Persistent Segment Trees

CS 4501: Advanced Algorithms and Implementation, Spring 2025

Riley Immel[1], **Windsor Antal**[1], **Brandon Yang**[1]

[1]University of Virginia

Date: April 23, 2025
Code: https://github.com/rileyimmel/persistentSegTreesProject

1 Introduction

Recall when we first introduced the concept of a **Segment Tree** in class. The Segment Tree is a data structure that allows us to efficiently query and update ranges of values in an array. Formally, given a list of integers $A = \{a_1, a_2, \ldots, a_n\}$, and a function $f(l, r)$ that operates on a continuous subsegment $A[l \ldots r]$, the Segment Tree provides two key operations. The first operation is a range query, which calculates $f(l, r) = f(a_l, \ldots, a_r)$ for any given indices l and r $(l \leq r)$ in $O(\log n)$ time. The second operation is a point update, which updates the value of an element a_i at any index i in $O(\log n)$ time. Overall, we can build a Segment Tree in $O(n)$ time, and it uses $O(n)$ space.

Suppose we want to maintain a Segment Tree for a list of integers $A = \{a_1, a_2, \ldots, a_n\}$, but we also want to keep track of the history of updates made to the array. In other words, we want to be able to query the value of a_i at *any* point in time, not just the current value. This is where the concept of a Persistent Segment Tree comes into play.

Formally, a **Persistent Segment Tree** is a data structure that extends the Segment Tree to support *versioned* range queries and updates, allowing access to any past state of the array efficiently. Given an initial array $A = \{a_1, a_2, \ldots, a_n\}$, the Persistent Segment Tree maintains a sequence of immutable versions $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_k$, where each version $\mathcal{T}_i$ corresponds to the state of the Segment Tree after i updates. This is achieved via *path copying*, where only $O(\log n)$ nodes are copied for each update. The Persistent Segment Tree supports the following operations:

- **Build:** Constructs the initial tree $\mathcal{T}_0$ from the array A in $O(n)$ time and space.
- **Persistent Update:** Given a version $\mathcal{T}_i$ and an index $p \in [1, n]$, and a new value v, creates a new version $\mathcal{T}_{i+1}$ such that a_p is replaced with v in $\mathcal{T}_{i+1}$, while all other values remain unchanged. This operation takes $O(\log n)$ time and $O(\log n)$ space.
- **Range Query:** Given a version $\mathcal{T}_i$ and indices $l, r \in [1, n]$, computes $f(l, r)$ in $O(\log n)$ time. This operation does not require any additional space.
- **Version Access:** Allows retrieval and independent querying of any previous version $\mathcal{T}_j$ for $j \leq i$. This operation takes $O(1)$ time and space.

2 Background

Segment Tree Recap

A Segment Tree is a binary tree structure built over an array A. Each node in the tree corresponds to a specific range $[l, r]$ of indices in the original array and typically stores the precomputed value of the function f over the subsegment $A[l \ldots r]$. We define the Segment Tree $\mathcal{T}$ recursively as follows:

- **Base Case (Leaf Node):** If $l = r$, then the node labeled as $a[l \ldots r]$ stores:

$$\text{sum} = A[l]$$

- **Recursive Case (Internal Node):** If $l < r$, let $m = \left\lfloor \frac{l+r}{2} \right\rfloor$. Then the node labeled as $a[l \dots r]$ stores:

$$\text{sum} = f\,(\text{left child sum},\ \text{right child sum})$$

The function f determines how two child nodes are combined. For example, in the sum Segment Tree shown in Figure 1, $f(x, y) = x + y$.

