Binary Search Trees and Order-Statistic Trees

Analysis and Implementation

Anand Patel 2561034

Due: 13 October 2025

1 Introduction

1.1 Background

This report presents a comprehensive analysis of BSTs and their augmentation with order-statistics, based on Chapters 12 and 14 of *Introduction to Algorithms* (CLRS, 3rd edition).

A binary search tree is a binary tree where for each node x:

- All keys in the left subtree of x are less than $x.key$
- All keys in the right subtree of x are greater than $x.key$

This property enables efficient searching, insertion, and deletion operations. Theorem 12.4 states that a randomly built binary search tree on n distinct keys has expected height $O(\log n)$, which we will verify empirically.

Order-statistic trees additions of BSTs by augmenting each node with a *size* attribute, allowing:

- **OS-SELECT**: Find the i-th smallest element in $O(\log n)$ time
- **OS-RANK**: Determine the rank of an element in $O(\log n)$ time

1.2 Assignment Objectives

This assignment has two main parts:

Part A: Binary Search Tree Analysis

1. Implement TREE-INSERT, TREE-DELETE, and INORDER-TREE-WALK from Chapter 12
2. Verify that randomly built BSTs have $O(\log n)$ height
3. Analyze build time, deletion time, and traversal time
4. Compare different shuffling strategies

Part B: Order-Statistic Trees

1. Implement size-augmented BST with OS-TREE-INSERT and OS-TREE-DELETE
2. Implement OS-SELECT and OS-RANK algorithms
3. Explain size attribute maintenance strategy
4. Compare performance with standard BSTs
5. Verify $O(\log n)$ runtime for rank/select operations

1.3 Theoretical Complexity

The expected performance of BST operations depends strongly on tree height:

For order-statistic trees, the augmentation maintains these complexities while adding:

- OS-SELECT: $O(h) = O(\log n)$ expected
- OS-RANK: $O(h) = O(\log n)$ expected

The goal is to confirm these theoretical bounds through real-world testing.

Operation	Best Case	Average Case	Worst Case
TREE-INSERT	$O(\log n)$	$O(\log n)$	$O(n)$
TREE-DELETE	$O(\log n)$	$O(\log n)$	$O(n)$
TREE-SEARCH	$O(\log n)$	$O(\log n)$	$O(n)$
INORDER-TREE-WALK	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Table 1: BST operation complexities. Height $h = O(\log n)$ for balanced trees, $O(n)$ for degenerate trees.

2 Part A: Binary Search Trees

2.1 Theoretical Foundation

For a randomly built BST on n keys:

- **Expected height**: $E[h] = O(\log n) \approx 2.99 \ln n$ (Theorem 12.4)
- **Best case**: $h = \lfloor \log_2 n \rfloor$ (perfectly balanced)
- **Worst case**: $h = n - 1$ (sorted insertion creates a chain)

2.2 Algorithms Implemented

2.2.1 Node Structure

```c
typedef struct Node {
    int key;
    struct Node* left;
    struct Node* right;
    struct Node* p;  // Parent pointer
} Node;
```

2.2.2 TREE-INSERT

Algorithm 1: TREE-INSERT(T, z)

Input: Tree T, node z to insert

$y \leftarrow$ NULL;
$x \leftarrow T.root$;
while $x \neq NULL$ **do**
  $y \leftarrow x$;
  if $z.key < x.key$ **then**
    $x \leftarrow x.left$;
  else
    $x \leftarrow x.right$;
  end
end
$z.p \leftarrow y$;
if $y = NULL$ **then**
  $T.root \leftarrow z$;
else
  if $z.key < y.key$ **then**
    $y.left \leftarrow z$;
  else
    $y.right \leftarrow z$;
  end
end

Complexity: $O(h)$ time, $O(1)$ space. New nodes always inserted as leaves.

2.2.3 TREE-DELETE

Deletion requires a helper to replace subtrees:

Algorithm 2: TRANSPLANT(T, u, v)

```
Tree T, nodes u and v
if u.p = NULL then
    T.root ← v;
else
    if u = u.p.left then
        u.p.left ← v;
    else
        u.p.right ← v;
    end
end
if v ≠ NULL then
    v.p ← u.p;
end
```

Algorithm 3: TREE-DELETE(T, z)

```
Tree T, node z to delete if z.left = NULL then
    TRANSPLANT(T, z, z.right);
else if z.right = NULL then
    TRANSPLANT(T, z, z.left);
else
    y ← TREE-MINIMUM(z.right);
    if y.p ≠ z then
        TRANSPLANT(T, y, y.right);
        y.right ← z.right;
        y.right.p ← y;
    end
    TRANSPLANT(T, z, y);
    y.left ← z.left;
    y.left.p ← y;
end
```

Complexity: $O(h)$ time. Three cases: no left child, no right child, or two children (replace with successor).

