PERSISTENT SEGMENT TREE

November 5, 2023

Adari Aravind (2019MCB1211) ,
Abhishek Shah (2020MCB1225) ,
Mohd Junaid (2019MCB1226)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Sravanthi Chede

Summary: A persistent data structure maintains a record of its previous states for each modification, enabling access to any version of the structure and the execution of queries on it. We have implemented an algorithm to address a real-life problem using a persistent segment tree. The Segment Tree is particularly useful when dealing with multiple range queries on an array and modifications to its elements. For instance, scenarios involving finding the sum or minimum (commonly known as the Range Minimum Query problem) of elements within a specified range in an array can be efficiently tackled using the versatile Segment Tree data structure. The persistent segment tree extends this capability, allowing for the execution of multiple range queries on various versions of the array while retaining previous data.

A practical application of persistent data structures is in version control systems, where branching from the history enables updates to be made and subsequently merged back into the original master root. The immutability aspect of persistent structures contributes to partial or full persistency. The time complexity is directly linked to the amount of extra memory utilized during updates, avoiding the need for unnecessary large copies of previous updates. Updates typically operate in logarithmic time and memory for a single update.

Persistent segment trees find application in various data structures such as Red-black trees, treaps, and random access deques. Our objective is to implement persistency in the segment tree while ensuring that each change does not exceed $O(\log n)$ time and space.[5].

---

1. Introduction

Segment Tree is itself a great data structure that comes into play in many cases. Here we will introduce the concept of Persistency in this data structure. Persistency, simply means to retain the changes. But obviously, retaining the changes cause extra memory consumption and hence affect the Time Complexity. Our aim is to apply persistency in segment tree and also to ensure that it does not take more than $O(logn)$ time and space for each change.

Basic idea of Persistent segment Tree Algorithm

Since we want to preserve the previous state after each update operation we can build a new version of the segment tree each time an update operation is performed. We can store all the previous versions of the Segment Tree in a two-dimensional array or in a vector of vectors depending on whether we have used an array or a vector to build the Segment Tree. However, as the build query takes $O(N)$ time and space complexity. Thus, if there are Q update queries it will take $O(Q * N)$ time and space complexity to get performed. We can use a persistent segment to solve the problem much efficiently where each update operation can be done in $O(logN)$

time and space complexity. Let us see how we can implement the same.

What is Segment Tree?

A Segment Tree is a data structure that stores information about array intervals as a tree. This allows answering range queries over an array efficiently, while still being flexible enough to allow quick modification of the array. This includes finding the sum of consecutive array elements $a[l....r]$, or finding the minimum element in a such a range in $O(log(N))$ time. Between answering such queries, the Segment Tree allows modifying the array by replacing one element, or even changing the elements of a whole subsegment (e.g. assigning all elements $a[l....r]$ to any value, or adding a value to all element in the subsegment).[1].

In general, a Segment Tree is a very flexible data structure, and a huge number of problems can be solved with it. Additionally, it is also possible to apply more complex operations and answer more complex queries (see Advanced versions of Segment Trees). In particular the Segment Tree can be easily generalized to larger dimensions. For instance, with a two-dimensional Segment Tree you can answer sum or minimum queries over some subrectangle of a given matrix in only $O(log^2(N))$ time.

One important property of Segment Trees is that they require only a linear amount of memory. The standard Segment Tree requires $4n$ vertices for working on an array of size n.[4]. A new section is created by the command

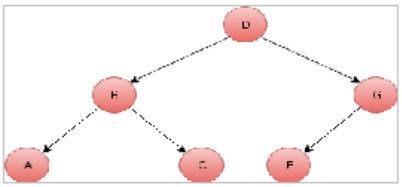
What is Persistent data structures?

A persistent data structure is a data structure that always preserves the previous version of itself when it is modified. They can be considered as 'immutable' as updates are not in-place.

