Go, change the world

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SEGMENT TREE

Data Structures & Application

(IS233AI)

2023-2024

Submitted by

Name 1: MANJUSHREE YADAV D	**USN1: 1RV23CS406**
Name2: NAGAPRASAD NAIK	**USN2: 1RV23CS410**
Name3: MANOJ KUMAR B V	**USN3: 1RV23CS407**

Under the guidance of

Dr. SUMA B
Assistant Professor
Department of CSE
RV College of Engineering

ACKNOWLEDGEMENT

We would like to express our deep sense of gratitude to our respected guide Suma B, Assistant Professor, Dept. of CSE, RVCE for her valuable help and guidance, and are thankful forthe encouragement given to us in completing this Project. We are also grateful to Dr. Ramakanth Kumar P, Head of Department, Dept. of CSE, RVCE for permitting us to utilize the facilities of the department for research and implementation of our project. We are also grateful to our respected Principal, Dr. K N Subramanya. Lastly, we would like to thank our classmates and our parents for providing us with moral support and encouragement.

ABSTRACT

The Segment Tree is a handy tool in computer science for dealing with arrays. It helps quickly find things like the total, smallest, or largest value within a certain section of the array. It works by breaking the array into smaller parts until each part represents just one item. This makes it easy to find stuff fast, and it's really good for making programs run faster. In this paper, a new array based implementation of segment trees is proposed. In such an implementation of segment tree, the structural information associated with the tree nodes can be removed completely. Some primary computational geometry problems such as stabbing counting queries, measure of union of intervals, and maximum clique size of Intervals are used to demonstrate the efficiency of the new array based segment tree implementation. Each interval in a set $S = \{I_1, I_2, \cdots, I_n\}$ of n intervals can be insert into or delete from the heap based segment tree in $O(\log n)$ time. All the primary computational geometry problems can be solved efficiently.

CONTENTS

INTRODUCTION

A Segment Tree, at its core, is a powerful data structure utilized for storing and efficiently querying information about ranges of elements within an array. It offers a structured way to manage and manipulate data within these segments, enabling operations such as range queries and updates in significantly reduced time complexities compared to naïve approaches. This makes it particularly valuable in scenarios where multiple range-based computations need to be performed swiftly, such as in computational geometry, database systems, and algorithmic problem-solving. One of its key features is the ability to perform range-based operations, like summation or finding the minimum or maximum value, across a specified range of elements in the array swiftly, typically in O(log N) time complexity, where N represents the number of elements in the array. Furthermore, the Segment Tree facilitates efficient modifications to the array, allowing for the replacement of individual elements or entire ranges of elements with minimal overhead. In this blog post, we will delve into the fundamental operations of building a Segment Tree from a given array and conducting a range minimum operation, providing insights into its implementation and utility in various real-world applications.

2.1 Purpose and Utility

- Segment Trees are designed to efficiently perform range queries and updates on arrays
- They excel at tasks such as calculating the sum, finding the minimum or maximum value, or performing other operations over specific ranges of elements within the array.
- They excel at tasks such as calculating the sum, finding the minimum or maximum value, or performing other operations over specific ranges of elements within the array.

2.2 Structured Tree Representation

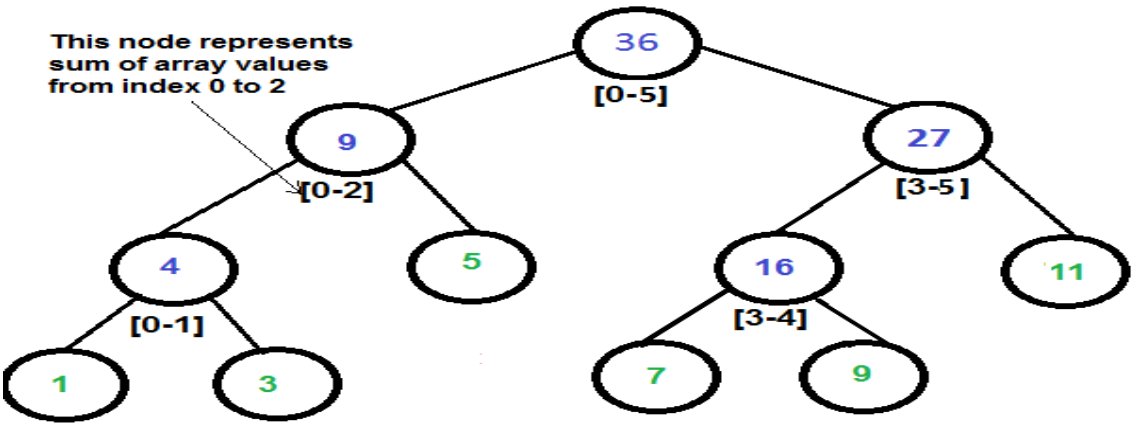
- They excel at tasks such as calculating the sum, finding the minimum or maximum value, or performing other operations over specific ranges of elements within the array.
- This structured representation allows for efficient traversal and manipulation of array segments.