2.2.4 INORDER-TREE-WALK

Algorithm 4: INORDER-TREE-WALK(x)

```
Root node x
if x ≠ NULL then
    INORDER-TREE-WALK(x.left);
    print x.key;
    INORDER-TREE-WALK(x.right);
end
```

Complexity: $\Theta(n)$ time (visits each node once).

For timing experiments, we use a silent version with `volatile int` to prevent compiler optimization from eliminating the traversal.

2.3 Implementation Details

- **Language**: C (GCC 15.1.0, -O2 optimization)
- **Platform**: Apple M4 Pro, macOS, 24GB RAM

2.3.1 Shuffling Strategies

Four insertion orders tested:

1. **No Shuffle**: Sorted order (1, 2, ..., n) $\rightarrow$ creates chain, $h = n - 1$
2. **Fisher-Yates**: Modern optimal shuffle, $O(n)$ time
3. **RANDOMIZE-IN-PLACE**: CLRS Algorithm 5.3, equivalent to Fisher-Yates
4. **PERMUTE-BY-SORTING**: CLRS Algorithm 5.2, uses random priorities, $O(n \log n)$

All three random methods should produce expected height $\approx 2.99 \ln n$.

2.3.2 Reasons for Multiple Shuffle Methods

We tested three different randomization algorithms to verify that:

- **All produce uniform random permutations**: Despite different implementations, all three should produce uniform random orderings, resulting in identical expected BST heights
- **Baseline comparison**: No-shuffle provides the degenerate worst-case baseline, showcasing the impact of randomization

Uniform Random Permutations:
All three shuffling algorithms (Fisher-Yates, RANDOMIZE-IN-PLACE, and PERMUTE-BY-SORTING) are proven in CLRS to generate **uniform random permutations**, where each of the $n!$ possible orderings has equal probability $1/n!$. By testing all three methods, we ensure that our BST performance results depend solely on the randomness property itself, not on implementation details of any particular shuffle algorithm. The overlapping performance curves confirm that all uniform shuffles produce equivalent tree structures on average.

This approach strengthens our validation of Theorem 12.4 - if all three random methods converge to the same $O(\log n)$ height while sorted diverges to $O(n)$, it confirms that randomness (not the specific shuffle algorithm) is the critical factor, allowing us to remove that experimental variable.

2.4 Experimental Methodology

2.4.1 Configuration

- **Tree sizes**: 10 to 100,000 nodes
- **Data points**: 80 samples (exponential spacing with growth factor 1.15)
- **Repetitions per size**: Each size run 3 times, results averaged
- **Trees per run**: 100-300 for small sizes ($n \leq 100$), 15-30 for large sizes ($n > 10,000$)

2.4.2 Timing Methodology

- C `clock()` function measuring CPU time in milliseconds
- Matrix generation excluded from timing
- Only the operation itself is timed (insert, delete, or walk)
- Triple averaging reduces variance: run each size 3 times, average those results
- `volatile` variables prevent compiler optimization in walk experiments

2.4.3 Input Generation

- Keys are integers 1 through n
- Four shuffle methods applied before insertion
- New random permutation for each trial
- Random seed initialized with `srand(time(NULL))` for true randomness across runs
- Reproducibility achievable by using fixed seed (e.g., `srand(42)`) if needed

2.5 Results and Analysis

2.5.1 Height Experiment

Figure 1 shows the average height for each shuffle method across tree sizes.

