Time and Space Complexity Analysis of Build, Update, and Query Operations in Persistent Segment Tree Data Structure

Jeremy Gerald Sutanto - 13525104
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: contact@i-jer.com , 13525104@std.stei.itb.ac.id

***Abstract*—Segment tree is a tree-based data structure commonly used to answer range queries and perform updates efficiently on an array. However, a regular segment tree only maintains the latest state of the data, so previous states are lost after an update is performed. This limitation can be solved by using a persistent segment tree, which preserves older versions of the structure through path copying. This paper analyzes the time and space complexity of build, update, and query operations in a persistent segment tree. The analysis begins by reviewing the regular segment tree, then continues with the persistent version and its versioning mechanism. The result shows that the initial build operation requires linear time and space, while each point update requires logarithmic time and logarithmic additional space. Range queries on any version can still be performed in logarithmic time in the worst case. Therefore, persistent segment tree provides access to historical versions while preserving efficient operation complexity.**

Keywords—persistent segment tree; segment tree; path copying; time complexity; space complexity; data structure persistence; range query

I. Introduction

Efficient range processing is a common requirement in computational problems. Given an array, a program may need to repeatedly answer queries such as range sum, minimum, or maximum, while also supporting updates to individual elements. A naive approach scans the queried interval directly, which becomes inefficient when the array size and the number of operations are large.

A segment tree is a binary tree data structure that solves this problem by storing information about array intervals. The root represents the whole array, while each child represents a smaller subinterval. With this structure, point updates and range queries can be performed efficiently because the algorithm only visits relevant nodes instead of processing the whole array.

However, a regular segment tree only stores the latest version of the data. When an update is performed, the affected nodes are overwritten, so previous versions can no longer be accessed. This becomes a limitation when a problem requires queries on historical states of the array. Copying the entire tree after every update is possible, but it is inefficient in both time and memory.

A persistent segment tree solves this limitation by preserving old versions through path copying. During a point update, only the nodes along the path from the root to the updated leaf are copied, while the unaffected subtrees are shared between versions. This allows multiple versions of the segment tree to exist without rebuilding or copying the entire structure.

This paper analyzes the time and space complexity of build, update, and query operations in a persistent segment tree. The analysis focuses on point updates and range sum queries. It also compares the persistent version with the regular segment tree to show the tradeoff between historical access and additional memory usage.

II. THEORETICAL FOUNDATION

A. Algorithm Complexity

Algorithm complexity is used to measure how the resource usage of an algorithm grows as the input size increases. The two most common resources analyzed are time and space. Time complexity describes the number of basic operations performed by an algorithm, while space complexity describes the amount of memory required during execution.

In this paper, the input size is denoted by N, where N represents the number of elements in the initial array. The number of update operations is denoted by U. The number of range queries is denoted by Q. The analysis uses asymptotic notation to describe the growth of each operation. Big-O notation is used for upper bounds, Big-Omega notation is used for lower bounds, and Big-Theta notation is used when the upper and lower bounds are asymptotically equal.

B. Binary Tree

A binary tree is a tree data structure in which each node has at most two children. These children are usually called the left child and the right child. Segment tree is based on this structure because each interval can be divided into two smaller intervals. The root node represents the whole interval, and each level below it represents smaller subintervals.

For an array of size N, the height of a balanced binary tree that repeatedly divides the interval into two parts grows logarithmically with respect to N. This height is important because update and query operations in a segment tree only need to visit a limited number of nodes on each level.

C. Regular Segment Tree

A segment tree is a binary tree data structure used to store information about intervals of an array. The root represents the entire array, while its children represent smaller subintervals. This division continues recursively until each leaf node represents one array element.

Each internal node stores a value obtained by combining the values of its children. In a range sum segment tree, this value is the sum of all elements in the node's interval. Other operations, such as minimum or maximum, can also be used if two child intervals can be combined.

A regular segment tree supports build, update, and query operations. The build operation constructs the tree from the initial array, the update operation changes one element and recalculates affected nodes, and the query operation retrieves the combined value of a given range. However, a regular segment tree only stores the newest state, so previous states are overwritten after updates.

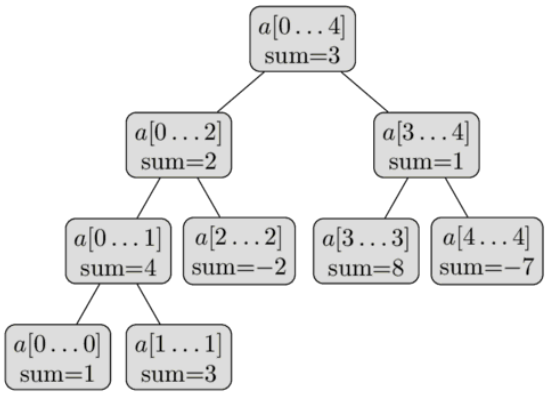

Fig. 2.1 Segment Tree Illustration
(Taken from cp-algorithms)

D. Persistent Data Structure

A persistent data structure is a data structure that preserves its previous versions after modifications. Instead of replacing the old structure completely, a persistent data structure creates a new version while keeping the old version accessible. This is useful when a program needs to answer queries not only on the current data, but also on earlier states of the data.