2.3 Structure of the Tree

The segment tree works on the principle of divide and conquer.

- At each level, we divide the array segments into two parts. If the given array had [0, . . ., N-1] elements in it then the two parts of the array will be [0, . . ., N/2-1] and [N/2, . . ., N-1].
- We will then recursively go on until the lower and upper bounds of the range become equal.
- The structure of the segment tree looks like a binary tree.

The segment tree is generally represented using an array where the first value stores the value for the total array range and the child of the node at the ith index are at (2*i + 1) and (2*i + 2).



Segment Tree for input array {1, 3, 5, 7, 9, 11}

2.4 Characteristics of Segment Tree:

- A segment tree is a binary tree with a leaf node for each element in the array.
- Each internal node represents a segment or a range of elements in the array.
- The root node represents the entire array.
- Each node stores information about the segment it represents, such as the sum or minimum of the elements in the segment.

2.5 Types of Segment Tree:

Based on the type of information stored in the nodes, the tree can be divided into the following few types:

- **Sum segment tree:** It stores the sum of elements in each segment of the array and is used in a problem where efficient answer queries are needed about the sum of elements.
- **Min/Max segment tree:** It basically stores the minimum or maximum element in each segment of the array.
- **Range update segment tree / Lazy Tree:** It allows you to update a range of array elements with a given value. It is used to efficiently update a range of array elements

2.6 Types of Operation

The operations that the segment tree can perform must be binary and associative. Additionally, the values operated on must form a semigroup. A semigroup is a set equipped with an associative binary operation. Furthermore, for each operation, there must be an identity element, which acts as a neutral element under that operation. For example, in the case of summation over a range of real numbers (belonging to \(\mathbb{R}\)), the neutral element is 0. Other examples of operations suitable for segment trees include finding the minimum or maximum value in a range. It's important to ensure that the chosen operation and semigroup adhere to these properties to ensure the correctness and efficiency of segment tree operations.

- Addition/Subtraction
- Maximum/Minimum
- GCD/LCM
- AND/OR/XOR

APPLICATIONS

Segment trees find applications in various fields due to their ability to efficiently handle range queries and updates.

- **Interval scheduling:** segment trees are valuable for efficiently scheduling non-overlapping intervals, such as managing appointments or allocating resources.
- **Range-based statistics**: segment trees are instrumental in computing range-based statistics such as variance, standard deviation, and percentiles.
- **Image processing**: segment trees find utility in image processing algorithms where they are utilized to partition an image into segments based on color, texture, or other attributes.

3.1 Advantages:

- **Efficient querying**: segment trees are adept at efficiently answering queries about the minimum, maximum, sum, or other aggregate value of a range of elements in an array.
- **Efficient updates:** segment trees facilitate efficient updates for a range of elements in an array, such as incrementing or decrementing a range of values.
- **Flexibility:** segment trees offer a versatile solution for a wide range of problems involving range queries and updates.

3.2 Disadvantages

- **Complexity**: implementing and maintaining segment trees can be challenging, especially for large arrays or high-dimensional data.
- **Time complexity:** the time complexity of segment tree operations like build, update, and query is o(log n), which may be higher than some other data structures like the Fenwick tree in certain scenarios.
- **Space complexity**: segment trees require a relatively high amount of memory with a space complexity of o(4n).

OPERATIONS

A segment tree is a binary tree in which each node stores information about a specific range, and the nature of that information depends on the problem. In our case, each node will store the minimum value within an interval. However, segment trees can also be used to store various types of information, such as range maximum, range sum, range XOR, and more.

To build a segment tree, we use the array representation of a binary tree. If we follow 0-based indexing, for any node at index i, its children will be found at indices 2i + 1 and 2i + 2. Similarly, with 1-based indexing, the children of a node at index i will be present at indices 2i and 2i + 1. In this report, we will follow 0-based indexing.

Suppose we are given an array arr[] with a size of n and the segment tree st[].