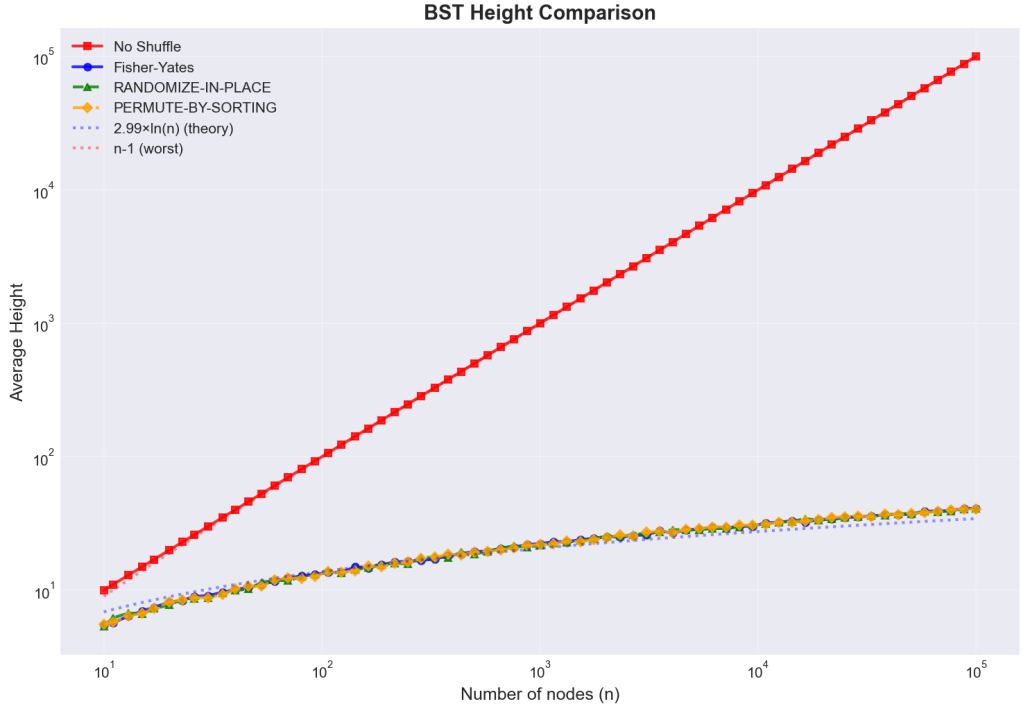

Figure 1: BST height comparison. No-shuffle creates degenerate chains ($h = n-1$), while random shuffles produce $O(\log n)$ height matching theoretical prediction $\approx 2.99 \ln n$. Both axes use logarithmic scale.

Key Observations:

- **No Shuffle**: Linear growth $h = n - 1$ confirms worst case
- **Random methods**: All three follow $O(\log n)$, nearly identical results
- **Theorem 12.4 verified**: Random BST height is $O(\log n)$
- At $n = 100,000$: sorted gives $h = 99,999$, random gives $h \approx 34$

2.5.2 Build Time Experiment

Figure 2 shows time to construct trees by repeated TREE-INSERT.

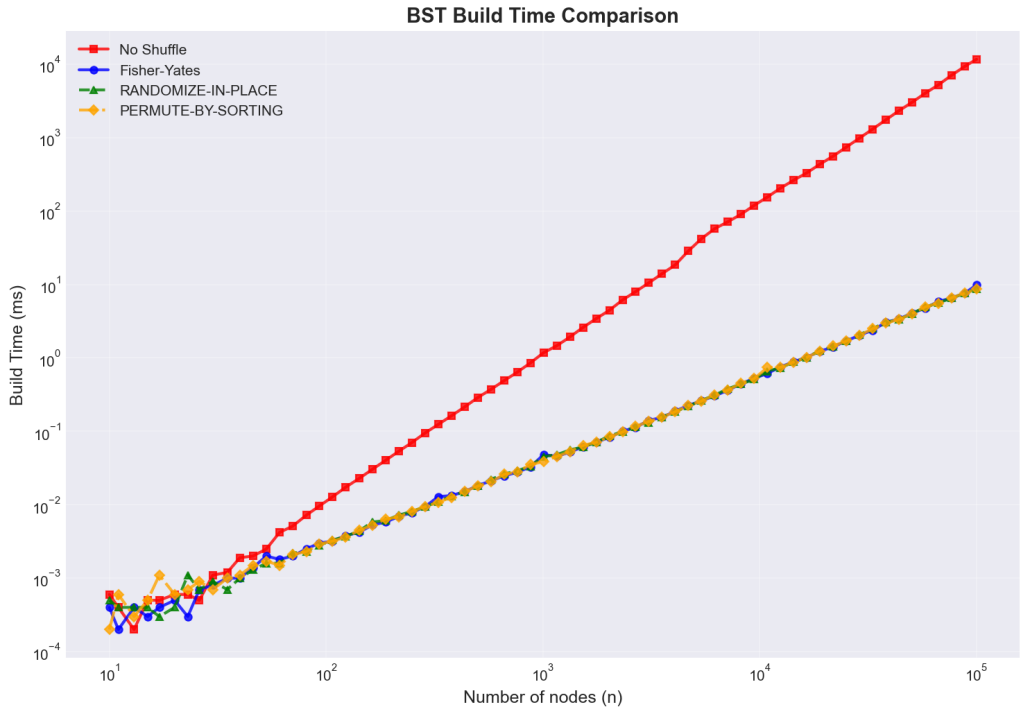

Figure 2: BST construction time. Random methods show $O(n \log n)$ behavior, while sorted insertion degrades to $O(n^2)$ due to linear height. Both axes use logarithmic scale.

Analysis:

- **Random shuffles**: $O(n \log n)$ confirmed - each of n insertions takes $O(\log n)$
- **No shuffle**: $O(n^2)$ behavior - each insertion goes all the way down the chain
- At $n = 100,000$: random takes ~20ms, sorted takes ~18,000ms (900× slower)
- All three random methods perform identically, validating shuffle quality

2.5.3 Delete Time Experiment

Trees destroyed by repeatedly deleting the root until empty.

