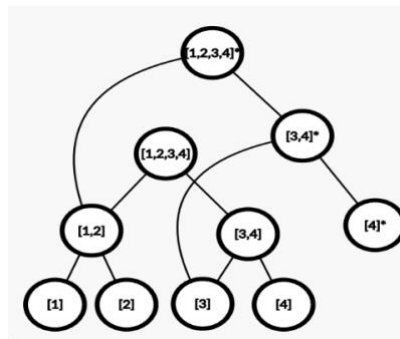




**AMERICAN  
UNIVERSITY<sub>OF</sub> BEIRUT**  
**FACULTY OF ARTS & SCIENCES**

**American University of Beirut  
School of Arts and Sciences  
Department of Computer Science**

## *Persistent Segment Trees with Lazy Propagation*



By  
**Mohammad Khaled Charaf**  
(mmc51@mail.aub.edu)  
**Mohammad Ayman Charaf**  
(mmc50@mail.aub.edu)

A Report  
submitted to Dr. **Amer Abdo Mouawad** in fulfillment of the project for the course  
CMPS 314 – Advanced Data Structures

May 2024

## Abstract:

In this report, we embark on a journey to explore segment trees, unraveling their properties and delving into various exercises that showcase their utility. We delve into the fundamentals of segment trees, including their construction, query operations, and unique applications.

Additionally, we introduce the concept of lazy propagation, a technique essential for ensuring non-commutative operations work in segment trees. Lastly, we delve into the fascinating realm of persistence in segment trees, offering insights into its implementation and significance.

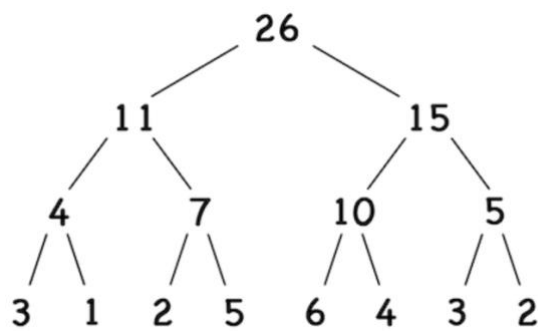
## Table of Contents

<b><i>I- What is a Segment Tree .....</i></b>	<b><i>3</i></b>
1- Definition:.....	3
2- It's Significance.....	4
3- Requirement for $O(\log(n))$ complexity .....	4
<b><i>II- Construction .....</i></b>	<b><i>6</i></b>
1. Analogous to Binary Heap Representation .....	6
2. Query Operation.....	6
<b><i>III- Some Cool Exercises .....</i></b>	<b><i>8</i></b>
1. The Segment with the maximum sum .....	8
2. K-th One.....	9
3. The first element greater than or equal to x .....	10
<b><i>IV- Interval Updates .....</i></b>	<b><i>12</i></b>
1. First Approach.....	12
2. Second Approach .....	13
<b><i>V- Lazy Propagation For Non-commutative Updates .....</i></b>	<b><i>14</i></b>
<b><i>VI- Adding Persistence: .....</i></b>	<b><i>16</i></b>
1- A Cool Persistence Exercise: .....	18
<b><i>13-Instructor's Feedback .....</i></b>	<b><i>19</i></b>

## I- What is a Segment Tree

### 1- Definition:

A Segment Tree is a data structure used for solving problems that involve queries over intervals of an array (l,r) with operations that modify elements within the array. Constructing it involves padding the array with zeros at the end of the array until the size of the array becomes a power of 2. This is needed to construct a complete binary tree over the array where the elements are the leaves of the binary tree. The space complexity involved is  $O(n)$ . This is because padding the array with zeros will at most double the size of the array. Moreover, the number of non-leaf nodes in a complete binary tree is one less than the number of leaves.



Each node within the segment tree reflects its segment i.e. the leaves at the tree rooted at that node.

Notice here, that the sum of the segment is maintained at each node.

## 2- It's Significance

In the context of array manipulation, the significance of segment trees lies in their ability to efficiently handle both update and query operations.