There are several ways to implement persistence. One common method is path copying. In this method, when a node must be changed, the algorithm creates a new copy of that node instead of modifying the original node. Then, all ancestors of that node must also be copied so that the new version has its own path from the root. Parts of the structure that are not affected by the update can still be shared between the old and new versions.

E. Persistent Segment Tree

A persistent segment tree applies the idea of persistence to a segment tree. Each version of the array is represented by a different root node. The first root represents the initial array, while every update creates a new root that represents the updated version. Since each point update only affects one path from the root to a leaf, only the nodes on that path need to be copied.

The unaffected subtrees are shared between versions. Therefore, the persistent segment tree avoids copying the entire tree after each update. This makes it possible to keep many historical versions while still maintaining efficient update and query operations.

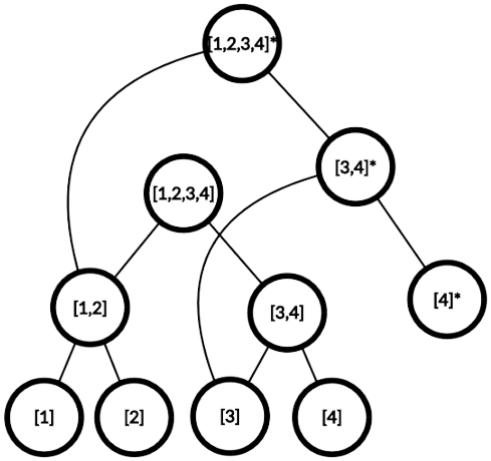

Fig. 2.2 Persistent Segment Tree Illustration
(Taken from Stack Overflow)

III. SEGMENT TREE AND PERSISTENT SEGMENT TREE MECHANISM

A. Regular Segment Tree Mechanism

A regular segment tree is built recursively by dividing an array interval into two smaller intervals until each leaf represents one array element. For a range sum segment tree, each internal node stores the sum of its two children. This allows range queries to use stored interval values directly instead of scanning every element.

Point updates are also performed recursively. The algorithm follows the path from the root to the updated leaf, changes the leaf value, and recalculates the affected ancestors. However, these changes are applied directly to the existing tree nodes, so the previous version of the tree is overwritten.

B. Limitation of Regular Segment Tree

The main limitation of a regular segment tree is that it only stores the latest version of the array. After an update, the old values of the updated leaf and its ancestors are lost. Copying the entire tree before every update could preserve old versions, but this would require copying many nodes that are not actually affected by the update.

C. Persistent Segment Tree Mechanism

A persistent segment tree avoids this problem by using path copying. Instead of modifying old nodes, the update operation creates new nodes only along the affected root-to-leaf path. Unchanged subtrees are reused between versions.

Each version is represented by a root pointer. The initial root represents version 0, and every update returns a new root for the new version. Since the old root is still stored, old versions can still be queried.

D. *Build, Update, and Query Operations*

The build operation creates the initial tree. The update operation receives a root, an index, and a new value, then returns a new root after copying only the affected path. The query operation receives the root of the chosen version and performs the same range query logic as a regular segment tree. Since queries do not modify the tree, they do not create new nodes.

IV. REGULAR SEGMENT TREE IMPLEMENTATION

This section describes the regular segment tree implementation used as the comparison basis for the persistent segment tree. The implementation supports point updates and range sum queries. Unlike the persistent version, a regular segment tree modifies its nodes directly, so only the latest version of the array is stored.

A. *Tree Representation*

The regular segment tree is represented by an array, and the root of the tree is placed at index 0. If a node is stored at index cur, then its left child is stored at index 2cur + 1 and its right child is stored at index 2cur + 2. This indexing method allows the tree to be represented without using pointers.

```
const int N = 1e5;
int A[N + 5];
int TREE[4 * N + 5];
```

Fig. 4.1 Tree Representation Implemetation of Regular Segment Tree

The array A stores the initial input values, while TREE stores the values of the segment tree nodes. Since the implementation is used for range sum queries, each node in TREE stores the sum of the array interval represented by that node.

B. *Build Operation*

The build operation constructs the segment tree recursively. If the current interval contains only one element, the current node stores the value of that element. Otherwise, the interval is divided into two subintervals. The left and right children are built first, then the current node stores the sum of both children.

```
void build(int cur, int l, int r) {
    if (l == r) {
        TREE[cur] = A[l];
        return;
    }
    int mid = (l + r) / 2;
    build(cur * 2 + 1, l, mid);
    build(cur * 2 + 2, mid + 1, r);
    TREE[cur] = TREE[cur * 2 + 1] + TREE[cur * 2 + 2];
}
```

Fig. 4.2 Build Implementation of Regular Segment Tree

In this function, cur represents the current node index in the TREE array, while l and r represent the left and right boundaries of the interval covered by that node. The initial call is build(0, 0, N - 1), because the root is stored at index 0 and represents the whole array.

