Segment Trees Data Structure

November 3, 2023

Akshat Nagpal (2022MCB1255) ,
Rhythm Sachdeva (2022MCB1337)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Manpreet Kaur

Summary: The primary objective of this project is to study Segment Trees and implement them in C. Segment Tree is a very powerful and versatile data structure used in computer science and mathematics for efficiently solving a wide range of problems related to range-based queries and updates. It is designed to address scenarios where one needs to perform operations on specific segments or intervals of an array or data set. Segment trees are known for their ability to handle various range-based queries, such as finding the sum, minimum, maximum, or applying custom functions to elements within a specified range.

---

1. Introduction

This project is divided into three parts:

1. Implementation of Segment Trees data structure:

Segment tree is an effective and widely used data structure that supports a variety of range-query problems. These range-based queries basically include working on an array and finding the sum, minimum, maximum, or other functions over a specific range of elements. Segment trees are quite popular due to the better time complexity they provide. As we will study later Segment Trees can perform the operation of answering range query and making point updates in $O(\log N)$ each.

2. Implementation of Lazy Variant of Segment Trees data structure:

Lazy Segment Tree is a variant which handles the problem of range updates. Using the traditional segment tree to update a range can take upto $O(N \log N)$ time. Using lazy variant range updates can be handled in $O(\log N)$ while still maintaining logarithmic time for answering range queries making it a very strong tool.

3. Application of Segment Trees in geometry:

One of the first applications of segment trees was in geometry. Although not the original question, in this section we will deal with a geometrical application of segment trees. Given n line segments and q query line segment on the OX axis, efficiently return the number of line segments intersecting the each query line segment.

2. Analysis

Time Complexity:

In order to find the sum of all elements of an array, we can run a loop over the array and compute the sum, which takes $O(N)$ time. Another query is updating the element at i^{th} index, which requires only changing the value at that index, that takes $O(1)$ time.

Another approach for the same could be to use prefix sum data structure which stores the sum of elements in the range $[0, i]$ of an array at i^{th} index of a newly created array. This way, the summing operation takes $O(1)$ time. However, the update operation still requires $O(N)$ time.

Segment trees, on the other hand, take $O(\log N)$ time to perform both sum and update operations and thus, provide better efficiency in results. To build a segment tree, the input array is recursively divided into smaller segments until each segment contains only one element. The values in the non-leaf nodes are computed based on the values in their child nodes.

They are thus used to perform range-based queries, which include performing operations over a specified range of elements, some of which are listed below-

Range Sum Query: Find the sum of elements in a given range.
Range Minimum Query (RMQ): Find the minimum element in a given range.
Range Maximum Query: Find the maximum element in a given range.

Structure: A segment tree is a binary tree where each leaf node represents an element of the input array, and each non-leaf node represents a segment or range of elements.

It makes use of the divide-and-conquer approach where we compute and store the sum of the elements of the whole array, i.e. the sum of the segment $a[0 \dots n-1]$. We then split the array into two halves $a[0 \dots n/2-1]$ and $a[n/2 \dots n-1]$ and compute the sum of each halve and store them. Each of these two halves in turn are split in half, and so on until all segments reach size 1.

Space Complexity:
The segment tree requires $\Theta(2^{\lceil \log_2 n \rceil}) \approx \Theta(n)$space. It is so because as we move upwards in the tree, the number of nodes become half at each level.

For instance, an input array of size N contains N nodes at the bottom of the segment tree, $\frac{N}{2}$ nodes at the preceding level, $\frac{N}{4}$ nodes at the level preceding this level and so on. So, overall it takes $N + \frac{N}{2} + \frac{N}{4} + ... + 1 = 2N$ space. Since we are taking input in the form of an exact power of 2, thus, the exact space complexity is $\Theta(2^{\lceil \log_2 n \rceil}) \approx \Theta(n)$

Applications: Segment trees have a diverse range of applications in fields including competitive programming, database systems, and geographic information systems, dynamic programming, persistent data structures, parallel and distributed computing. They are essential for solving problems that involve range-based queries and updates.