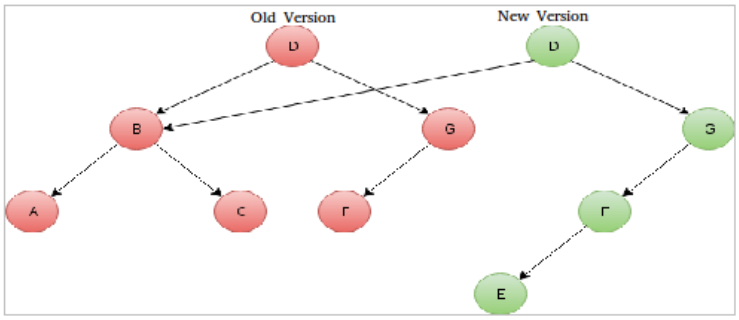
A data structure is partially persistent if all versions can be accessed but only the newest version can be modified. Fully persistent if every version can be both accessed and modified. Confluently persistent is when we merge two or more versions to get a new version. This induces a DAG on the version graph.

Persistence can be achieved by simply copying, but this is inefficient in CPU and RAM usage as most operations will make only a small change in the DS. Therefore, a better method is to exploit the similarity between the new and old versions to share structure between them. We have implemented PST using binary search tree insertion.[2].

Binary Search Tree Insertion: Consider the problem of insertion of a new node in a binary search tree. Being a binary search tree, there is a specific location where the new node will be placed. All the nodes in the path from the new node to the root of the BST will observe a change in structure (cascading). For example, the node for which the new node is the child will now have a new pointer. This change in structure induces change in the complete path up to the root. Consider tree below with value for node listed inside each one of them[3].



Let's add node with value 'E' to this tree.



2. Equations

As an input we receive two integers l and r and we have to compute the sum of the segment $a[l..r]$ in $O(log(n))$.

To do this, we will traverse the Segment Tree and use the precomputed sums of the segments. Let's assume that we are currently at the vertex that covers the segment $a[tl...tr]$. There are three possible cases.

The easiest case is when the segment $a[l...r]$ is equal to the corresponding segment of the current vertex (i.e. $a[l...r] = a[tl...tr]$), then we are finished and can return the precomputed sum that is stored in the vertex.

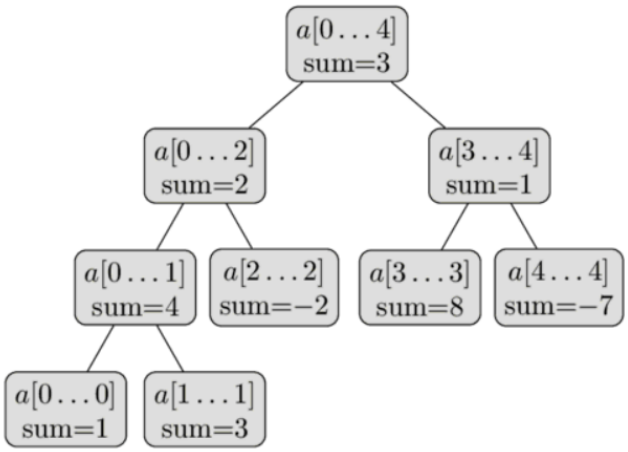
Alternatively the segment of the query can fall completely into the domain of either the left or the right child. Recall that the left child covers the segment $a[tl...tm]$ and the right vertex covers the segment$a[tm+1...tr]$ with $tm = (tl + tr)/2$. In this case we can simply go to the child vertex, which corresponding segment covers the query segment, and execute the algorithm described here with that vertex.

And then there is the last case, the query segment intersects with both children. In this case we have no other option as to make two recursive calls, one for each child. First we go to the left child, compute a partial answer for this vertex (i.e. the sum of values of the intersection between the segment of the query and the segment of the left child), then go to the right child, compute the partial answer using that vertex, and then combine the answers by adding them. In other words, since the left child represents the segment $a[tl...tm]$ and the right child the segment $a[tm+1...tr]$, we compute the sum query $a[l...tm]$ using the left child, and the sum query $a[tm+1...r]$ using the right child.

So processing a sum query is a function that recursively calls itself once with either the left or the right child (without changing the query boundaries), or twice, once for the left and once for the right child (by splitting the query into two subqueries). And the recursion ends, whenever the boundaries of the current query segment coincides with the boundaries of the segment of the current vertex. In that case the answer will be the precomputed value of the sum of this segment, which is stored in the tree.

In other words, the calculation of the query is a traversal of the tree, which spreads through all necessary branches of the tree, and uses the precomputed sum values of the segments in the tree.

