query and O(N) for pre computing prefix sum, that’s how we get O(Q+N)

Twist:

Let’s say now we will get one more type of query, apart from the range sum query. So in the new query we will get two number X and Y and we are expected to update the Xth index of the original array with the value Y (Point Update Query)

Now how can we add this query to our solution ?

Simple Solution: Let’s handle the range sum query with prefix sum approach only, which is going to answer every query in O(1) time, and for the point update query we have to update the original given array. If the original given array will change then we have to

change our prefix sum array also. Because the prefix sum array was created by the old elements of the original array, so now for every point update we have to re-calculate the prefix sum array which will take O(N) time.

So the total complexity becomes O(Q(1 + N)) ⇒ O(QN) —→ TLE

Alternate solution: We can avoid taking any prefix sum array and make the point update directly in the original array. Now this point update query becomes O(1) but the problem that arises now is that the range sum query will be now O(N) as we have to iterate on the original array only in order to fetch the sum. O(QN) —→ TLE

Can we do something better ?

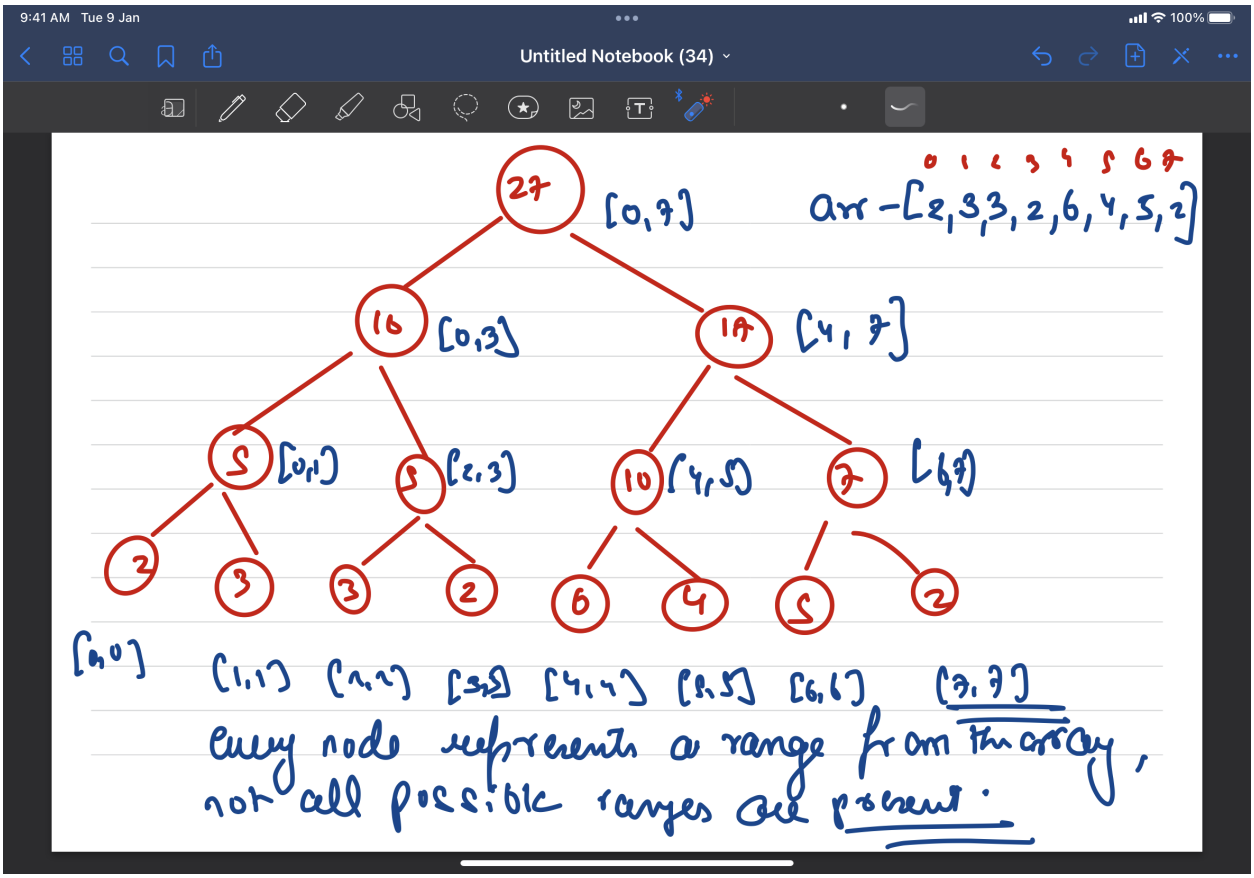
Introduction to Segment trees:

Till now you might have encountered various data structures, some of them might be linear arrays, linked lists, strings, stacks, queues etc. And some of them might be hierarchical in nature like Binary trees, Binary search trees, tries etc. But `Segment Trees` are one of those special data structures which is basically portrayed as a hierarchical data structure but represented in linear fashion, like heaps.