Consider the problem with updates(Change  $A[i]$  to value  $v$ ) and queries(Sum over the interval  $(i,j)$ ) where an array  $A$  undergoes frequent updates at various indices ( $0 \leq i < n$ ). In such cases, direct access allows for direct updates with constant time complexity. However, if we were to adopt a linear approach to compute the sum of a particular interval  $(l, r)$ , it would become inefficient as the number of queries increases.

Conversely, in scenarios where there are numerous queries on the array, computing a prefix sum array for every prefix in the array can facilitate efficient query operations with constant time complexity. However, this approach falls short when updates are required, as updating necessitates recomputing all prefix sums, resulting in linear time complexity.

Segment trees offer a balanced solution to these challenges. By constructing a segment tree for the array, both update and query operations can be performed in logarithmic time. The segment tree's structure facilitates efficient update operations by precomputing interval sums while still allowing for rapid queries.

## 3- Requirement for $O(\log(n))$ complexity

To ensure that segment tree operations achieve a time complexity of  $O(\log(n))$ , several requirements must be met:

1. **Merge Function for Queries:** The merge function, denoted as  $f$ , must satisfy the property that  $\text{Query}(x, A \cup B) = f(\text{Query}(x, A), \text{Query}(x, B))$ , where  $A$  and  $B$  represent two disjoint intervals and  $x$  is the query point(interval) within those intervals. This property ensures that the result of querying a combined interval can be efficiently computed from the results of querying its constituent intervals. In addition, we require  $f$  to be a constant time function.
2. **Commutativity and Lazy Propagation:** Commutativity, the property that the order of operations does not affect the result, is essential. However, with Lazy Propagation non-commutative operations can be applied within logarithmic time.
3. **Associativity:** Another crucial requirement is associativity. This property ensures that the grouping of operations does not affect the final result. In the context of segment trees, it means that for any element  $A[i]$  and operations  $op$ , the following holds true:  $A[i] \text{ op } (X \text{ op } Y) = ((A[i] \text{ op } X) \text{ op } Y)$ . This property enables correct aggregation of updates and queries within the segment tree structure.

By meeting these requirements, segment trees can efficiently handle both updates and queries on intervals of an array, achieving a time complexity of  $O(\log(n))$ . This makes them a valuable data structure for a wide range of applications, including range queries and updates in various algorithms and problems.  $f$  may not necessarily be a constant time function for applications such as queries for computing the GCD over an interval of the array.

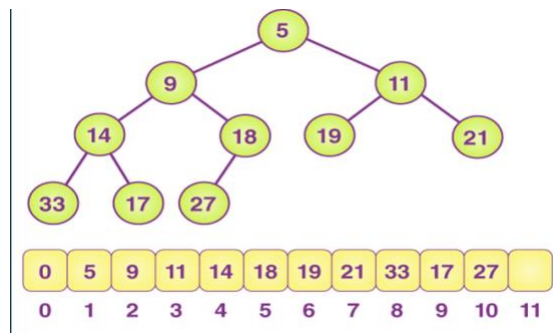
## II- Construction

### 1. Analogous to Binary Heap Representation

Segment tree construction can be approached similarly to the construction of a binary heap, with the added flexibility of potential future persistence. To construct a segment tree, the given array  $A$  is padded with zeros to reach the next power of 2, ensuring that the resulting segment tree is complete. This padding incurs a space complexity of  $O(n)$ .

For time complexity considerations, if the initial state of the array is inserted through consecutive updates, the time complexity becomes  $O(n \log n)$ . This is because each update operation propagates through the leaf-to-root path, taking  $O(\log n)$  time per update, resulting in a total time complexity of  $O(n \log n)$ .

Alternatively, segment tree construction can be optimized to achieve a worst-case time complexity of  $O(n)$  by filling up the tree level by level from leaf to root. This approach ensures that each element of the array is efficiently incorporated into the segment tree structure, resulting in a linear time complexity.



### 2. Query Operation

Let QI represent the queried interval, CI denotes the current interval, and A denotes the segment tree.

1. If CI lies entirely within QI, we simply return the value stored at  $A[i]$ .
2. If CI lies entirely outside QI, we return the identity element or nothing, signifying that the queried interval does not overlap with the current interval.
3. The most significant case occurs when CI intersects with QI. In this scenario, we recursively query the left and right child intervals, then merge their results using the merge function  $f$ .