Obviously we will start the traversal from the root vertex of the Segment Tree. For example let us assume we have an $array = [1, 3, -2, 8, -7]$ with the visual representation.



we can already conclude that a Segment Tree only requires a linear number of vertices. The first level of the tree contains a single node (the root), the second level will contain two vertices, in the third it will contain four vertices, until the number of vertices reaches n. Thus the number of vertices in the worst case can be estimated by the sum $1 + 2 + 4 + ... + 2^{log_2(n)} < 2^{log_2(n)+1} < 4n$.

(n) It is worth noting that whenever n is not a power of two, not all levels of the Segment Tree will be completely filled. We can see that behavior in the image. For now we can forget about this fact, but it will become important later during the implementation.

The height of the Segment Tree is $O(logn)$, because when going down from the root to the leaves the size of the segments decreases approximately by half.

3. Algorithms

Problem statement : Given an array A[] and different point update operations.Considering each point operation to create a new version of the array. We need to answer the queries of type
Q v l r : output the sum of elements in range l to r just after the v-th update.

EXPECTED OUTPUT:

```
Enter the size of the array: 5
Enter array elements separated by spaces:
1 2 3 4 5
Enter number of versions to be constructed for the segement tree: 3
Enter Index to be updated and new value separated with spaces: 0 10
Enter Index to be updated and new value separated with spaces: 4 50
Press q to quit or y to continue: y
Enter the value of l in arr[l,r]: 0
Enter the value of r in arr[l,r]: 4
Enter desired version: 0

The sum of elements in the segment [0,4] in version [0] = 15
Press q to quit or y to continue: y
Enter the value of l in arr[l,r]: 0
Enter the value of r in arr[l,r]: 4
Enter desired version: 1

The sum of elements in the segment [0,4] in version [1] = 24
Press q to quit or y to continue: y
Enter the value of l in arr[l,r]: 0
Enter the value of r in arr[l,r]: 4
Enter desired version: 2

The sum of elements in the segment [0,4] in version [2] = 69
Press q to quit or y to continue: q

Exiting Program.............


...Program finished with exit code 0
Press ENTER to exit console.
```

Pseudo-algorithms for above persistent segment tree is as follows.

Algorithm 1 Function makeTree (root, start, finish)

```
/*Function for the version-0 tree construction*/
int centre = ( start + finish ) / 2
if start = finish then
    root -> data = array[start]
    return
end if
newNode -> left = createNode (NULL, NULL, 0)
newNode -> right = createNode (NULL, NULL, 0)
makeTree ( newNode -> left, start, centre)
makeTree ( newNode -> right, start, centre)
newNode -> data = newNode -> left -> data + newNode -> right -> data
```

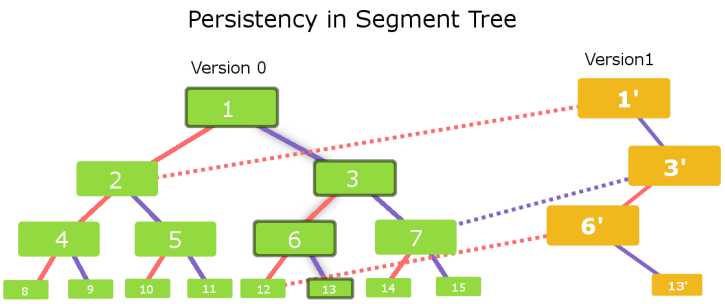
Algorithm 2 Function updateTree (past node, present node, start, finish, index, key)

```
/*Function to create a new version of the tree whenever an update is made at any index of the input array*/
if index > finish or index < start or start > finish then
    return
end if
if start = finish then
    present -> data = key
    return
end if
if index <= centre then
    present -> right = past -> right
    present -> left = createNode( NULL, NULL, 0)
    updateTree ( past -> left, present -> left, start, centre, index, key)
end if
if index > centre then
    present -> left = past -> left
    present -> right = createNode ( NULL, NULL, 0)
    updateTree ( past -> right, present -> right, centre + 1, finish, index, key)
end if
present -> data = present -> left -> data + present -> right -> data
```

Algorithm 3 Function Request (myNode, start, finish, begin, end)

```
/*Function to perform a query on any version of the tree*/
centre = ( start + finish ) / 2;
if begin > finish or end < start or start > finish then
    return 0
end if
if begin <= start and finish <= end then
    return myNode -> data
end if
temp1 = request(myNode− > left, start, centre, begin, end)
temp2 = request(myNode− > right, centre + 1, finish, begin, end)
answer = temp1 + temp2
return answer
```

The pseudo-algorithm written above will be easy to understand if you follow below diagram.