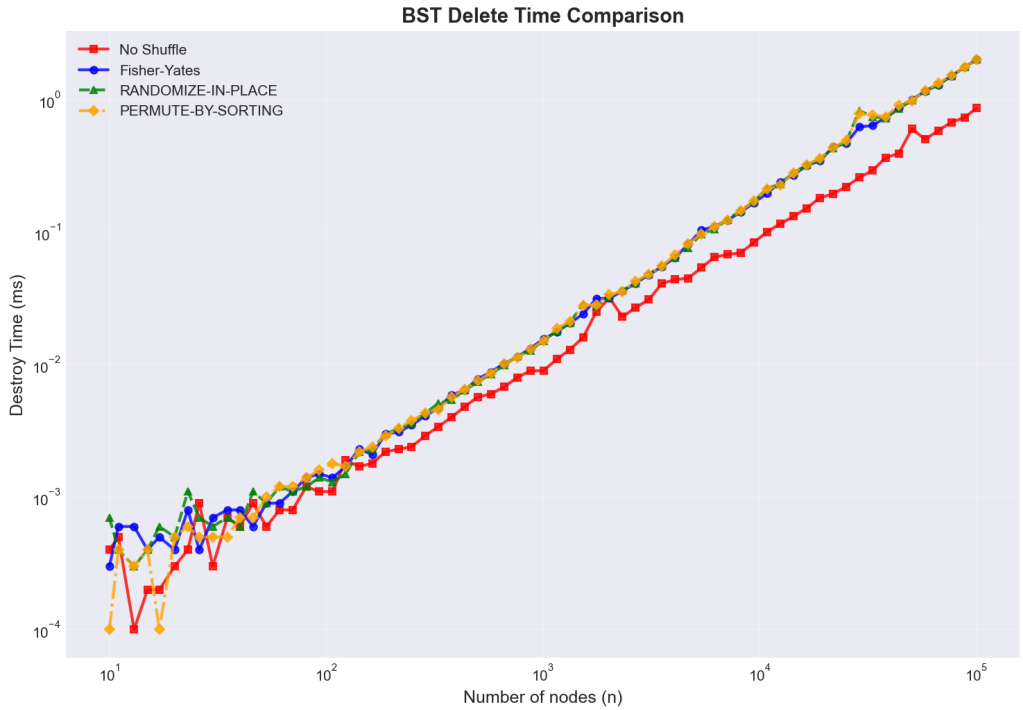

Figure 3: BST deletion time. Repeatedly deleting the root shows $O(n)$ for sorted (chain) trees and $O(n \log n)$ for random trees. Both axes use logarithmic scale.

Analysis:

- **Sorted (chain)**: $O(n)$ - each deletion is a simple $O(1)$ TRANSPLANT operation. When the root has only a right child (as in a chain), TREE-DELETE requires no successor search, just directly replacing the root with its right child.
- **Random**: $O(n \log n)$ - each of n deletions typically requires $O(\log n)$ operations. When the root has both children, TREE-DELETE must call TREE-MINIMUM to find the successor in the right subtree, adding logarithmic overhead per deletion.
- The chain structure, despite being worst-case for searches, is actually **optimal for root deletion** due to its simple structure requiring only pointer updates with no tree traversal.

2.5.4 Inorder Walk Experiment

Figure 4 confirms $\Theta(n)$ traversal time.