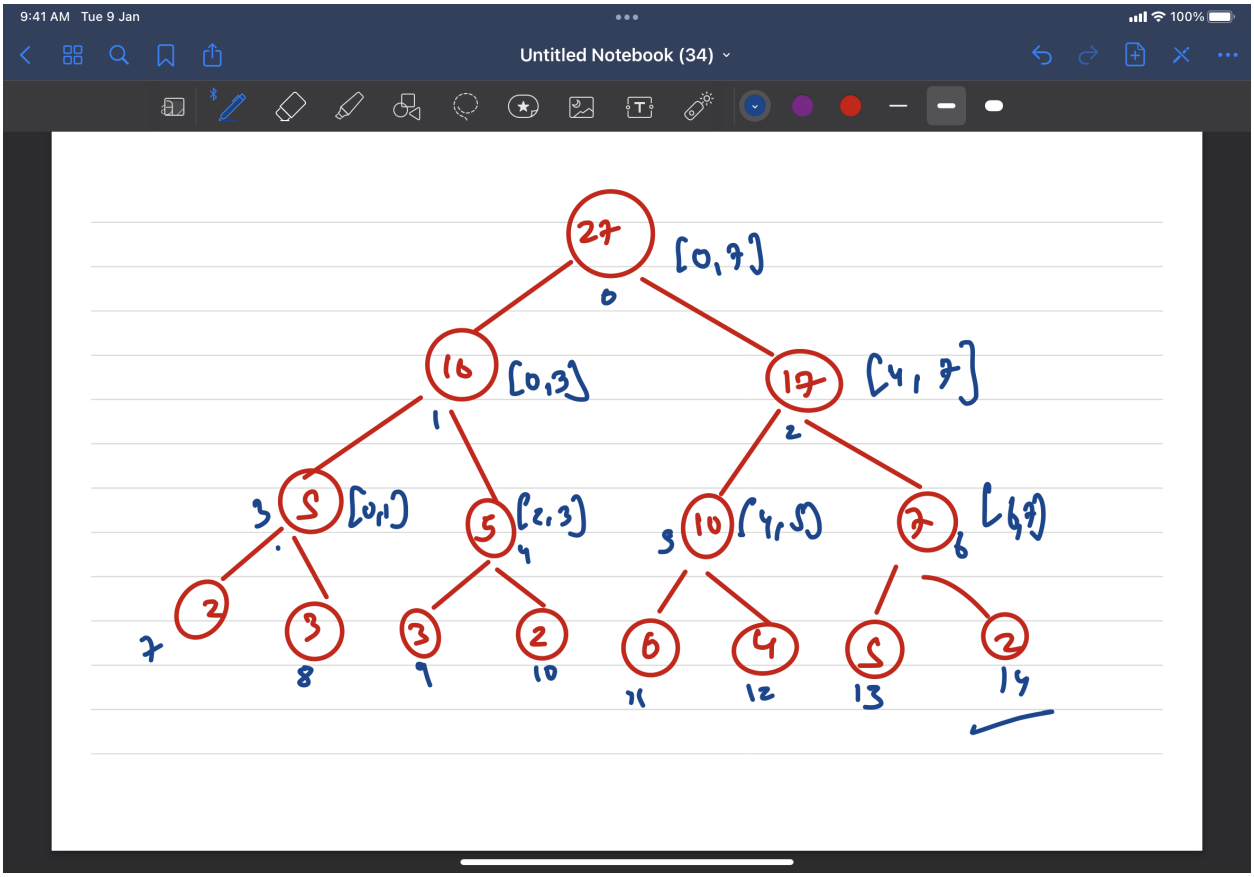
Properties of Segment trees:

- It is a binary tree.
- It is a full binary tree (every node either has zero or two children).
- It is a balanced binary tree (The absolute difference between the height of the left and right subtree is less than or equal to 1).