C. *Update Operation*

The update operation changes the value of one array element. The algorithm recursively searches for the leaf node that represents the updated index. After the leaf value is changed, every ancestor on the path back to the root is recalculated.

```
void update(int cur, int l, int r, int idx, int val) {
    if (l > r || r < idx || l > idx) return;
    if (l == r) {
        TREE[cur] = val;
        return;
    }
    int mid = (l + r) / 2;
    update(cur * 2 + 1, l, mid, idx, val);
    update(cur * 2 + 2, mid + 1, r, idx, val);
    TREE[cur] = TREE[cur * 2 + 1] + TREE[cur * 2 + 2];
}
```

Fig. 4.3 Update Implementation of Regular Segment Tree

The initial call is update(0, 0, N - 1, idx, val), where idx is the index to change and val is the new value. The condition at the beginning of the function checks whether the current interval contains the updated index. If the interval does not contain the index, the function stops immediately. If the interval is a leaf, the value is updated. Then, on the way back from recursion, the affected internal nodes are recalculated.

D. *Query Operation*

The query operation returns the sum of elements inside a given range. If the current interval is completely outside the query range, the function returns 0 because 0 is the identity value for addition. If the current interval is completely inside the query range, the stored value of the current node is returned directly. If the current interval partially overlaps with the query range, the query continues recursively to both children.

```
int query(int cur, int l, int r, int ql, int qr) {
    if (l > r || l > qr || r < ql) return 0;
    if (ql <= l && r <= qr) return TREE[cur];
    int mid = (l + r) / 2;
    return query(cur * 2 + 1, l, mid, ql, qr)
         + query(cur * 2 + 2, mid + 1, r, ql, qr);
}
```

Fig. 4.4 Query Implementation of Regular Segment Tree

The initial call is query(0, 0, N - 1, ql, qr), where ql is the left bound and qr is the right bound of the range query. The query operation does not modify the tree. It only reads values from nodes that are relevant to the requested range.

V. REGULAR SEGMENT TREE COMPLEXITY ANALYSIS

Let N be the data size, U be the number of update operations, and Q be the number of query operations. This section analyzes the time and space complexity of the regular segment tree implementation. The analysis is based on the number of nodes visited, updated, or stored by the algorithm.

A. Build Operation Complexity

The build operation constructs the segment tree from the initial array. Each element of the array becomes one leaf node, so there are N leaf nodes. Since the segment tree is a full binary tree, the number of internal nodes is N - 1. Therefore, the total number of logical nodes in the segment tree is

$$T_{build}(N) = N + (N - 1)$$

$$T_{build}(N) = 2N - 1$$

Because the build operation visits and computes each node exactly once, the number of operations is proportional to the number of nodes.

To prove the upper bound, for every N ≥ 1,

$$T_{build}(N) = 2N - 1 \leq 2N$$

Let C = 2. Then,

$$T_{build}(N) \leq C\,N$$

Therefore,

$$T_{build}(N) \in O(N)$$

To prove the lower bound, for every N ≥ 1,

$$T_{build}(N) = 2N - 1 \geq N$$

Let c = 1. Then,

$$T_{build}(N) \geq c\,N$$

Therefore,

$$T_{build}(N) \in \Omega(N)$$

Because its upper bound is the same as its lower bound, then,

$$T_{build}(N) \in \Theta(N)$$

In the implementation, the array TREE is allocated with size 4N to safely store all possible node indices. Therefore,

$$S_{build}(N) \leq 4N$$

So

$$S_{build}(N) \in O(N)$$

Since at least N leaf values must be represented,

$$S_{build}(N) \geq N$$

So

$$S_{build}(N) \in \Omega(N)$$

Therefore,

$$S_{build}(N) \in \Theta(N)$$

B. Update Operation Complexity

The update operation changes one element in the array. The function starts from the root and recursively searches for the leaf node that represents the updated index. In the implementation, both children are called, but the child whose interval does not contain the updated index returns immediately.

Let d be the depth of the updated leaf. Along the root-to-leaf path, the algorithm visits d + 1 relevant nodes. At each internal level, the algorithm also calls one sibling subtree that immediately returns. Therefore, the number of function calls can be written as

$$T_{update}(N) = 2d + 1$$

Since the segment tree divides the interval into two smaller intervals at each level, the depth d is bounded by

$$d \leq ceil(\log_2 N)$$

Thus,

$$T_{update}(N) \leq 2ceil(\log_2 N) + 1$$

For N ≥ 2,

$$ceil(\log_2 N) \leq \log_2 N + 1$$

so

$$T_{update}(N) \leq 2(\log_2 N + 1) + 1$$

$$T_{update}(N) \leq 2\log_2 N + 3$$

Since $\log_2 N \geq 1$ for N ≥ 2,

$$2\log_2 N + 3 \leq 5\log_2 N$$

Therefore,

$$T_{update}(N) \leq 5\log_2 N$$

So

$$T_{update}(N) \in O(\log N)$$

For the lower bound in the worst case, the update operation must reach the leaf representing the updated index. Since the height of the segment tree grows logarithmically with respect to N, there exists a constant c > 0 such that

$$T_{update}(N) \geq c\log_2 N$$

Therefore,

$$T_{update}(N) \in \Omega(\log N)$$

$$T_{update}(N) \in \Theta(\log N)$$

The regular update operation does not allocate new segment tree nodes. It only modifies values already stored in the TREE array. Therefore, the permanent additional space used by one update is

$$S_{update}(N) \in \Theta(1)$$

However, because the implementation is recursive, the recursion stack follows the height of the segment tree. Thus, the auxiliary stack space is

$$S_{update-stack}(N) \in \Theta(\log N)$$

C. Query Operation Complexity

The query operation returns the sum of elements inside a requested range. The function starts from the root and checks whether the current interval is outside, inside, or partially overlapping with the query range.