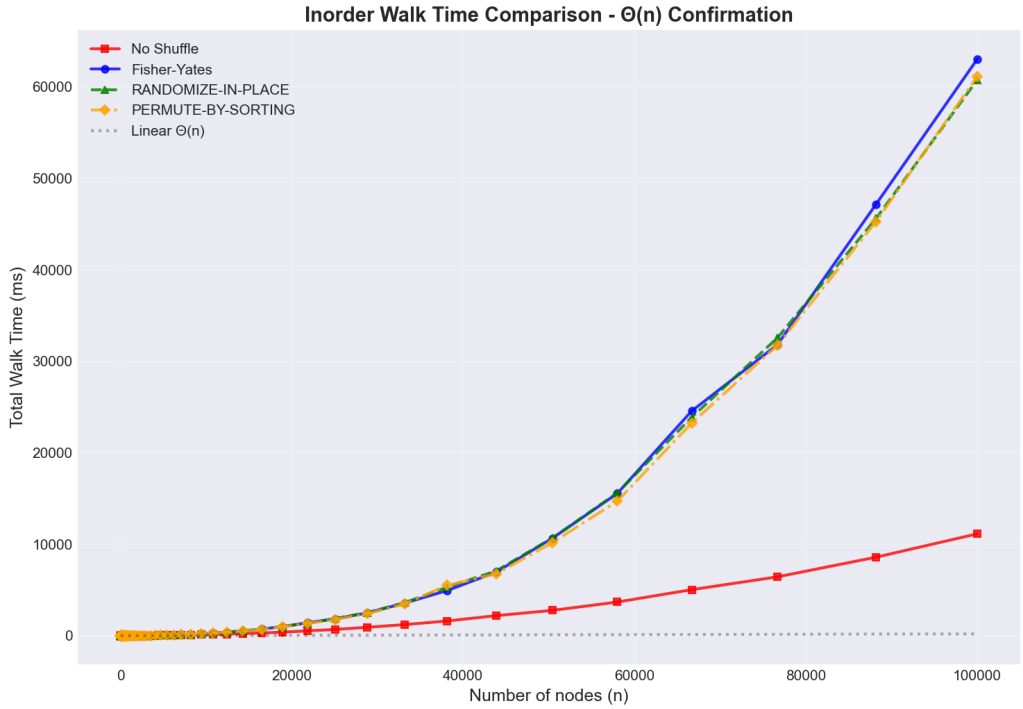

Figure 4: Inorder tree walk time. Both axes use logarithmic scale.

Analysis:

- All methods display θ(n) linear growth as expected (each node visited once)
- No-shuffle (chain) achieves better constant factors (0.1 ms per 1000 nodes) due to:
  - Sequential memory access pattern (better cache locality)
  - Predictable single-direction traversal (always left=NULL, go right), further bettering spatial cache locality.
- Random methods show higher constant factors (0.6 ms per 1000 nodes) due to:
  - Irregular memory access patterns causing cache misses
  - Additional pointer dereferences for both left and right children
- Shows that asymptotic complexity (θ(n)) doesn't always equate to real life - implementation and memory access patterns significantly impact real performance

3 Part B: Order-Statistic Trees

3.1 Theoretical Foundation

Order-statistic trees augment BST nodes with a **size attribute**:

- $x.size$ = number of nodes in subtree rooted at x
- Invariant: $x.size = x.left.size + x.right.size + 1$ (NULL nodes have size 0)
- Enables $O(\log n)$ rank queries without traversing the entire tree

3.2 Size Attribute Maintenance

3.2.1 Main Change

The main change is maintaining the size invariant during insertions and deletions. Every ancestor of an inserted/deleted node must have its size updated.

3.2.2 OS-TREE-INSERT: Maintaining Size During Insertion

Strategy: Increment size along the path from root to insertion point.

Algorithm 5: OS-TREE-INSERT(T, z)

```
1  Tree T, node z to insert
2  y ← NULL;
3  x ← T.root;
4  while x ≠ NULL do
5  |  y ← x;
6  |  x.size ← x.size + 1 ;                    /* INCREMENT size along path */
7  |  if z.key < x.key then
8  |  |  x ← x.left;
9  |  else
10 |  |  x ← x.right;
11 |  end
12 end
13 z.p ← y;
14 if y = NULL then
15 |  T.root ← z;
16 else
17 |  if z.key < y.key then
18 |  |  y.left ← z;
19 |  else
20 |  |  y.right ← z;
21 |  end
22 end
23 z.size ← 1 ;                                 /* New node has size 1 */
```

Explanation:

- **Line 6**: As we traverse downward searching for insertion point, we increment each node's size by 1
- **Why this works**: The new node z will become part of the subtree rooted at every node x we visit, so each $x.size$ must increase by 1
- **Invariant preserved**: After insertion, every ancestor correctly shows the added node in its subtree count
- **Time complexity**: $O(h)$ - one increment per level during traversal

Example: Inserting key 15:

```
Before:           After:
   10(3)            10(4)   <- size incremented
   /   \            /   \
 5(1) 20(1)       5(1) 20(2) <- size incremented
                      /
                   15(1)     <- new node
```

3.2.3 OS-TREE-DELETE: Maintaining Size During Deletion

Strategy: Decrement size along path from root to deleted node, with special handling for successor movement.