## III- Some Cool Exercises

To deepen our understanding of segment trees and their practical applications, we engaged in solving various exercises sourced from a competitive programming platform called CODEFORCES. These exercises were carefully selected to explore different aspects of segment tree usage, including its application, versatility, and functionality.

### 1. The Segment with the maximum sum

#### **Operations:**

set(i, v): set the element with index i to v

max\_segment(): find the segment of the array with the maximum sum.

---

To tackle the problem of finding the segment with the maximum sum and supporting set operations, a divide-and-conquer approach proves effective. This strategy involves recursively identifying the left and right maximum sums. A potential location for the maximal sum lies at the intersection of both halves.

To maintain the maximal prefix and suffix sums, along with the entire segment sum, we employ a method that involves storing these values at each node of the segment tree.

Here's a summary of the approach:



At each node, maintain values for maximal sum, prefix sum, suffix sum, and entire segment sum. For a segment with halves (h1, h2), the maximal sum can be determined as follows:

$$\text{Maximal sum} = \max(\text{h1.maximalSum}, \text{h2.maximalSum}, \text{h1.suffix} + \text{h2.prefix})$$

Update the maximal prefix and suffix sums as follows:

$$\text{Maximal Prefix Sum} = \max(\text{Maximal Prefix Sum}, \text{h1.entireSum} + \text{h2.prefixSum})$$

$$\text{Maximal Suffix Sum} = \max(\text{Maximal Suffix Sum}, \text{h2.entireSum} + \text{h1.suffixSum})$$

Maintain the entire segment sum as the sum of the two halves: **Entire Sum = h1.entireSum + h2.entireSum**

By storing and updating these values at each node, the problem can be efficiently solved in  $O(\log n)$  time, making it suitable for segment tree implementation.

## 2. K-th One

### Operations:

set(i, v): Set element i to v where  $v \in \{0,1\}$

find(k): Find the index of the k-th one.

---

To address the problem of finding the index of the k-th one in a segment where elements can only be 0 or 1, we can employ a segment tree and maintain the sum of each segment at each node.

The following pseudo-code demonstrates a solution to the problem:

**Find(node, k):**

**If (node.left.sum < k):**

**Find(node.left, k)**

**Else:**

**Find(node.right, k - node.left.sum)**

This algorithm works by recursively traversing the segment tree. If the sum of the left child node is less than k, the algorithm recursively searches the left subtree. Otherwise, it searches the right subtree with an adjusted k value.

This approach ensures logarithmic time complexity due to having only one recursive call at each depth of the tree. As a result, the problem of finding the index of the k-th one can be efficiently solved using this method.

### **3. The first element greater than or equal to x**

#### **Operations:**

set(i, v): set element i to v

first\_above(x): find the leftmost element greater than or equal to x

---

To find the leftmost element greater than or equal to x in a segment tree, we can leverage the property of maintaining the maximum element at each node. Here's how the problem can be approached:

1. Start from the root of the segment tree.
2. If the maximum element of the current node is less than  $x$ , it implies that all elements in the subtree rooted at this node are less than  $x$ . Therefore, we recursively explore the right child.
3. If the maximum element of the left child is greater than or equal to  $x$ , it indicates that there exists an element in the left subtree that is greater than or equal to  $x$ . Thus, we recursively explore the left child.
4. If neither of the above conditions holds, it means that the current node itself is the leftmost element greater than or equal to  $x$ .