3. Figures

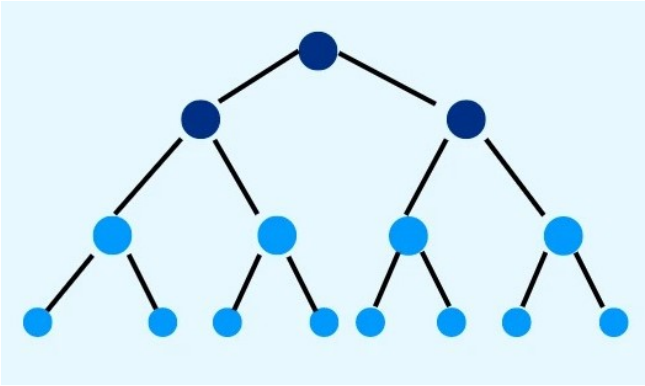

Figure 1: Representation of a Segment Tree

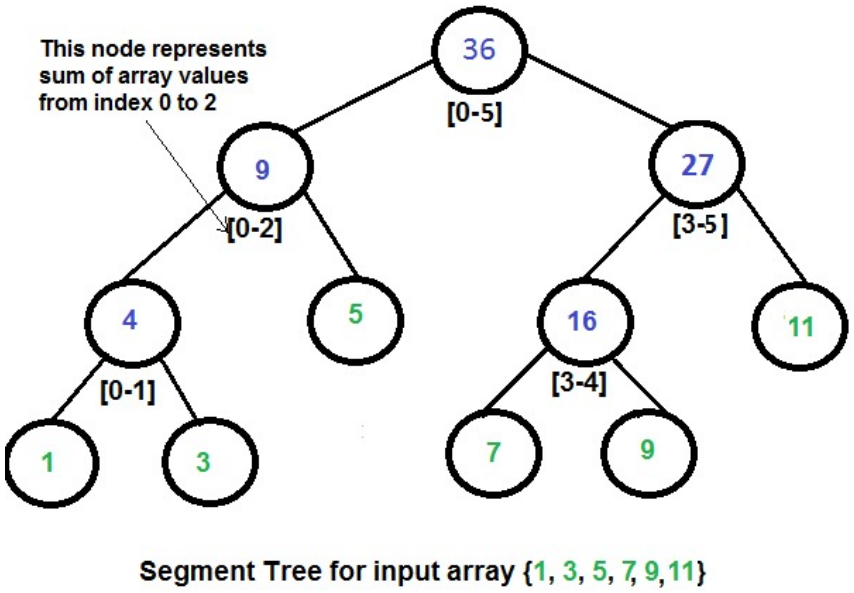

Figure 2: Example Segment Tree

4. Algorithms

4.1. Building the Segment Tree

To build the segment tree, we first initialize the segment tree with the elements of the given array. The remainder of the elements are 0 to ensure the number of elements is a power of 2 for easier implementation.

After initializing the array elements we can complete the segment tree in a bottom-up DP fashion, using the information we have already computed to fill in all the parent nodes.

This is the pre-processing step in a segment tree. As each segment tree index is computed once in constant time the total time complexity is $\Theta(2^{\lceil \log_2 N \rceil}) \approx \Theta(N)$.

Algorithm 1 Building a Segment Tree

BUILD(arr)

```
N = size of arr
for each arr[i] do
    segment tree[i+N] = arr[i]
end for
for i = N-1, N-2, ..., 0 do
    segment tree[i] = Σsegment tree[parent(i)]
end for
```

The implementation of this function in lazy segment is nearly identical. We simply have another array initialized as 0.

```
for all lazy[i] do
    lazy[i] = 0
end for
```

4.2. Updating a Node

Updating a node requires only updating a logarithmic number of nodes, as the nodes which contain the information about i^{th} node are indexed $i, i/2, i/4, ..., 1$. Thus each update takes $O(\log N)$ time.

Algorithm 2 Updating a Node

UPDATE(idx, value)

```
i = idx+N
segment tree[i] += value
i /= 2
while i≠0 do
    segment tree[i] = Σsegment tree[parent(i)]
    i /= 2
end while
```

For the case of lazy segment tree it becomes different. As we postpone the actual update of actual nodes as much as possible. We update the minimum number of nodes possible such that the entire information of the update is still conveyed.

Algorithm 3 Making a Lazy Update

UPDATE (l, r, value, first, last, idx)

```
segment tree[idx] += (last-first+1)lazy[idx]
lazy[parent(idx)] += lazy[idx]
lazy[idx] = 0
if [l, r] ⋃ [first, last] = ∅ then
    return
end if
if [first, last] ⊆ [l, r] then
    segment tree[idx] += (last-first+1)lazy[idx]
    if last ≠ first then
        lazy[parent(idx)] += value
    end if
    return
end if
mid = (first+last)/2
UPDATE (l, r, first, mid, 2 idx)
UPDATE (l, r, mid+1, last, 2 idx+1)
segment tree[idx] = Σsegment tree[parent(idx)]
```

4.3. Range Query

For answering range queries in a segment tree we have two options. Bottom-up approach and Top-down approach similar to the types of DP. Here's an implementation of answering range query using the bottoms up approach.

Algorithm 4 Range Query

QUERY (l, r)

```
sum = 0
l += N
r += N
while l<=r do
   if l is odd then
      sum += segment tree[l]
      l+=1
   end if
   if r is even then
      sum += segment tree[r]
      r-=1
   end if
   l /= 2
   r /= 2
end while
return  sum
```

The lazy implementation however can not be implemented by the bottom-up approach, as we need to take into account the lazy updates.

Algorithm 5 Range Query on Lazy Tree

QUERY (l, r, first, last, idx)

```
segment tree[idx] += (last-first+1)lazy[idx]
lazy[parent(idx)] += lazy[idx]
lazy[idx] = 0
if [l, r] ⋃ [first, last] = ∅ then
   return 0
end if
if [first, last] ⊆ [l, r] then
   return segment tree[idx]
end if
mid = (first+last)/2
return QUERY (l, r, first, mid, 2 idx) + QUERY (l, r, mid+1, last, 2 idx + 1)
```

5. Conclusions

Segment trees are a valuable data structure for solving complex problems that require efficient handling of intervals and segments within a data array. They are commonly used by programmers and computer scientists to optimize a wide range of algorithmic tasks, making them an important topic in the study of data structures and algorithms. Another emerging field where segment trees find their application is that of computational geometry where they are extensively used to solve various geometric problems. As computational geometry involves working with geometric objects and shapes,segment trees are used for solving problems that involve 1D or 2D intervals and geometric ranges.

6. Acknowledgements

We would like to thank our Course Instructor (Dr. Anil Shukla) and Mentor (Manpreet Kaur) for providing us with such a significant opportunity to work on this project. We believe that we will indulge in more such projects in the future. We guarantee that this project was created entirely by us and is not a forgery. Finally, We'd like to express our gratitude to our parents and friends for their excellent comments and guidance during the completion of this project

7. References

1. Wikipedia
2. CP Algorithms
3. CP Algorithmica
4. Planet Math
5. Geeks For Geeks