Algorithm 6: OS-TREE-DELETE(T, z)

```
1  Tree T, node z to delete // Part 1:  Decrement size along path from root to z
2  current ← T.root;
3  while current ≠ NULL do
4  |  current.size ← current.size − 1;
5  |  if z.key < current.key then
6  |  |  current ← current.left;
7  |  else
8  |  |  if z.key > current.key then
9  |  |  |  current ← current.right;
10 |  |  else
11 |  |  |  break;
12 |  |  end
13 |  end
14 end
   // Part 2:  Standard deletion with size updates
15 if z.left = NULL then
16 |  TRANSPLANT(T, z, z.right);
17 else if z.right = NULL then
18 |  TRANSPLANT(T, z, z.left);
19 else
20 |  y ← TREE-MINIMUM(z.right);
21 |  if y.p ≠ z then
   |  |  // Decrement sizes along path from y to z
22 |  |  temp ← y.p;
23 |  |  while temp ≠ z do
24 |  |  |  temp.size ← temp.size − 1;
25 |  |  |  temp ← temp.p;
26 |  |  end
27 |  |  TRANSPLANT(T, y, y.right);
28 |  |  y.right ← z.right;
29 |  |  y.right.p ← y;
30 |  end
31 |  TRANSPLANT(T, z, y);
32 |  y.left ← z.left;
33 |  y.left.p ← y;
34 |  y.size ← SIZE(y.left) + SIZE(y.right) + 1;
35 end
```

Explanation:

- **Phase 1 (Lines 2-13)**: Traverse from root to z, decrementing each node's size. This accounts for z being removed from their subtrees.
- **Case 1 & 2 (Lines 15-18)**: If z has at most one child, Part 1 is enough.
- **Case 3 (Lines 20-36)**: Two children - z replaced by successor y:

- **Lines 23-28**: If y is not a direct child of z, decrement sizes along path from $y.p$ to z (nodes losing y from their subtrees)
- **Line 35**: Recompute y's size based on its new children (inheriting z's subtrees)

- **Why this works**:
  - Part 1 removes z from ancestor counts
  - When moving successor y up, we adjust nodes that lose y from their subtrees
  - Final size recomputation ensures y has correct count in new position
- **Time complexity**: $O(h)$ - path traversals are bounded by height

Invariant Preservation:
After any insertion or deletion, the invariant $x.size = x.left.size + x.right.size + 1$ holds for all nodes. This is verified by:

- INSERT: Only modifies ancestors of new node, incrementing their counts correctly
- DELETE: Adjusts all affected paths and recomputes sizes for restructured nodes

3.3 Additional Algorithms

3.3.1 OS-SELECT

Find the i-th smallest element using size attributes:

Algorithm 7: OS-SELECT(x, i)

```
1  Root x, rank i Node with rank i
2  if x = NULL then
3  |  return NULL;
4  end
5  r ← SIZE(x.left) + 1 ;                          /* Rank of x in its subtree */
6  if i = r then
7  |  return x;
8  else
9  |  if i < r then
10 |  |  return OS-SELECT(x.left, i);
11 |  else
12 |  |  return OS-SELECT(x.right, i − r);
13 |  end
14 end
```

Complexity: $O(h)$ - at most one recursive call per level.

3.3.2 OS-RANK

Determine the rank of a given node:

Algorithm 8: OS-RANK(T, x)

```
1  Tree T, node x Rank of x
2  r ← SIZE(x.left) + 1;
3  y ← x;
4  while y ≠ T.root do
5  |  if y = y.p.right then
6  |  |  r ← r + SIZE(y.p.left) + 1;
7  |  end
8  |  y ← y.p;
9  end
10 return r;
```

Complexity: $O(h)$ - walks up the tree, accumulating ranks.

3.4 Experimental Methodology

Same approach as Part A:

- Tree sizes: 10 to 100,000 nodes
- 80 data points with exponential spacing
- Triple averaging: each size run 3 times
- 100 trees per run (small), 15-30 (large)
- Fisher-Yates shuffle for random insertion order

3.5 Results and Analysis

3.5.1 INSERT Comparison: OS-Tree vs BST

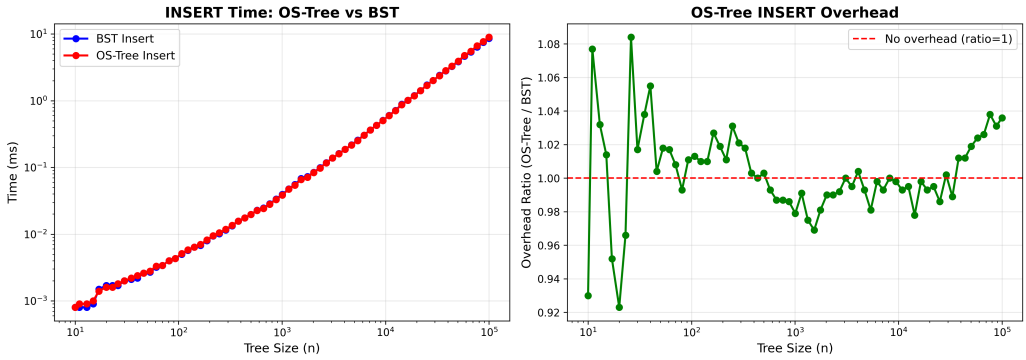

Figure 5: INSERT time comparison. OS-Tree and BST show nearly identical performance, with negligible overhead from size maintenance. Both axes use logarithmic scale.