Time Complexity:

Build Query
A Segment tree can contain a maximum of $4n + 1$ odes (1 based indexing). As we visit every node once in the traversal while building the Segment Tree. Hence the time complexity of the build function is $O(n)$. Note that here we consider that the making of new nodes is done in $O(n)$ time complexity.

Update Query
In an upgrade query, we move to the leaf node that needs to be updated. While backtracking we update the value in the non-leaf nodes of the newer version of the segment tree with the sum of the value present in its left and right child. As the height of a Segment Tree can't be more than $(logn)$ to reach the leaf node in our traversal. Hence the time complexity for the upgrade query is $O(logn)$. Here also we consider that the making of new nodes is done in $O(1)$ time complexity.

Range Sum Query The time complexity for a range sum query in a Persistent Segment Tree is the same as that in a Simple Segment Tree. As the addition of two numbers is done in $O(1)$ time complexity thus a Range Sum Query in a Persistent Segment Tree takes a time complexity of $O(logn)$.

Example of Persistent Segment Tree

Problem Statement:

Avnish lives with his girlfriend, Deepika. He plans to spend today's evening with her,driving his new car.The cities in his state are connected to each other via bi-directional roads in such a way that they form a tree.
As he has many enemies in the city, he needs to be careful in choosing the path for the road trip. In each city, resides a member of a certain gang, which wants to kill Avnish. Whenever Avnish passes through that city, the members of that gang become alert. Now, the good thing about these gangs for Avnish is that these gangs hate each other and don't go well with each other. So, a path is dangerous for Avnish if and only if a gang exists on that path in majority. Otherwise the gang members end up fighting each other and the path becomes clear for Avnish.

Definition of majority on a path : If the number of cities occupied by gang members of gang G on a path are x and the number of cities on that path are n, then gang G is in majority on that path if and only if $x > n/2$.
There are N cities in the state and there are N-1 bi-directional roads. Deepika comes up with M plans for the road trip. Each plan is of the form u,v, where road trip will take place from the city u to the city v along the unique path between them. For each plan, you need to tell whether the path in that plan is dangerous or not, and if it is dangerous, you need to output the number of the gang which is in majority on that path.

Input:
First line of input contains two integers N and M, where N is the number of cities in the state and M is the number of plans suggested by Deepika. Next line contains N space separated integers Gi, where Gi is the number of the gang of the gang member in city i. Each of the next N1 lines contain two integers u and v, where u and v have a bi-directional road between them. Next M lines describe the query, i.e., the plan suggested by Deepika. Each of these lines contain two integers p and q, which represents the path between p and q.

Output:
You need to output M lines. ith query is answered by ith line. If the path in the ith query is safe for Dexter, output "S" (withour quotes). If that path is dangerous and gang numbered x is in majority on that path, output "D x" (without quotes).

Pseudo-algorithm code of solution:

Algorithm 4 Function JoinTree (root, N)

```
/*Function to make tree*/
while i != N do
    if 2 * i > N then
        break
        i++
    end if
    p[i]->left = p[2*i+1]
    if 2 * i > N then
        break
        i++
    end if
    p[i]->left = p[2*i+1]
    i++
end while
```

Algorithm 5 Function gangOnPath (root, gang[ ], L, R)

```
/*Function To fill gang[] with gang members in the cities*/
n = 0 // initiated
while l != r do
    if l! = 0 and l < r then
        gang[(p[l]- > gang)]+ = 1
        l=l/2
        n++
    end if
    if r! = 0 and r > l then
        gang[(p[r]- > gang)]+ = 1
        r=r/2
        n++
    end if
end while
gang[(p[l]- > gang)]+ = 1//l = r
n++
return n
```

Algorithm 6 Function SafeOrDangerous (gang[ ], N, n)

