COMS4032A: Applications of Algorithms – Assignment 2

Nihal Ranchod
2427378
BSc Hons

September 17, 2024

Contents

1 Introduction

This assignment explores the claim in Theorem 12.4 from Chapter 12 of *Introduction to Algorithms* by Thomas H et al. 2009 that a randomly built binary search tree (BST) on n distinct keys has an expected height of $\mathcal{O}(n \log n)$. To investigate this, we will conduct experiments on BSTs built from randomly shuffled lists of keys, examining both the height and the time complexity of tree operations. Specifically, we will evaluate the performance of the `TREE-INSERT` and `TREE-DELETE` algorithms as described in the textbook.

In addition to analysing the asymptotic behaviour of BST height, we will measure the time taken to build and destroy trees of varying sizes. Finally, we extend the analysis to order-statistic trees by implementing `OS-SEARCH` and `OS-RANK` operations on a BST augmented with size attributes, ensuring that all operations maintain the integrity of the size information throughout insertion and deletion.

All algorithms implemented in C++ and the results of the experiments are available in the following GitHub repository: GitHub Link.

2 Graphing & Experimental Methodology

This section describes the methodology used to obtain the graphs, including the range of dimensions, key values, number of trees of each size, and other relevant details.

2.1 List Generation

The list of keys for each tree was generated as follows:

- For each value of n (the number of keys), a list of integers from 1 to n was created.
- This list was then randomised to simulate different insertion orders.

2.2 Randomisation

To ensure that the insertion order of keys was randomised:

- The `random_shuffle` function from the C++ Standard Library was used to shuffle the list of keys.
- This randomisation was performed for each trial to simulate different tree structures.

2.3 Number of Keys (n)

The number of keys (n) varied across a range of values to observe the behaviour of the Binary Search Tree (BST) for different sizes. The specific values of n used were:

- 10, 50, 100, 500, 1000, 5000, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000

The range of n values spans from small to large sizes:

Small Sizes: Starting from 10 keys, the experiments include small trees, which help in understanding the behaviour and performance of the BST in scenarios where the tree is not deeply nested.

Large Sizes: Going up to 100,000 keys, the experiments cover large trees, which are crucial for analysing the scalability and efficiency of the BST operations as the tree grows.

The **incremental steps** in the range of n values are designed to capture the performance trends effectively:

Initial Small Increments: For smaller values of n (e.g., 10, 50, 100), the increments are relatively small. This helps in capturing the detailed performance characteristics and any anomalies that might occur in smaller trees. Larger Increments for Larger Sizes: As n increases, the increments become larger (e.g., 10,000, 20,000, ..., 100,000). This is because the performance trends for larger trees can be observed over broader ranges, and larger increments help in reducing the number of experiments while still capturing the overall trend.

2.4 Number of Tress of Each Size

For each value of n, 50 trials were conducted. In each trial:

- A new list of keys was generated and randomised.
- A BST was built using the randomised list of keys.
- The height of the BST was calculated.
- The time taken to build and destroy the BST was measured.

2.5 Measurements

The following measurements were taken for each trial:

1. Height of the BST: The height of the tree was calculated after all keys were inserted.
2. Build Time: The time taken to insert all keys into the BST was measured using the `high_resolution_clock` from the C++ Standard Library.
3. Destroy Time: The time taken to delete all keys from the BST was measured using the `high_resolution_clock`.

2.6 Averaging Results

For each value of n, the average height, build time, and destroy time were calculated over the 50 trials. These averages were then recorded in a CSV file for further analysis and graphing.

3 Correctness Check

To ensure the correctness of the order-statistic tree, particularly the proper maintenance of the `'size'` attribute, I conducted several operations—insertions and deletions—while inspecting the tree structure after each modification. This allowed me to verify that the `'size'` attribute of each node was correctly updated and reflected the number of nodes in its subtree, which is crucial for efficient order-statistic operations like `OS-SEARCH` and `OS-RANK`.

The following diagrams will illustrate how the tree structure evolves and how the size attributes of the nodes are adjusted accordingly. Additionally, the output of the program after each operation will be included to confirm that the `'size'` attribute is correctly maintained throughout all modifications.