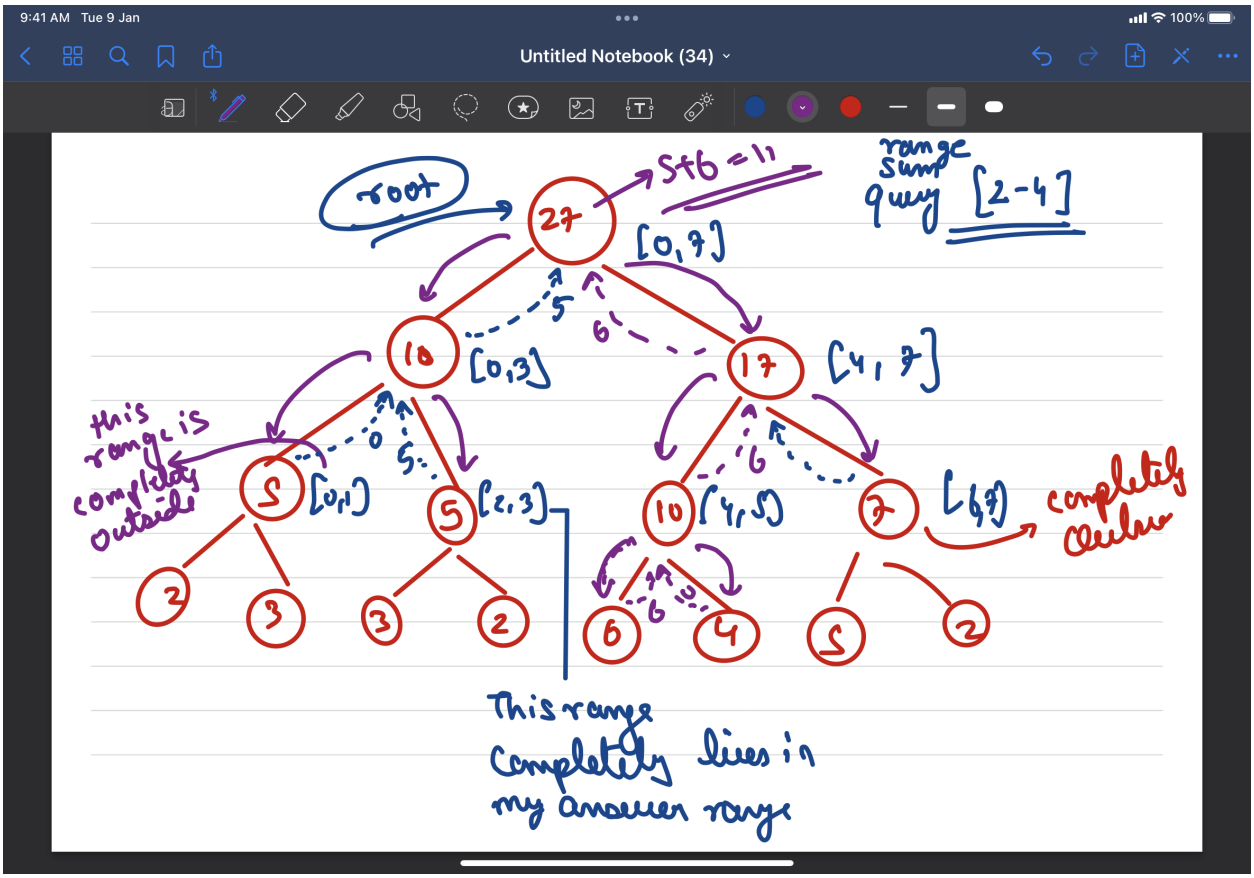
A segment tree is a binary tree built on top of an array. This structure allows aggregation of queries and updates over array intervals in logarithmic time.



The above image shows the default structure of a segment tree. In the segment tree every node represents answer for a particular range, but not all ranges are directly fitted into the nodes. For ex: the range 0, 3 is available but 2-5 is not.



Now, we can visualise the hierarchical structure of segment tree very easily inside an array. Because of the structural symmetry and being a full balanced binary tree, if we have a node at index `x` then the left child will be at index `2*x+1` and the right child will be at index `2*x+2`. (this notations is very similar to that of a heap). Also if we have a node `x` then we can fetch the parent by `floor(x/2)`.



Let’s say we got a range sum query for 2-4, so there can 3 cases:

Case 1: The range represented in the node is completely outside the range of query.

Case 2: The range represented in the node is completely inside the range of query.

Case 3: The range of the node has a partial overlap to the range of query.

So to solve the range sum query problem, we can devise a recursive algorithm. We can check the the root node is following which case, if it is case 1, the return 0 no action is required. If it is case 2, then return the value present at the node as this value represents the sum of the range.

Otherwise if it is case 3, then there is a partial overlap. In this case we need to explore both our subtrees so that can get the answer of precise range.

Let's say F(root, L, R, NL, NR) be a function that returns the range sum from L to R where NL is the start of the range of node and NR is the end of the range of node.

```
int query(std::vector<int> &tree, int NL, int NR, int L, int R, int treeNode) {
  if(NL > R or NR < L) return 0; // complete outside
  if(NL >= L and NR <= R) return tree[treeNode]; // complete overlap
  // if the other cases don't hit then it's a partial overlap
  int mid = (NL + NR) / 2;
  int a1 = query(tree, NL, mid, L, R, 2*treeNode + 1); // fetch the sum from LST
  int a2 = query(tree, mid+1, NR, L, R, 2*treeNode + 2);// fetch the sum from RST
  return a1 + a2;
}
```