If the current interval is completely outside the query range, the function returns 0. If the current interval is completely inside the query range, the stored value is returned directly. If the interval partially overlaps, the function continues recursively to its children.

In the worst case, a range query visits nodes around the left and right boundaries of the query interval. At each level, only a constant number of nodes can partially overlap with the query boundary. Since the height of the tree is at most ceil($\log_2$ N), the number of visited nodes is bounded by

$$T_{query}(N) \leq 4ceil(\log_2 N) + 1$$

For N ≥ 2,

$$ceil(\log_2 N) \leq \log_2 N + 1$$

so

$$T_{query}(N) \leq 4(\log_2 N + 1) + 1$$
$$T_{query}(N) \leq 4\log_2 N + 5$$

Since $\log_2$ N ≥ 1 for N ≥ 2,

$$4\log_2 N + 5 \leq 9\log_2 N$$

Therefore,

$$T_{query}(N) \leq 9\log_2 N$$

So

$$T_{query}(N) \in O(\log N)$$

For the lower bound in the worst case, consider a query that asks for only one index. The algorithm must follow a path from the root to the leaf representing that index. This path has logarithmic height. Therefore, there exists a constant c > 0 such that

$$T_{query}(N) \geq c \log_2 N$$

So

$$T_{query}(N) \in \Omega(\log N)$$

then the worst-case query complexity is

$$T_{query}(N) \in \Theta(\log N)$$

However, not every query takes logarithmic time. If the query range is the entire array, the answer is stored directly at the root, so the query can be answered in constant time:

$$T_{query-best}(N) \in \Theta(1)$$

The query operation does not allocate new segment tree nodes. Therefore, the permanent additional space used by one query is

$$S_{query}(N) \in \Theta(1)$$

Since the implementation is recursive, the auxiliary stack space in the worst case is

$$S_{query-stack}(N) \in \Theta(\log N)$$

D. Total Complexity for Multiple Operations

If the program performs U update operations and Q query operations after the initial build, the total running time is

$$T_{total(N,U,Q)} = T_{build(N)} + U\,T_{update(N)} + Q\,T_{query(N)}$$

From the previous analysis,

$$T_{build}(N) \in \Theta(N)$$
$$T_{update}(N) \in \Theta(\log N)$$
$$T_{query}(N) \in \Theta(\log N)$$

Therefore,

$$T_{total}(N,U,Q) \in \Theta(N + U\log N + Q\log N)$$

This can be simplified as

$$T_{total}(N,U,Q) \in \Theta(N + (U + Q)\log N)$$

The total permanent space of the regular segment tree remains

$$S_{total}(N,U,Q) \in \Theta(N)$$

This is because all update operations modify the same TREE array, and query operations do not create new nodes. Therefore, even after U updates and Q queries, the regular segment tree only stores the latest version of the array.

VI. MAKING THE SEGMENT TREE PERSISTENT

The regular segment tree implementation is efficient for update and query operations, but it only stores the latest state of the array. When an update is performed, the affected leaf and its ancestors are modified directly inside the same TREE array. This means that the previous version is destroyed. Therefore, the main problem is not the update time itself, but the loss of historical information after an update.

A. The Bottleneck of Preserving Versions

Suppose a program needs to keep every version of the segment tree after each update. A simple approach is to copy the entire segment tree before applying an update. Since a segment tree contains Θ(N) nodes, copying the whole tree for every update would require

$$S_{update-copy}(N) \in \Theta(N)$$

additional space per update. If there are U update operations, then the total space required to store all versions becomes

$$S_{total-copy}(N,U) \in \Theta(N + UN).$$

This approach preserves all versions, but it is inefficient because a point update does not actually change the whole tree.

In a segment tree, changing one array element only affects the leaf representing that element and the ancestors of that leaf. All other subtrees remain unchanged. Therefore, copying the entire tree wastes memory by duplicating many nodes whose values do not change.

B. The Key Observation

The key observation is that a point update only affects one path from the root to a leaf. If the updated index is idx, then only the intervals containing idx need to be changed. These intervals appear exactly along the root-to-leaf path. Subtrees whose intervals do not contain idx are not affected by the update.

Because the height of a segment tree is logarithmic with respect to N, the number of affected nodes is only proportional to log N. This means that, instead of copying Θ(N) nodes, an update only needs to copy Θ(log N) nodes.