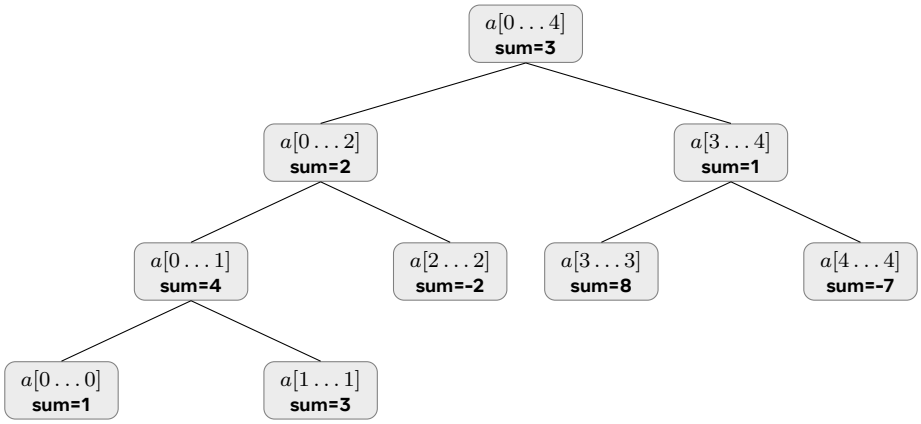

Figure 1 Segment Tree for $A = [1, 3, -2, 8, -7]$ with sum aggregation. Each node stores the range $[l, r]$ and the corresponding aggregated value, which is the sum of the values in that range. The root node represents the entire array, while the leaf nodes represent individual elements. The internal nodes represent the sum of their respective child nodes.

A standard segment tree supports the following operations:

- **Build (Algorithm 2):** Constructs the Segment Tree in $O(n)$ time.
- **Range Query (Algorithm 3):** Computes $f(l, r)$ in $O(\log n)$ time.
- **Point Update (Algorithm 4):** Updates the value at index i in $O(\log n)$ time.

3 Approach

The key idea behind the Persistent Segment Tree is that we want to preserve all previous versions of the Segment Tree while still allowing efficient updates and queries. A naïve way to do this would be to copy the entire tree every time we perform an update, but this would lead to an inefficient $O(n)$ time and space cost per update. Instead, we use a technique called **path copying**, which ensures that only $O(\log n)$ new nodes are created per update, while the rest of the structure is shared.

Each time we perform a point update on the Persistent Segment Tree, we create a new version of the tree that reflects the update, while keeping all previous versions intact. Rather than constructing a completely new tree, we reuse all parts of the existing tree that remain unchanged. The only nodes that need to be recreated are those that lie on the path from the root to the updated leaf. This is done by creating a new node for the the updated value and recursively copying the necessary nodes along the path to the root. The rest of the tree remains unchanged, allowing us to maintain a history of all previous versions.

Each version of the Segment Tree is identified by a unique root node. These root nodes are stored externally, so that querying any historical version simply requires passing the corresponding root into a standard query function. All queries and updates remain $O(\log n)$, just like in a regular Segment Tree.

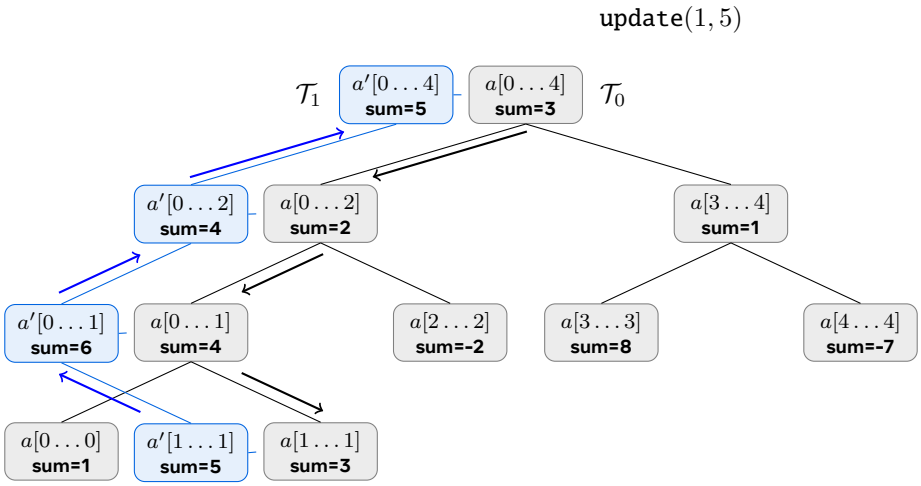

Figure 2 Path Copying Illustration: Given Segment Tree $\mathcal{T}_0$ for $A = [1, 3, -2, 8, -7]$ with sum aggregation, we perform a persistent update to change the value at index 1 from 3 to 5. This creates a new version $\mathcal{T}_1$ by copying only the nodes along the path from the root to the updated leaf. All other nodes are shared with the original version. Nodes created for $\mathcal{T}_1$ are shown in blue, and shared nodes are connected by dashed edges.