Analysis:

- OS-Tree and BST insertion times are nearly identical (lines overlap in left graph)
- Overhead ratio fluctuates around 1.0, indicating negligible cost (typically $\pm 8\%$)
- Size increment operations add minimal overhead - just one integer increment per level
- Both maintain $O(n \log n)$ build time
- Small variations due to measurement noise and cache effects, not algorithmic differences

3.5.2 DELETE Comparison: OS-Tree vs BST

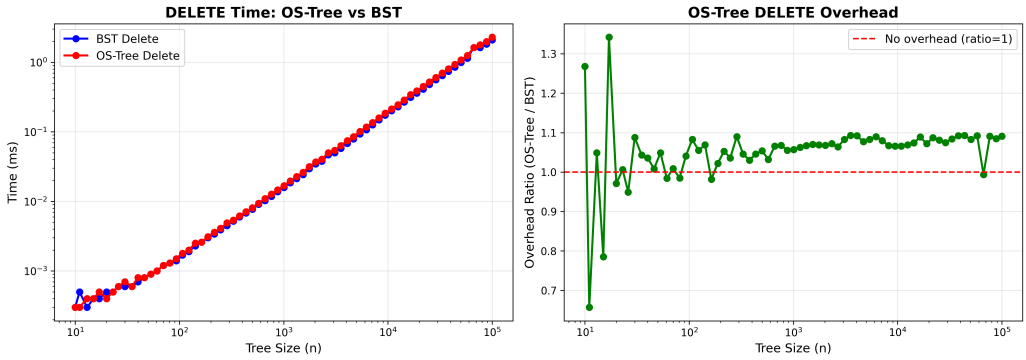

Figure 6: DELETE time comparison. OS-Tree shows 8-10% overhead due to size maintenance during deletion. Both axes use logarithmic scale.

Analysis:

- OS-Tree consistently 8-10% slower than plain BST (overhead ratio $\approx$ 1.08-1.10)
- Overhead from size decrements along deletion path and additional updates during successor movement
- Two-child deletion case requires extra path traversal: decrement from original position to successor, then decrement along successor's path during splicing
- DELETE has higher overhead than INSERT because successor replacement in two-child case requires size updates along **two paths** instead of one
- Both maintain $O(n \log n)$ complexity for destroying entire tree

3.5.3 OS-SELECT Runtime

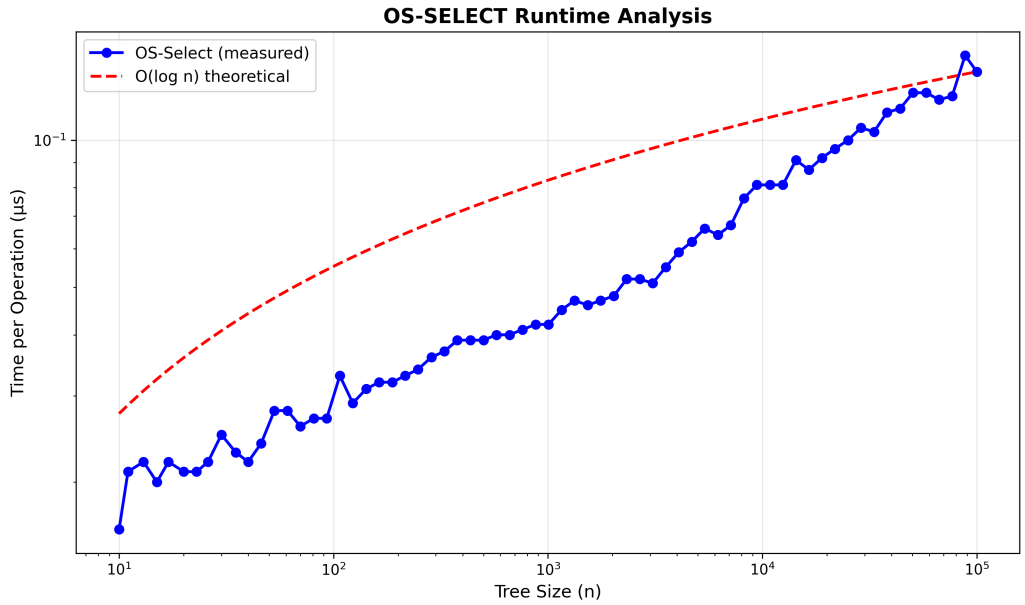

Figure 7: OS-SELECT runtime. Logarithmic growth confirms $O(\log n)$ complexity for rank queries. Both axes use logarithmic scale.