C. Path Copying

The technique used to make the segment tree persistent is called path copying. Instead of modifying old nodes, the update operation creates new copies of the nodes along the affected root-to-leaf path. The unaffected children are reused by pointing to the same nodes as the previous version.

For example, when the update goes to the left child, the new parent node will contain a new left child and reuse the old right child. Similarly, when the update goes to the right child, the new parent node will reuse the old left child and contain a new right child. This process continues until the updated leaf is reached.

Assume there is an array [2, 1, 5, 3] and a persistent segment tree is used to compute its range sum. The segment tree visualization can be seen in the figure below.

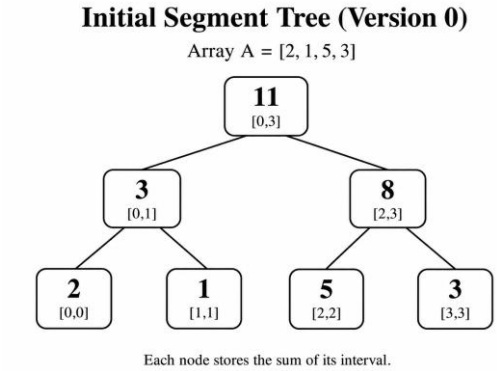

Fig. 6.1 Initial Persistent Segment Tree Illustration

Suppose there is an update that changes the value of index 2 in A from 5 to 7, in other words, A[2] = 7. A new tree will be created, but only includes the affected root-to-leaf path. The rest of the nodes will be filled by pointing to nodes in the original segment tree. The visualization of the segment tree can be seen in the following figure.

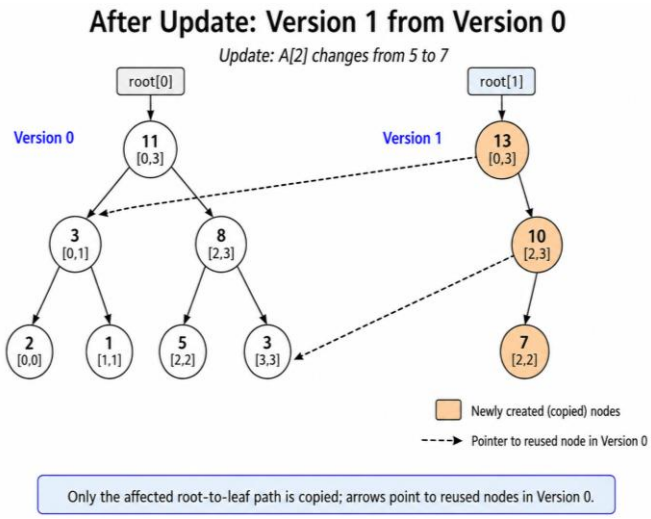

Fig. 6.2 Updated Persistent Segment Tree Illustration

As a result, the old version remains unchanged, while the new version has its own path to the updated leaf. Both versions can still share all unchanged subtrees.

D. Version Roots

A persistent segment tree cannot be represented conveniently using a single array like the regular implementation, because different versions may share some nodes but have different root-to-leaf paths. Therefore, the persistent implementation usually uses pointers or node indices.

Each version is represented by one root pointer. The first root points to the segment tree built from the initial array. Every update returns a new root pointer, which represents the new version. The old root pointer is kept, so the old version can still be queried.

If root[0] represents the initial version, then root[1] can represent the version after the first update, root[2] can represent the version after the second update, and so on. Querying a version only requires choosing the appropriate root before running the query operation.

E. Effect of Persistence

Persistence changes the memory behavior of the segment tree, but it does not significantly change the logic of range queries. The query operation still works by checking whether the current interval is outside, inside, or partially overlapping with the requested range. The only difference is that the query starts from the root of the selected version.

The update operation is the main difference. In a regular segment tree, the update modifies existing nodes. In a persistent segment tree, the update creates new nodes along the affected path and shares the unchanged subtrees.

VII. PERSISTENT SEGMENT TREE IMPLEMENTATION

This section describes the persistent segment tree implementation used in this paper. Unlike the regular segment tree implementation, which stores all nodes inside one array, the persistent segment tree uses dynamically allocated nodes and

pointers. This is necessary because several versions may share some nodes while also having different copied paths.

The implementation supports point assignment updates and range sum queries. Each version is represented by a root pointer. The initial root represents the segment tree built from the original array, while each update produces a new root pointer for the new version.

A. *Node Structure*

Each node stores the sum of its interval and two pointers to its children. The interval boundaries are not stored inside the node. Instead, they are passed as parameters in recursive functions.

```
struct Node {
    int sum;
    Node *l;
    Node *r;
    Node(int _sum) {
        sum = _sum;
        l = r = nullptr;
    }
    Node(Node *L, Node *R) {
        l = L; r = R;
        sum = 0;
        if (l != nullptr) sum += (l->sum);
        if (r != nullptr) sum += (r->sum);
    }
};
```

Fig. 7.1 Node Structure of Persistent Segment Tree

The first constructor is used to create a leaf node. The second constructor is used to create an internal node from two child nodes. For a range sum segment tree, the value of an internal node is the sum of the values stored in its left and right children.