Figure 2 illustrates the path copying mechanism when an update occurs. The tree on the left, T_0, represents the original Segment Tree. When an update is applied, a new version T_1 is created. Nodes that are not affected by the update (i.e., not on the path to the updated index) are shared between the two versions. Only the nodes along the update path are newly allocated and updated.

This design makes the Persistent Segment Tree a powerful tool for problems that require historical access, undo functionality, or multi-version computations. In the next section, we will describe the detailed recursive implementation of the Persistent Segment Tree, including how versions are stored, how queries are performed, and how updates generate new roots with minimal copying.

4 Implementation

We showcase the full solution in Python (B) in the appendix, and we also have Java and C++ implementations available in the GitHub repository.

Given an array $A = [a_1, a_2, \ldots, a_n]$, we first build a standard Segment Tree $\mathcal{T}_0$ in $O(n)$ time. We also need to initialize a list of roots $\mathcal{R} = [\mathcal{T}_0]$ to store the roots of all versions.

Persistent Update Algorithm

Let `Node` represent a node in the Segment Tree, containing:

- `val`: the aggregate value over the range $[l, r]$,
- `left`, `right`: pointers to the left and right child nodes.

Let $f : V \times V \to V$ be an associative *merge function* (e.g., sum, min, max) used to combine the results from the left and right subtrees. We define the update procedure `update`(`node`, l, r, p, v), where:

- `node` is the root of the current version,
- $[l, r]$ is the range that `node` covers,
- p is the index to update,
- v is the new value to assign at index p.

We demonstrate the update algorithm in Algorithm 1.

Algorithm 1 Persistent Segment Tree Update

```
function UPDATE(node, l, r, p, v)
    if l = r = p then
        return NODE(v, None, None)
    end if
    m ← ⌊(l + r)/2⌋
    if p ≤ m then
        left ← UPDATE(node.left, l, m, p, v)
        right ← node.right
    else
        left ← node.left
        right ← UPDATE(node.right, m + 1, r, p, v)
    end if
    return NODE(f(left.val, right.val), left, right)
end function
```

As demonstrated in Algorithm 1, the update function that operates on $\mathcal{T}_t$ eventually returns the new root node of the updated version $\mathcal{T}_{t+1}$. We then simply apppend this new root to the list of roots $\mathcal{R}$, so that we can access it later.

The update procedure has a worst-case time and space complexity of $\Theta(\log n)$ per operation. This is because the algorithm only creates new nodes along the path from the root to the target leaf node during an update, and the height of the balanced Segment Tree is $O(\log n)$. Every recursive call in this path results in the creation of one new node, while all other parts of the tree are reused through structural sharing. Thus, the number of nodes created is exactly proportional to the depth of the tree, which is $\Theta(\log n)$ in the best, worst, and average cases. The use of persistent data structures here avoids full copying and enables access to any prior version in constant time via stored root pointers, without compromising the logarithmic efficiency of updates or queries.

Range Query and Version Access

To query any version $\mathcal{T}_k$, we simply pass its root into the standard Segment Tree range query function. Since version roots are stored externally, accessing a particular version takes $O(1)$ time (assuming that we query the i^{th} version/edit), and the query itself runs in $O(\log n)$ time as usual.

5 Summary of Programming Challenge

ChronoCorp, the world's most valuable tech company, utilizes its innovative Desk Matrix™ system to track employee identities and their evaluation scores across different versions as they move between desks, departments, or rejoin the company. The system records every change, creating a history of who worked at each desk and their corresponding evaluations. The input describes the initial employee evaluation scores for each desk and a series of commands. These commands include swapping employees between two desks (S), replacing an employee at a specific desk with a new one and a new score (R), replacing an employee at a desk with the highest-scoring employee who ever occupied that desk (P), and querying a range of desks for the sum of employee evaluations in two specified versions of the desk matrix (Q). For each query, the output should be the absolute difference between the sums of the evaluation scores in the given range for the two specified versions. The swap, replacement, and perfect commands do not produce any output.

The key twist in the assignment is handling not just value updates (replacements), but also **nontrivial operations like swaps and max-history-based replacements** ("perfect"), while keeping the structure efficiently queryable across versions. This goes beyond basic persistent trees by requiring students to design a strategy to manage shared structure with non-monotonic changes.