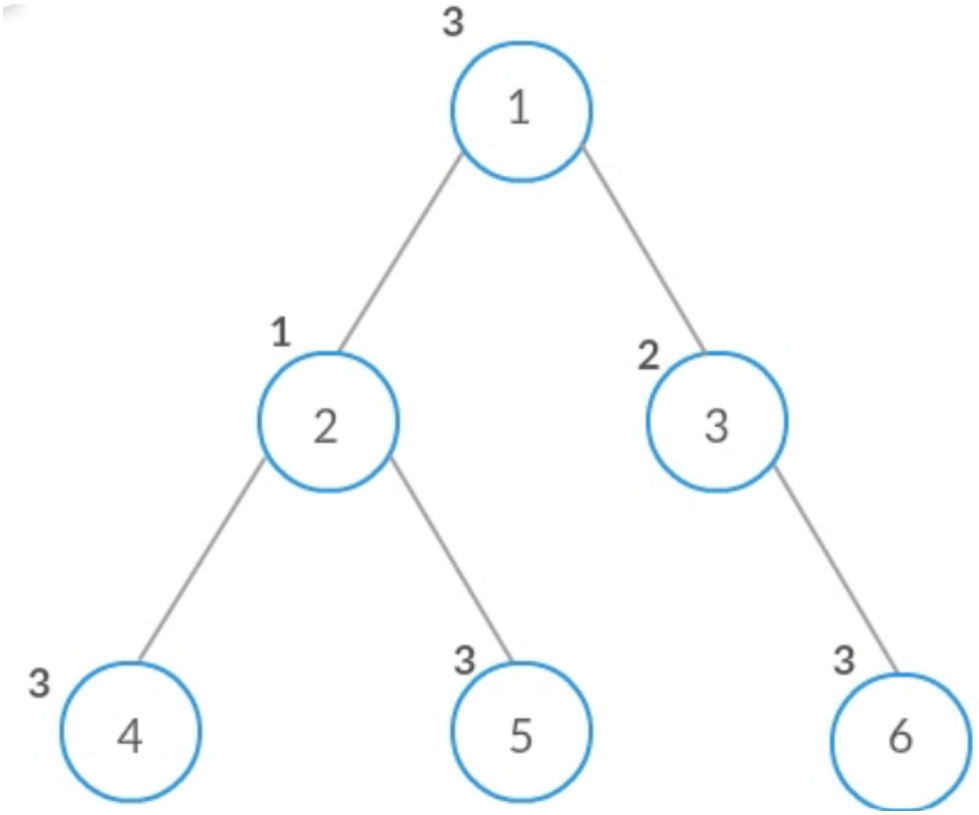
```
/*Function to determine if paths in the plan made by deepika are safe or not*/
i = 0 and k = 0// initiated
while i != N do
    if gang[i] > n/2 then
        print "D x" //where x is i
        k++
    end if
    i++
end while
if k = 0 then
    print "S"
end if
```

Expected Outcome:

```
Enter no. of cities and Plans : 6 8
Gang number seperated by space: 3 1 2 3 3 3
Enter l(starting city) and r(End city):
1 2
1 3
2 4
2 5
3 6
4 6
2 3
4 5
SAFE
SAFE
SAFE
SAFE
SAFE
DANGEROUS AS GANG NO. 3 IS IN MAJORITY!!
SAFE
DANGEROUS AS GANG NO. 3 IS IN MAJORITY!!


...Program finished with exit code 0
Press ENTER to exit console.
```

The pseudo-algorithm written above will be easy to understand if you follow below diagram.



4. Conclusions

A Persistent Segment tree is nothing but a data structure that allows us to answer range queries effectively and at the same time also provides flexibility to update the array. It's time complexity is $O(n)$ for creating segment tree, $O(logn)$ for creating every new version and $O(logn)$ for the range query. And its implementation is as easy as given below.

1. At first, we build Version-0 of our persistent segment tree using the makeTree function with the data present in the array.

2. Then we build Version-1 ,Version-2,..,Version-n according to the user requirement of the segment tree using the updateTree function with the help of its previous versions.

3. We will use the version array to store the root node of all the versions of the segment tree.

4. Finally, we solve range sum queries in all three versions using the Request function.

References

[1] Siddharth Agarwal. Persistent segment tree.

[2] Benjamin Qi Andi Qu. Persistent data structures.

[3] GFG. Persistent data structures.

[4] GFG. Segment tree | sum of given range.

[5] Nitish Kumar. Persistent segment tree.