B. *Build Operation*

The build operation constructs the first version of the segment tree. This operation is similar to the build operation in the regular segment tree, because there is no older version that needs to be preserved during the initial construction.

```
Node* build(int l, int r) {
    if (l == r) return new Node(A[l]);
    int mid = (l + r) / 2;
    return new Node(build(l, mid), build(mid + 1, r));
}
```

Fig. 7.2 Build Implementation of Persistent Segment Tree

If the current interval contains only one element, a leaf node is created. Otherwise, the interval is divided into two subintervals, and the current node is created from the two child nodes. The root returned by this function becomes the root of version 0.

C. *Update Operation*

The update operation is the main difference between the regular and persistent implementations. In the regular segment tree, an update modifies existing nodes directly. In the persistent segment tree, an update creates new nodes along the path from the root to the updated leaf.

```
Node* update(Node *cur, int l, int r, int idx, int val) {
    if (idx < l || idx > r) return cur;
    if (idx == l && idx == r) return new Node(val);
    int mid = (l + r) / 2;
    return new Node(
        update(cur->l, l, mid, idx, val),
        update(cur->r, mid + 1, r, idx, val)
    );
}
```

Fig. 7.3 Update Implementation of Persistent Segment Tree

If the current interval does not contain the updated index, the function returns the existing node. This means that the subtree is reused by the new version. If the current interval is a leaf, a new leaf node is created with the updated value. Otherwise, the function recursively updates the appropriate path and creates a new internal node.

Although the function calls both children, only the child whose interval contains the updated index creates new nodes. The other child immediately returns the old node. Therefore, the new version shares all unaffected subtrees with the previous version.

D. *Query Operation*

The query operation retrieves the sum of a given range from a selected version. The algorithm is almost the same as the query operation in a regular segment tree. The only difference is that the function starts from the root pointer of the version being queried.

```
int query(Node *cur, int l, int r, int ql, int qr) {
    if (l > r || r < ql || l > qr) return 0;
    if (ql <= l && r <= qr) return cur->sum;
    int mid = (l + r) / 2;
    return query(cur->l, l, mid, ql, qr)
        + query(cur->r, mid + 1, r, ql, qr);
}
```

Fig. 7.4 Query Implementation of Persistent Segment Tree

If the current interval is outside the query range, the function returns 0. If the current interval is completely inside the query range, the stored sum is returned directly. If the interval partially overlaps the query range, the query continues recursively to both children.

The query operation does not create new nodes. Therefore, querying an old version does not change any version of the data structure.

E. *Version Roots*

All versions are stored using root pointers. The first root is created by the build operation. Each update takes one existing root and returns a new root.

```
vector<Node*> roots;
roots.push_back(build(0, N - 1));
roots.push_back(update(roots.back(), 0, N - 1, idx, val));
```

Fig. 7.5 Version Control of Persistent Segment Tree

In this example, roots[0] represents the initial version of the array. After one update, roots[1] represents the updated version. Since roots[0] is not modified, queries can still be performed on the original version.

For example, the following two queries may return different results because they are performed on different versions.

```
query(roots[0], 0, N - 1, ql, qr);
query(roots[1], 0, N - 1, ql, qr);
```

Fig. 7.6 Query Example of Persistent Segment Tree

This version-root mechanism is the main reason the persistent segment tree can preserve historical states. Each root represents one version, while unchanged nodes may be shared between multiple versions.

VIII. PERSISTENT SEGMENT TREE COMPLEXITY ANALYSIS

Let N be the data size, U be the number of update operations, and Q be the number of query operations. This section analyzes the time and space complexity of the persistent segment tree implementation. The main difference between the regular and persistent segment tree is in the update operation. A regular segment tree overwrites old nodes, while a persistent segment tree creates new nodes along the affected path and reuses unchanged subtrees.

A. Build Operation Complexity

The build operation constructs the initial version of the persistent segment tree. This operation is similar to the build operation in a regular segment tree. Each array element becomes one leaf node. Since the tree is a full binary tree, the number of internal nodes is N - 1. Therefore, the total number of nodes created during the build operation is

$$T_{build}(N) = N + (N - 1)$$

$$T_{build}(N) = 2N - 1$$

Because every node is created exactly once, the build time is proportional to the number of created nodes.

For the upper bound, for every N ≥ 1,

$$T_{build}(N) = 2N - 1 \leq 2N$$

Let C = 2. Then,

$$T_{build}(N) \leq C\,N$$

Therefore,

$$T_{build}(N) \in O(N)$$

For the lower bound, for every N ≥ 1,

$$T_{build}(N) = 2N - 1 \geq N$$

Let c = 1. Then,

$$T_{build}(N) \geq c\,N$$

Therefore,

$$T_{build}(N) \in \Omega(N)$$

so

$$T_{build}(N) \in \Theta(N)$$

The build operation also allocates 2N - 1 nodes in memory. Each node stores a sum value and two child pointers, which are constant-sized information. Therefore,

$$S_{build}(N) = 2N - 1$$

Using the same upper and lower bound argument,

$$S_{build}(N) \in \Theta(N)$$

B. Update Operation Complexity

The update operation creates a new version of the segment tree. It does not modify the old version. Instead, it creates new nodes along the path from the root to the updated leaf.