3.1 Insertion Operations

The insertion process starts with adding a series of values to the tree. After each insertion, the size of the affected nodes is recalculated to include the newly inserted node. For instance, when a node is inserted, every ancestor of that node should have its size incremented by one. This ensures that the `'size'` attribute is accurate across the entire tree.

In the code provided, after inserting the values: `10, 5, 15, 3, 7, 12, 18, 1, 4, 6, 8`

I display the tree, which allows us to visually inspect whether the structure is correct. When a new node, such as `13`, is inserted, we observe the change in tree structure and verify that the `'size'` attribute of each affected node is updated accordingly. Said operation can be seen in the following figures 1 - 4:

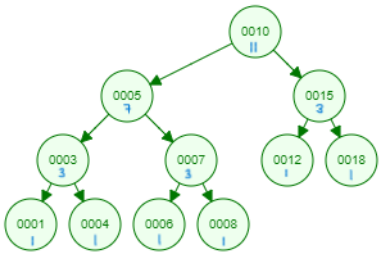

Figure 1: Correct Order-Statistic Tree after inserting list values.

```
Inorder Traversal of the Tree: Key: 1, Size: 1 | Key:
3, Size: 3 | Key: 4, Size: 1 | Key: 5, Size: 7 | Key:
6, Size: 1 | Key: 7, Size: 3 | Key: 8, Size: 1 | Key:
10, Size: 11 | Key: 12, Size: 1 | Key: 15, Size: 3 | K
ey: 18, Size: 1 |
```

Figure 2: Code output showing keys and respective values after inserting list values.

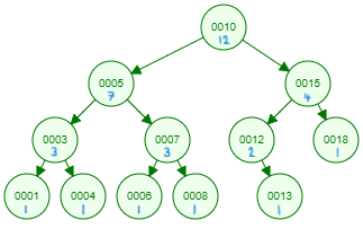

Figure 3: Correct Order-Statistic Tree after inserting new node 13.

```
Inserted key 13
Inorder Traversal of the Tree: Key: 1, Size: 1 | Key:
3, Size: 3 | Key: 4, Size: 1 | Key: 5, Size: 7 | Key:
6, Size: 1 | Key: 7, Size: 3 | Key: 8, Size: 1 | Key:
10, Size: 12 | Key: 12, Size: 2 | Key: 13, Size: 1 | K
ey: 15, Size: 4 | Key: 18, Size: 1 |
```

Figure 4: Code output showing keys and respective values after inserting new node 13.

3.2 Deletion Operations

Removing a node affects not only the structure of the tree but also the ‘size‘ attributes of all its ancestors. When deleting a node, such as the root (key 10), the tree must properly maintain its binary search tree properties, while also ensuring that the ‘‘size‘ attribute is correctly decremented for all affected nodes. After deleting the root, the tree is restructured to ensure that the next appropriate node becomes the new root, and the ‘size‘ attributes of the ancestors of the deleted node are updated. In the provided example, after deleting the root (10), the new root (after the tree rearrangement) is displayed, and we can confirm that the ‘size‘ values reflect the correct subtree sizes. This can be seen in figures 5 - 8.

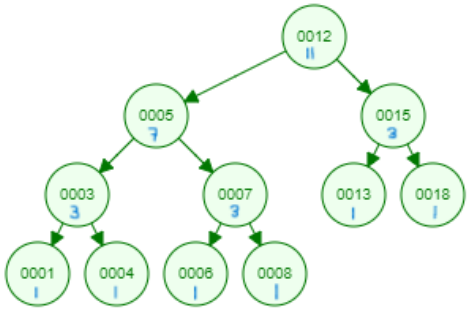

Figure 5: Correct Order-Statistic Tree after deleting node 10 (root).

```
Deleted root 10
Inorder Traversal of the Tree: Key: 1, Size: 1 | Key:
3, Size: 3 | Key: 4, Size: 1 | Key: 5, Size: 7 | Key:
6, Size: 1 | Key: 7, Size: 3 | Key: 8, Size: 1 | Key:
12, Size: 11 | Key: 13, Size: 1 | Key: 15, Size: 3 | K
ey: 18, Size: 1 |
New root: 12
```

Figure 6: Code output showing keys and respective values after deleting node 10 (root).