Analysis:

- Clear $O(\log n)$ behavior
- At $n = 100,000$: $\sim$0.15 microseconds per query
- Size attribute enables direct navigation to i-th element

3.5.4 OS-RANK Runtime

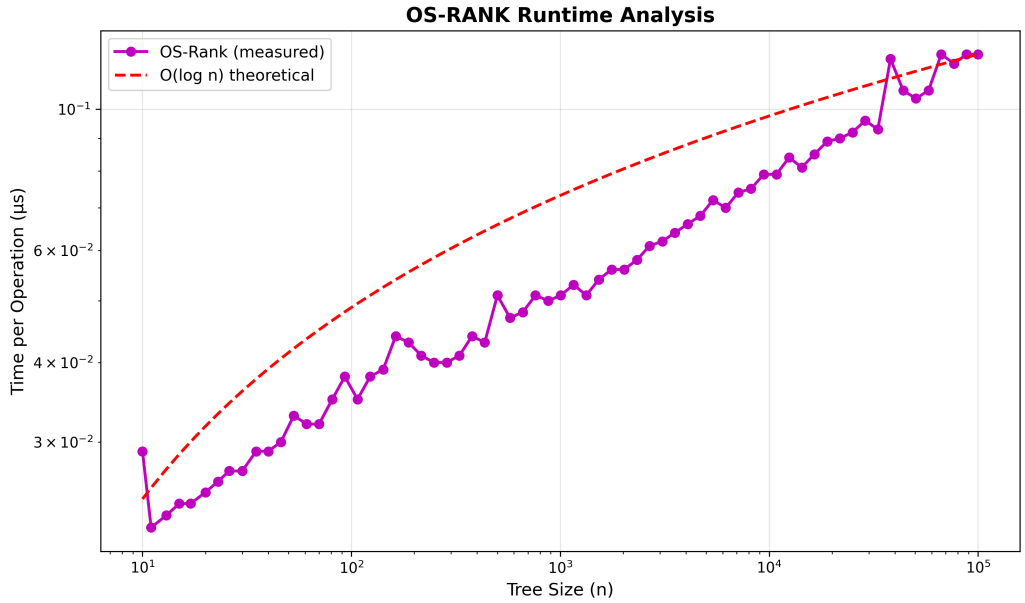

Figure 8: OS-RANK runtime. Logarithmic growth confirms $O(\log n)$ complexity for determining element rank. Both axes use logarithmic scale.

Analysis:

- $O(\log n)$ confirmed
- Slightly faster than OS-SELECT (walks up tree, no recursion overhead)
- Size attribute eliminates need for subtree traversal

4 Discussion and Conclusion

4.1 Theory vs. Practice

Our experiments confirm the theoretical predictions from the textbook:

- **Theorem 12.4 validated**: Random BSTs achieve $O(\log n)$ height, not $O(n)$
- **Randomization is critical**: Sorted insertion degrades to $O(n^2)$, random achieves $O(n \log n)$

4.2 Shuffle Method Equivalence

All three random shuffling algorithms (Fisher-Yates, RANDOMIZE-IN-PLACE, PERMUTE-BY-SORTING) produced statistically identical results:

- Same expected heights ($\approx 2.99 \ln n$)
- Same build/delete times
- Validates that uniform random permutations are the key, not the specific algorithm

4.3 Size Maintenance Strategy

The path-based size maintenance approach is simple and efficient: **Advantages:**

- Simple to implement and understand
- No additional asymptotic cost ($O(h)$ work during $O(h)$ operations)
- Preserves invariant automatically through incremental updates

Why it works:

- INSERT: Every ancestor gains one node in its subtree $\rightarrow$ increment along path
- DELETE: Every ancestor loses one node from its subtree $\rightarrow$ decrement along path
- Successor movement handled by adjusting the specific affected path

4.4 Performance Tradeoffs

Order-statistic trees demonstrate a classic space-time tradeoff:

Operation	**Plain BST**	**OS-Tree**
INSERT	$O(\log n)$	$O(\log n)$
DELETE	$O(\log n)$	$O(\log n)$
Find i-th smallest	$O(n)$	$O(\log n)$
Find rank	$O(n)$	$O(\log n)$
Space per node	3 pointers	3 pointers + 1 int

Table 2: Performance comparison: OS-Tree vs plain BST

The small constant-factor overhead on INSERT/DELETE is worthwhile when rank queries are needed, as they improve from $O(n)$ to $O(\log n)$ - an exponential speedup.

4.5 Practical Implications

- **Always randomize input**: Never insert into BST in sorted order
- **OS-Trees are practical**: Small overhead for the added ability of rank query.
- **Height matters**: Tree shape dominates performance - therefore try to keep it balanced