Let d be the depth of the updated leaf. The update operation creates one new node for each level on the path from the root to that leaf. This includes the new leaf itself and all copied ancestors. Therefore, the number of newly allocated nodes is

$$S_{update}(N) = d + 1$$

Since the height of the segment tree is bounded by ceil($\log_2$ N),

$$d \leq ceil(\log_2 N)$$

Thus,

$$S_{update}(N) \leq ceil(\log_2 N) + 1$$

For N ≥ 2,

$$ceil(\log_2 N) \leq \log_2 N + 1$$

so

$$S_{update}(N) \leq \log_2 N + 2$$

Since $\log_2$ N ≥ 1 for N ≥ 2,

$$\log_2 N + 2 \leq 3\log_2 N$$

Therefore,

$$S_{update}(N) \leq 3\log_2 N$$

Let C = 3. Then,

$$S_{update}(N) \leq C \log_2 N$$

So

$$S_{update}(N) \in O(\log N)$$

For the lower bound in the worst case, the update must create a new path from the root to the updated leaf. Since the depth of a leaf in a balanced segment tree grows logarithmically with respect to N, there exists a constant c > 0 such that

$$S_{update}(N) \geq c \log_2 N$$

Therefore,

$$S_{update}(N) \in \Omega(\log N)$$

Then

$$S_{update}(N) \in \Theta(\log N)$$

The time complexity follows the same path structure. In the implementation, the update function may call both children, but the child whose interval does not contain the updated index returns immediately. The algorithm only continues deeply along one root-to-leaf path. Therefore, the number of recursive calls is proportional to the height of the tree. Thus,

$$T_{update}(N) \in \Theta(\log N)$$

C. Query Operation Complexity

The query operation retrieves a range sum from a selected version. It starts from the root pointer of that version. The query algorithm is the same as the regular segment tree query algorithm because the structure of each version is still a segment tree.

If the current interval is completely outside the query range, the function returns 0. If the current interval is completely inside the query range, the function returns the stored sum directly. If the interval partially overlaps, the function continues recursively to both children.

In the worst case, a query visits nodes around the boundary of the query interval. At each level, only a constant number of nodes can partially overlap with the query boundary. Since the height of the segment tree is bounded by ceil($\log_2$ N), the number of visited nodes is bounded by

$$T_{query}(N) \leq 4ceil(\log_2 N) + 1$$

For N ≥ 2,

$$ceil(\log_2 N) \leq \log_2 N + 1$$

so

$$T_{query}(N) \leq 4(\log_2 N + 1) + 1$$

$$T_{query}(N) \leq 4\log_2 N + 5$$

Since $\log_2$ N ≥ 1 for N ≥ 2,

$$4\log_2 N + 5 \leq 9\log_2 N$$

Let C = 9. Then,

$$T_{query}(N) \leq C \log_2 N$$

Therefore,

$$T_{query}(N) \in O(\log N)$$

For the lower bound in the worst case, consider a query for a single index. The query must follow a path from the root to one leaf. Since the height of the tree grows logarithmically with respect to N, there exists a constant c > 0 such that

$$T_{query}(N) \geq c \log_2 N$$

Therefore,

$$T_{query}(N) \in \Omega(\log N)$$

then the worst-case query time complexity is

$$T_{query}(N) \in \Theta(\log N)$$

However, not every query takes logarithmic time. A query that asks for the whole array can be answered directly from the selected root node. Therefore, the best-case query time is

$$T_{query-best}(N) \in \Theta(1)$$

The query operation does not allocate new nodes. Therefore, the permanent additional space used by one query is

$$S_{query}(N) \in \Theta(1)$$

However, because the implementation is recursive, the auxiliary stack space is proportional to the height of the tree. Thus,

$$S_{query-stack}(N) \in \Theta(\log N)$$

D. Total Space After Multiple Updates

The main advantage of the persistent segment tree is that it avoids copying the whole tree for every update. The initial build creates 2N - 1 nodes. Each update creates only the copied path from the root to the updated leaf. Let d_i be the depth of the updated leaf in the i-th update. Then the total number of nodes after U updates is

$$S_{total}(N,U) = (2N - 1) + \sum_{i=1}^{U}(d_i + 1)$$

Since each d_i is bounded by ceil($\log_2$ N),

$$S_{total}(N,U) \leq (2N - 1) + U(ceil(\log_2 N) + 1)$$

For N ≥ 2,

$$ceil(\log_2 N) + 1 \leq 3\log_2 N$$

So

$$S_{total}(N,U) \leq 2N + 3Ulog_2 N$$

Therefore,

$$S_{total}(N,U) \in O(N + U \log N)$$

For the lower bound, the initial build already requires N leaf nodes, so the structure needs at least Ω(N) space. In addition, each update must create a new path with logarithmic length in the worst case. Therefore,