Key Ideas for Solving Programming Challenge

To solve this challenge, students must use a **persistent segment tree** to store historical versions of evaluation scores, enabling range queries across any pair of versions. The main challenge lies in supporting:

- **Swaps**, which affect two positions and break the naïve one-path update assumption.
- **Perfect replacements**, which require maintaining per-index history of maximums.

These challenges require students to reason about how to store auxiliary data (like max-history per index) and adapt standard persistence techniques. The final solution has $O(\log n)$ update and query time per operation, and $O(n + c \log n)$ space for all versions. The problem is rich enough to reinforce key data structure principles without exceeding the capabilities of an upper-level undergraduate student.

6 Conclusion

We discuss the implementation of a Persistent Segment Tree, a powerful data structure that allows efficient updates and queries across multiple versions of an array. The key idea is to use path copying to minimize the space and time complexity of updates while maintaining the ability to access any previous version. This design is particularly useful for problems that require historical access or multi-version computations.

We also designed a programming challenge, "Desks of Eternity," to reinforce the concepts of persistent data structures and their applications in real-world scenarios. The challenge requires students to implement a system that tracks dynamic updates to an array while maintaining full historical access to any prior state, using a Persistent Segment Tree.

A Standard Segment Tree Operations

Algorithm 2 Segment Tree Build

```
function BUILD(node, l, r)
    if l = r then
        tree[node] ← A[l]
        return
    end if
    m ← ⌊(l + r)/2⌋
    BUILD(2 · node, l, m)
    BUILD(2 · node + 1, m + 1, r)
    tree[node] ← f(tree[2 · node], tree[2 · node + 1])
end function
```

Algorithm 3 Segment Tree Range Query

```
function QUERY(node, l, r, ql, qr)
    if qr < l or ql > r then
        return 0                                   ▷ Out of range
    end if
    if ql ≤ l and r ≤ qr then
        return tree[node]                          ▷ Fully within range
    end if
    m ← ⌊(l + r)/2⌋
    x ← QUERY(2 · node, l, m, ql, qr)
    y ← QUERY(2 · node + 1, m + 1, r, ql, qr)
    return f(x, y)
end function
```

Algorithm 4 Segment Tree Point Update

```
function UPDATE(node, l, r, i, v)
    if l = r = i then
        tree[node] ← v
        return
    end if
    m ← ⌊(l + r)/2⌋
    if i ≤ m then
        UPDATE(2 · node, l, m, i, v)
    else
        UPDATE(2 · node + 1, m + 1, r, i, v)
    end if
    tree[node] ← f(tree[2 · node], tree[2 · node + 1])
end function
```

B Python Implementation of Persistent Segment Tree

Code 1: Python Implementation of Persistent Segment Tree

```python
class Node:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.leftChild = left
        self.rightChild = right


class PST:
    MAX = 100

    def __init__(self, n):
        self.n = n
        self.version = 0
        self.root = [None] * self.MAX

    def construct(self, val):
        self.root[0] = self._construct(val, 0, self.n - 1)

    def _construct(self, val, l, r):
        node = Node()
        if l == r:
            node.value = val[l] if val else 0
            node.maximum = val[l] if val else 0
            return node
        middle = (l + r) // 2
        node.leftChild = self._construct(val, l, middle)
        node.rightChild = self._construct(val, middle + 1, r)
        node.value = node.leftChild.value + node.rightChild.value
        return node

    def _update(self, l, r, prev, idx, new_val):
        node = Node()
        if l == r:
            node.value = new_val
        else:
            middle = (l + r) // 2
            if idx <= middle:
                node.leftChild = self._update(l, middle, prev.leftChild, idx, new_val)
                node.rightChild = prev.rightChild
            else:
                node.rightChild = self._update(
                    middle + 1, r, prev.rightChild, idx, new_val
                )
                node.leftChild = prev.leftChild
            node.value = node.leftChild.value + node.rightChild.value
        return node

    def _query(self, node, l, r, ql, qr):
        if l > qr or r < ql:
            return 0
        if ql <= l and qr >= r:
            return node.value
        middle = (l + r) // 2
        return self._query(node.leftChild, l, middle, ql, qr) + self._query(
            node.rightChild, middle + 1, r, ql, qr
        )

    def update(self, idx, val):
        self.version += 1
        self.root[self.version] = self._update(
            0, self.n - 1, self.root[self.version - 1], idx, val
        )

    def query(self, query_version, l, r):
        return self._query(self.root[query_version], 0, self.n - 1, l, r)
```