Further deletions, such as deleting node 5, continue to follow the same process. We check that after each deletion, the structure of the tree is still valid and the ‘size‘ attributes correctly track the number of nodes in each subtree.

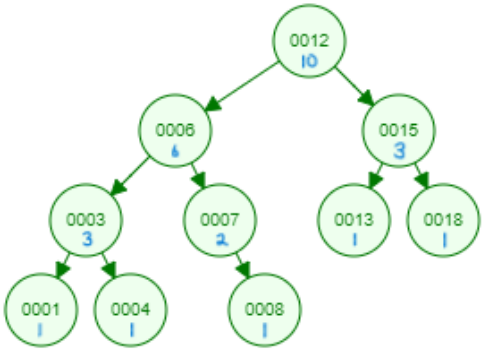

Figure 7: Correct Order-Statistic Tree after deleting node 5.

```
Delete node 5:
Inorder Traversal of the Tree: Key: 1, Size: 1 | Key:
3, Size: 3 | Key: 4, Size: 1 | Key: 6, Size: 6 | Key:
7, Size: 2 | Key: 8, Size: 1 | Key: 12, Size: 10 | Key
: 13, Size: 1 | Key: 15, Size: 3 | Key: 18, Size: 1 |
New left child of root: 6
```

Figure 8: Code output showing keys and respective values after deleting node 5.

4 Results

4.1 Part A Graphs: Binary Search Tree

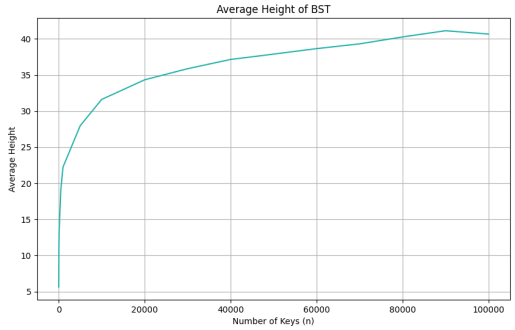

Figure 9: BST Average Expected Height vs. Number of keys (n)

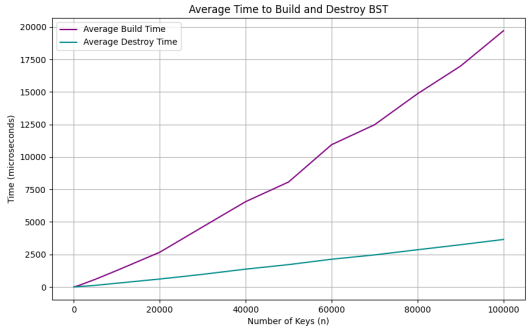

Figure 10: BST Average Build and Destroy Times vs. Number of keys (n)

4.2 Part B Graphs: Order-Statistic Tree

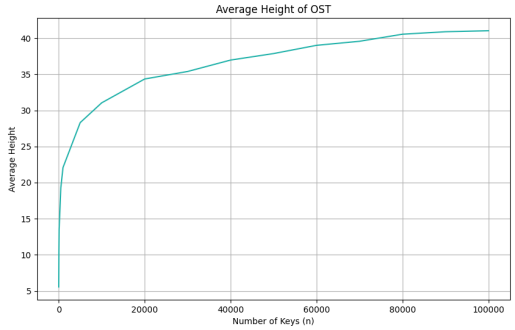

Figure 11: OST Average Expected Height vs. Number of keys (n)

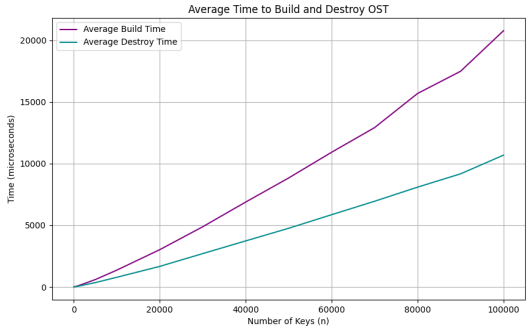

Figure 12: OST Average Build and Destroy Times vs. Number of keys (n)