$$S_{total}(N,U) \in \Omega(N + U \log N)$$

Since the upper and lower bounds are equal asymptotically,

$$S_{total}(N,U) \in \Theta(N + U \log N)$$

The vector of root pointers also stores one root for each version. Since there are U updates after the initial version, the number of roots is U + 1. Therefore, the root pointer storage is Θ(U). For N ≥ 2, this is dominated by Θ(U log N), so the total space remains

$$S_{total}(N,U) \in \Theta(N + U \log N)$$

E. Total Time for Multiple Operations

If the program performs U update operations and Q query operations after the initial build, then the total running time is

$$T_{total}(N,U,Q) = T_{build}(N) + U\,T_{update}(N) + Q\,T_{query}(N)$$

From the previous analysis,

$$T_{build}(N) \in \Theta(N)$$

$$T_{update}(N) \in \Theta(\log N)$$

$$T_{query}(N) \in \Theta(\log N)$$

Therefore,

$$T_{total}(N,U,Q) \in \Theta(N + (U + Q)\log N)$$

F. Comparison with the Regular Segment Tree

The persistent segment tree has the same asymptotic time complexity as the regular segment tree for build, update, and query operations. The difference is in memory usage and version access. The regular segment tree uses Θ(N) permanent space because it only stores one version. The persistent segment tree uses Θ(N + U log N) permanent space because each update creates a new version by allocating Θ(log N) new nodes.

Operation	Regular Segment Tree	Persistent Segment Tree
Build time	Θ(N)	Θ(N)
Build space	Θ(N)	Θ(N)
Point update time	Θ(log N)	Θ(log N)
Space per update	Θ(1)	Θ(log N)
Range query time	Θ(log N)	Θ(log N)
Total time	Θ(N + (U+Q)log N)	Θ(N + (U+Q)log N)
Total Space	Θ(N)	Θ(N + U log N)
Version Control	No	Yes

Table 8.1 Regular and Persistent Segment Tree Comparison

IX. CONCLUSION

Persistent segment tree extends the regular segment tree by preserving previous versions after updates. While a regular segment tree overwrites affected nodes and only keeps the latest array state, a persistent segment tree uses path copying to create a new version without destroying the old one.

The analysis shows that both regular and persistent segment trees require Θ(N) time and Θ(N) space for the build operation. Both structures also support point updates and range queries in Θ(log N) time in the worst case. The main difference is memory usage: a regular segment tree uses Θ(N) permanent space, while a persistent segment tree uses Θ(N + U log N) permanent space after U updates.

Therefore, persistent segment tree is useful when historical versions of an array must remain queryable. Its main tradeoff is additional memory usage, but this cost is much smaller than copying the entire segment tree after every update.

VIDEO LINK AT YOUTUBE

https://youtu.be/6wW8vqC1zj4

ACKNOWLEDGMENT

The author would like to express gratitude to God Almighty for the opportunity and ability to complete this paper. The author also expresses appreciation to the lecturer of IF1220 Discrete Mathematics, Prof. Dr. Ir. Rinaldi, M.T. for providing the theoretical foundations of algorithm complexity and tree structures, which support the analysis discussed in this paper. Finally, the author would like to thank family, friends, and all parties who provided support during the writing process.

REFERENCES

[1] Munir, Rinaldi. 2025. "Kompleksitas Algoritma (Bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/25-Kompleksitas-Algoritma-Bagian1-2026.pdf. Accessed: Jun. 15, 2026.

[2] Munir, Rinaldi. 2025. "Kompleksitas algoritma (Bagian 2)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/26-Kompleksitas-Algoritma-Bagian2-2026.pdf. Accessed: Jun. 15, 2026.

[3] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," Journal of Computer and System Sciences, vol. 38, no. 1, pp. 86-124, Feb. 1989. Accessed: Jun. 15, 2026.

[4] D. R. Karger, "Persistent data structures," 6.854J Advanced Algorithms, Massachusetts Institute of Technology OpenCourseWare, Fall 2005. https://ocw.mit.edu/courses/6-854j-advanced-algorithms-fall-2005/resources/persistent/. Accessed: Jun. 15, 2026.

[5] Algorithms for Competitive Programming, "Segment tree," May 24, 2026. https://cp-algorithms.com/data_structures/segment_tree.html. Accessed: Jun. 15, 2026.

[6] E. Lehman, F. T. Leighton, and A. R. Meyer, Mathematics for Computer Science. Massachusetts Institute of Technology OpenCourseWare, 2015. https://ocw.mit.edu/courses/6-042j-mathematics-for-computer-science-spring-2015/mit6_042js15_textbook.pdf. Accessed: Jun. 15, 2026.

[7] E. D. Demaine and C. E. Leiserson, "Lecture 2: Asymptotic notation; recurrences; substitution, master method," 6.046J Introduction to Algorithms, Massachusetts Institute of Technology OpenCourseWare, Fall 2005. https://ocw.mit.edu/courses/6-046j-introduction-to-algorithms-sma-5503-fall-2005/resources/lecture-2-asymptotic-notation-recurrences-substitution-master-method/. Accessed: Jun. 15, 2026.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 18 Juni 2026

Jeremy Gerald Sutanto - 13525104