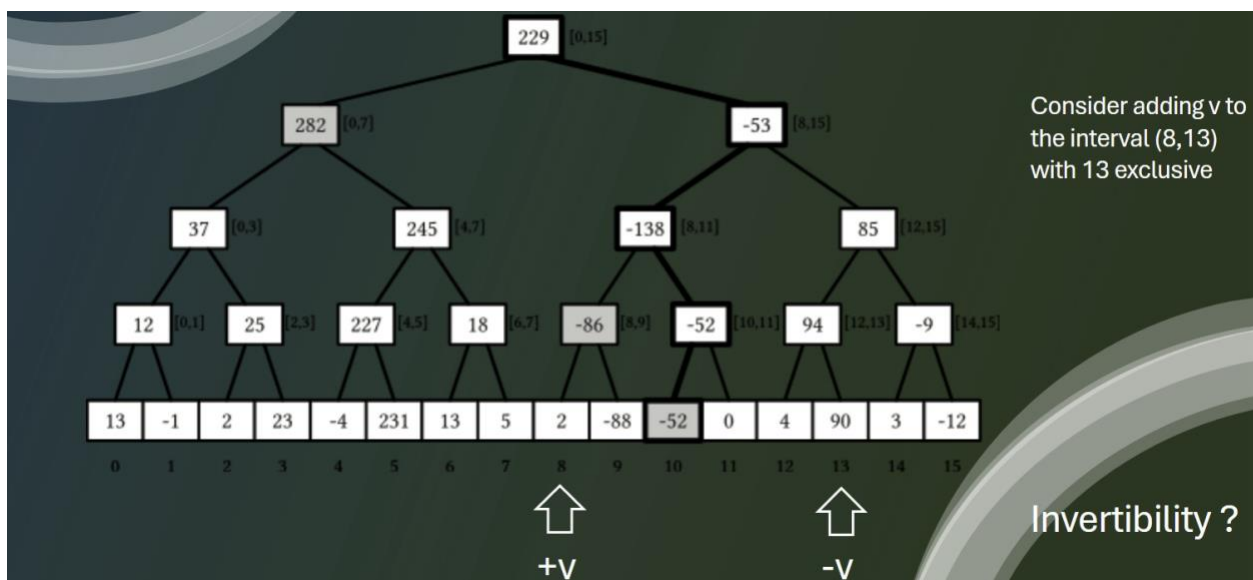
By following this recursive approach, we can efficiently locate the leftmost element greater than or equal to  $x$  in the segment tree. The logarithmic time complexity is guaranteed due to having only one recursive call at each depth of the tree.

## IV- Interval Updates

In the previous section, we explored updating a single array via propagating changes upwards using a function  $f$ . Now, let's consider a scenario where we're given an array  $A$ , and we need to perform two operations: adding a number  $v$  to a segment from index  $l$  to  $r-1$ , and querying for the current value of element  $I$  after that operation.

### 1. First Approach

we adopt a technique where we add  $+v$  to the element at index  $l$  and  $-v$  for the element at index  $r$ , allowing these changes to propagate upwards through the segment tree. To answer queries about the current value of element  $I$ , we calculate the prefix sum up to the index  $I$ . If  $I$  falls between  $l$  and  $r$ , then  $v$  will be added to the sum; if it is to the left of  $l$ , no change will occur; and if  $I$  is to the right of  $r$ , the  $-v$  will cancel out the  $+v$ , resulting in no net change. This approach effectively implements an interval update from  $l$  to  $r$  but requires invertibility for the operation.



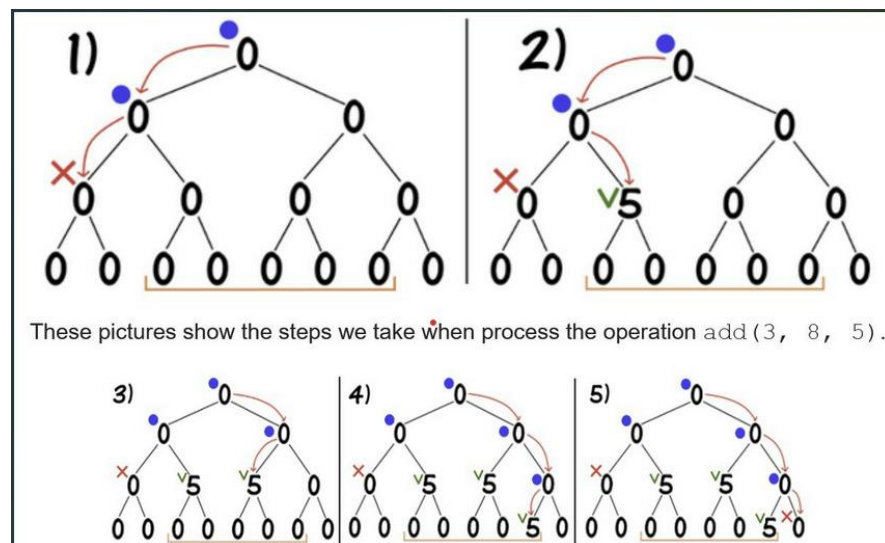
## 2. Second Approach

For our second approach, we opted to decompose the interval into logarithmic sub-intervals, akin to the query operation. Each node will then be updated individually in  $O(\log n)$  time. However, this approach relies on certain implicit assumptions.