- The root of the segment tree represents the entire range of the array from 0 to n - 1, i.e., arr[0: n-1].
- Leaf nodes of st[] represent individual elements from arr[i].
- Internal nodes represent an interval arr[i: j], where 0 <= i < j < n.

We then break down this interval into two half intervals. The children of the root node represent the intervals arr[0, n/2] and **arr[n/2** + 1, n - 1]. Each interval is further divided into two halves, and the two children of each node represent these halves. Since the segment tree halves in each step, the height of the segment tree is O(log n), and the tree has n leaves, corresponding to the n indices in the array.

4.1 Divide and Conquer Idea to Build Segment Tree:

As we know, both children of a node share the range of the node equally. Therefore, we can employ a divide-and-conquer strategy to build the segment tree. During the divide step, we proceed top-down and recursively divide the range into two halves until there is only one element left in the range.

The formation of the segment tree occurs during the conquer and combine step, which proceeds in a bottom-up fashion, starting from the leaf nodes and moving towards the root node. In other words, it updates the nodes in the path from a leaf to the root in this process, utilizing data from the child nodes to construct a parent node

at each step. This recursive process ultimately reaches the root node, which represents the entire array. It's important to note that the combine step may vary for different types of questions, such as range maximum, range sum, and so on.

4.2 Divide and Conquer steps to build Segment Tree:

We define a function buildSegTree(int st[], int arr[], int nodeIndex, int leftRange, int rightRange) that takes these values as input parameters:

- st[]: Segment tree array.
- arr[]: Input array.
- nodeIndex: Index of the current node in the segment tree.
- leftRange: Left limit of the current segment.
- rightRange: Right limit of the current segment.

Initially, nodeIndex = 0(root node), leftRange = 0, and rightRange = n - 1.

Divide Step: By calculating the mid-index, we divide the array into two halves: leftRange to mid and mid + 1 to rightRange, where mid = leftRange + (rightRange - leftRange)/2.

Conquer Step: We recursively call the same function with modified input parameters to build the segment tree for the left and right half of the array i.e. buildSegTree(st, arr, 2 * nodeIndex + 1, leftRange, mid) and buildSegTree(st, arr, 2 * nodeIndex + 2, mid + 1, rightRange).

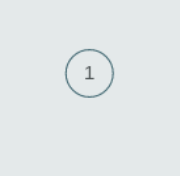
Combine Step: This is the most crucial step. We use the data of the child nodes to decide what information the parent node stores. In our case, we need to find the range minimum. So we take the minimum of the values stored in the child nodes and keep them in the parent node i.e. st[nodeIndex] = min (st[2 * nodeIndex + 1], st[2 * nodeIndex + 2]).

Note: If we need to find the range sum, we will take the sum of the values stored in the child nodes and store it in the parent node.

Base Case: This is the trivial case when leftRange == rightRange, i.e. this represents a single element from the array which is the situation of a leaf node. The segment tree will store this element i.e.
if(leftRange == rightRange), st[nodeIndex] = arr[leftRange].

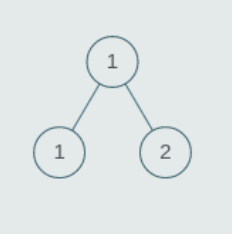
For example, take this array and the corresponding segment tree for finding the minimum in a range, where the size of the array varies from 1 to 5.

Initially, arr = [1], st = [1], size of the tree = 1.



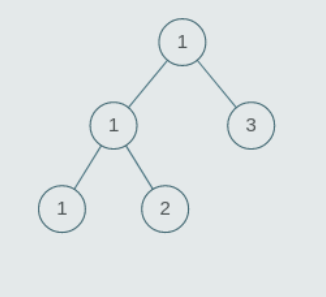
arr = [1, 2], st = [1, 1, 2]

The size of tree = 3 (2n -1, where n = 2).



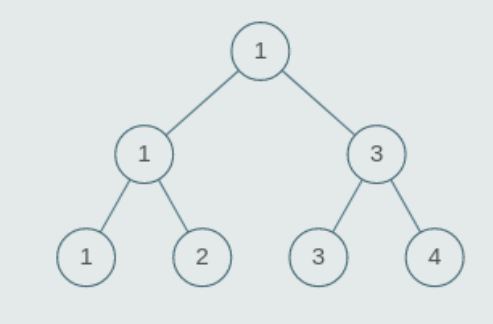
Now n = 3, arr = [1, 2, 3], st =[1, 1, 3, 1, 2, null, null]

- Here n is not a power of 2.
- The smallest power of 2 greater than n = k = 4.
- So the size of the tree = 2 * k - 1 = 2 * 4 - 1 = 7.
- If we observe, the height of the tree increases by 1.