Firstly, it assumes that the function  $F$  used for updating is of  $O(1)$  time complexity, ensuring that the update operation at each node remains efficient.

Secondly, the update operations across nodes are spread along the root-to-leaf path. We require the operation which is addition in the above case to be commutative

Lastly, it presumes that the update operation is both commutative and associative. These properties are crucial for ensuring that the order of updates does not affect the final result, allowing for consistent and reliable segment tree operations.

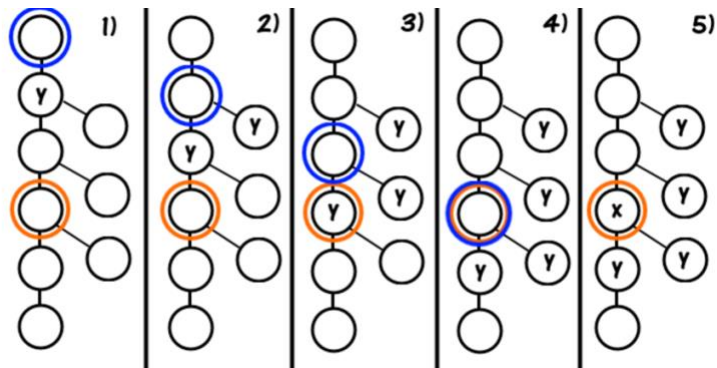
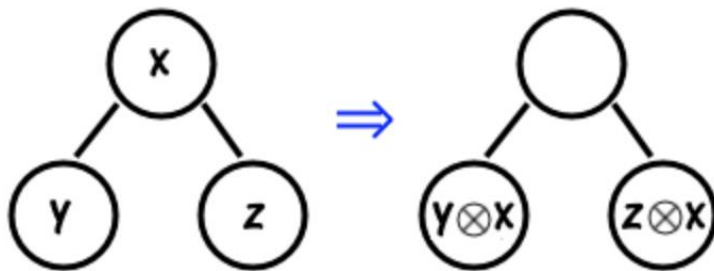
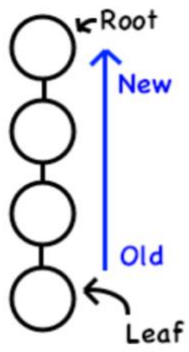


## V- Lazy Propagation For Non-commutative Updates

If the operation in the modify request is not commutative, we will employ the lazy propagation technique to maintain efficiency and correctness. Lazy propagation allows us to defer updates and propagate them only when necessary, ensuring that operations are applied in the correct order.

When handling non-commutative operations, we need to maintain the order of operations by pushing old operations deeper into the tree. Specifically, when we enter a node, we nullify the operation stored at the parent and propagate it to its children. This ensures that any pending updates are moved down to the appropriate nodes. The pending updates are stored in a lazy array associated with each node, indicating which operations need to be applied when the node is accessed.

As we propagate the changes to the children, we ensure that the operations are applied in the correct sequence. Upon exiting the node, we recalculate its value based on the updated values of its children. This method ensures that each node correctly reflects all operations performed on its segment, maintaining the overall efficiency and correctness of the segment tree. By using lazy propagation in this manner, we can handle interval updates effectively, even when dealing with non-commutative operations, ensuring logarithmic time complexity for update operations.



## VI- Adding Persistence:

In the exploration of segment trees and their persistence, one approach considered is modeling segment tree nodes with necessary back pointers. However, this approach can introduce unnecessary overhead and complexity.

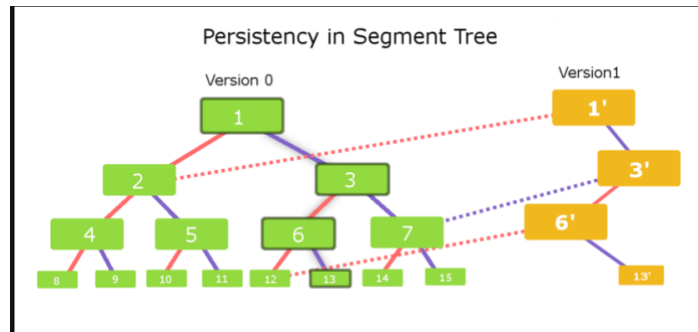
An alternative strategy involves recognizing that nearly every operation in a segment tree involves propagating changes to the root, passing through  $O(\log n)$  nodes. By focusing on this aspect, a more efficient strategy to track versions of the tree can be devised.

When applying an operation to a particular node, only a subset of nodes along the path to the root are affected. For instance, when operating on a specific node, only the nodes along its path to the root are impacted. By recursing downwards from the modified node, it is possible to create the necessary nodes to link to the root, effectively achieving persistence.


To implement this, an array of versions is maintained, with each version represented by a root node containing the modified nodes. This ensures that each change incurs only a constant multiplicative overhead in terms of time and space, making it feasible to track and manage multiple versions of the segment tree efficiently.


By strategically tracking versions of the segment tree and efficiently managing modifications, persistence can be incorporated into segment trees while maintaining optimal time and space complexity. This approach offers a practical and scalable solution for handling dynamic updates and queries in segment trees .





## 1- A Cool Persistence Exercise:

 [Range Queries and Copies](#)

1851 / 2009 

TASK | [SUBMIT](#) | [RESULTS](#) | [STATISTICS](#) | [TESTS](#) | [QUEUE](#)

**Time limit:** 1.00 s   **Memory limit:** 512 MB

Your task is to maintain a list of arrays which initially has a single array. You have to process the following types of queries:

1. Set the value  $a$  in array  $k$  to  $x$ .
2. Calculate the sum of values in range  $[a, b]$  in array  $k$ .
3. Create a copy of array  $k$  and add it to the end of the list.

**Input**

The first input line has two integers  $n$  and  $q$ : the array size and the number of queries.

The next line has  $n$  integers  $t_1, t_2, \dots, t_n$ : the initial contents of the array.

Finally, there are  $q$  lines describing the queries. The format of each line is one of the following: " $1 \ k \ a \ x$ ", " $2 \ k \ a \ b$ " or " $3 \ k$ ".

**Output**

Print the answer to each sum query.

**Constraints**

- $1 \leq n, q \leq 2 \cdot 10^5$
- $1 \leq t_i, x \leq 10^9$
- $1 \leq a \leq b \leq n$

**Example**

**Range Queries**

...

<a href="#">Pizzeria Queries</a>	-
<a href="#">Subarray Sum Queries</a>	-
<a href="#">Distinct Values Queries</a>	-
<a href="#">Increasing Array Queries</a>	-
<a href="#">Forest Queries II</a>	-
<a href="#">Range Updates and Sums</a>	-
<a href="#">Polynomial Queries</a>	-
<a href="#">Range Queries and Copies</a>	✓

**Your submissions**

2024-05-11 00:17:53	✓
2024-05-11 00:17:32	✗
2024-05-11 00:14:36	✗

This exercise involves implementing a persistent segment tree to manage a dynamic list of arrays, necessitating efficient updates, range sum queries, and array duplication. Persistence is essential in this context to maintain multiple versions of arrays over time, allowing modifications without redundancy. The problem requires handling up to 200,000 operations on arrays of size up to 200,000, with elements potentially as large as 109109. To achieve this, we use a node-based segment tree structure where each node stores segment sums and pointers to child nodes. Initial construction splits the array recursively, combining results to form parent nodes. For updates, new versions of the tree are created by modifying only necessary nodes along the path from the root to the target leaf, preserving previous versions. We maintain an array of root versions, each pointing to the root of a segment tree corresponding to a specific

version. This strategy ensures efficient versioning and allows for swift duplication and query operations, highlighting the practicality of persistent data structures in applications requiring dynamic updates.

### **13-Instructor's Feedback**