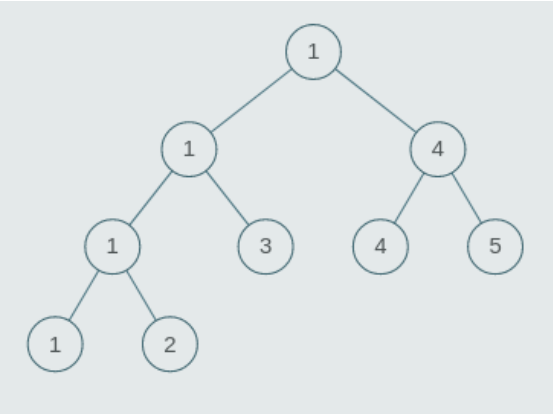
arr = [1, 2, 3, 4], st = [1, 1, 3, 1, 2, 3, 4]

The size of tree = 7 (2n -1, where n = 4).



arr = [1, 2, 3, 4, 5], st =[1, 1, 4, 1, 3, 4, 5, 1, 2, null, null, null, null, null, null]

- Here n is not a power of 2.
- The smallest power of 2 greater than n = k = 8.
- So the size of the tree = 2 * k - 1 = 2 * 8 - 1 = 15.
- The height of the tree again increases by 1 at this point.



4.3 Pseudocode for Building Segment Tree

```
void buildSegTree(int st[], int arr[], int nodeIndex, int leftRange, int rightRange)
{
   if (leftRange == rightRange)
   {
      // Leaf node
      st[nodeIndex] = arr[leftRange]
   }
   else
   {
      int mid = leftRange + (rightRange - leftRange) / 2

      // Recursively building segment tree for left child
      buildSegTree(st, arr, 2 * nodeIndex + 1, leftRange, mid)

      // Recursively building segment tree for right child
      buildSegTree(st, arr, 2 * nodeIndex + 2, mid + 1, rightRange)

      // Union of left and right, which is minimum in our case
      st[nodeIndex] = min(st[2 * nodeIndex + 1], st[2 * nodeIndex + 2])
   }
}
int[] buildSegmentTree(int arr[], int n)
{
   int st[2n - 1]
   for (int i = 0; i < 2n - 1; i = i + 1)
      st[i] = INT_MAX

   buildSegTree(st, arr, 0, 0, n - 1)

   return st
}
```

4.4 Pseudocode of Range Minimum Query

- st[]: Segment tree array.
- nodeIndex: Index of the current node in the segment tree. This value will be 0 Initially.
- leftRange: Left limit of the current node.
- rightRange: Right limit of the current node.
- l: left limit of the given query.
- r: right limit of the given query.

```
int rangeMinQuery(int st[], int nodeIndex, int leftRange, int rightRange, int l, int r)
{
   // Case 2: Range represented by node is completely outside the given range
   if (r < leftRange || rightRange < l)
      return INT_MAX
   // Case 1: Range represented by node is completely inside the given range
   if (l <= leftRange && rightRange <= r)
      return st[nodeIndex]
   // Case 3: Range represented by a node is partially inside and partially outside the given range
   int mid = leftRange + (rightRange - leftRange) / 2
   int leftMin = rangeMinQuery(st, 2 * nodeIndex + 1, leftRange, mid, l, r)
   int rightMin = rangeMinQuery(st, 2 * nodeIndex + 2, mid + 1, rightRange, l, r)
   return min(leftMin, rightMin)
}
```

CONCLUSION

The segment tree is a useful tool for solving range query problems efficiently, offering solutions for tasks like sum, minimum/maximum queries, and updates within intervals. It's versatile, finding applications in computational geometry, database systems, and algorithm design, thanks to its ability to handle dynamic updates and queries. Although it has drawbacks like memory usage and complexity in implementation, these can be managed with optimization techniques. Overall, the segment tree remains a valuable asset for programmers and algorithm designers, providing a balance between storage space and query time while solving a wide array of problems involving range queries and updates.

REFERENCES

1. https://www.geeksforgeeks.org/segment-tree-data-structure/
2. https://www.baeldung.com/cs/segment-trees#:~:text=The%20segment%20tree%20is%20a,structure%20such%20as%20an%20array.
3. https://www.hackerearth.com/practice/data-structures/advanced-data-structures/segment-trees/tutorial/