4.3 Combined Graphs

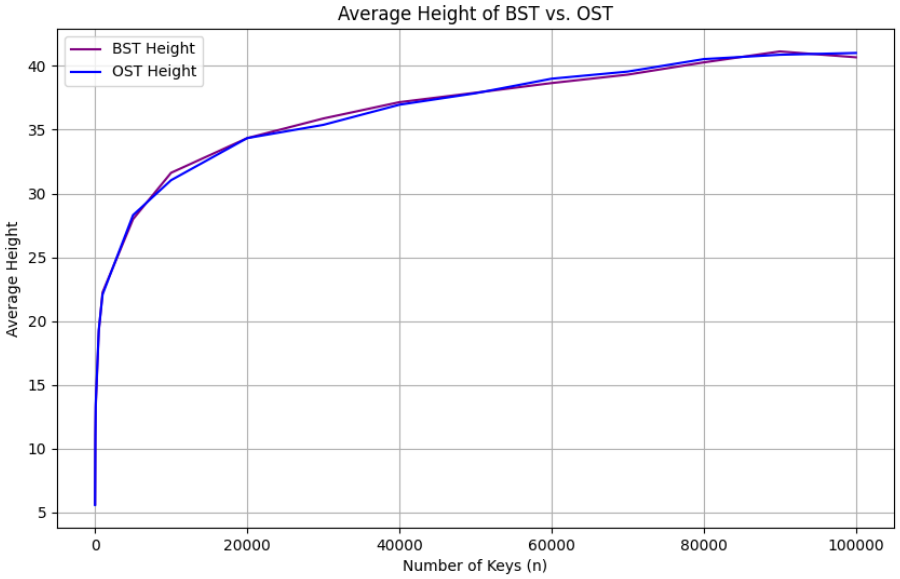

Figure 13: BST & OST Average Expected Height vs. Number of keys (n)

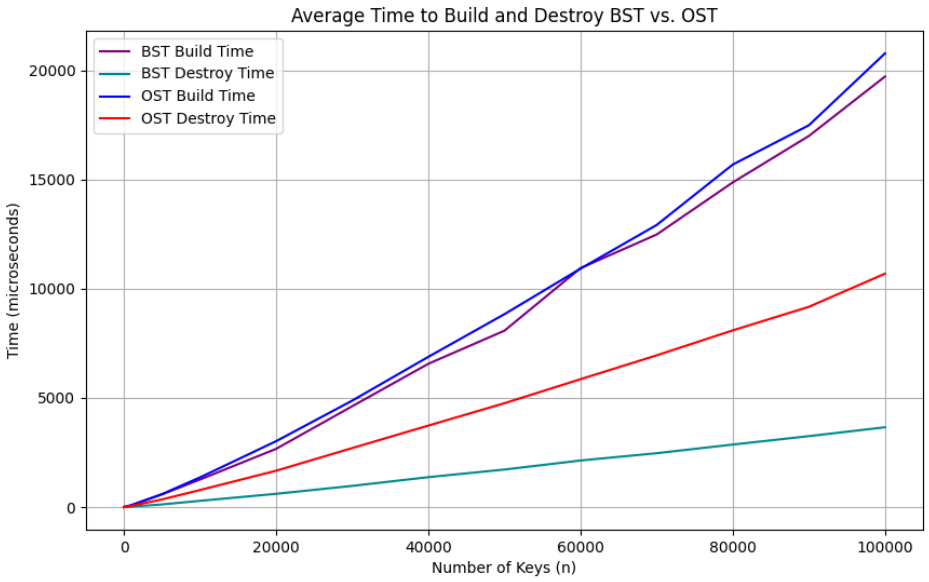

Figure 14: BST & OST Average Build and Destroy Times vs. Number of keys (n)

5 Analysis

5.1 Average Height of BST vs. OST

Figure 13 depicts the average height of the trees as the number of keys (n) increases. Both BSTs and OSTs exhibit very similar growth patterns in height, with the height stabilising around $\mathcal{O}(\log n)$, as predicted by Theorem 12.4 in the textbook. The slight differences between the heights of the two trees are minimal, which indicates that augmenting the BST to create an OST does not significantly affect the tree's height.

As n increases, the height of both trees follows the expected logarithmic growth, as the number of levels increases slowly relative to the number of nodes. This behaviour aligns with theoretical expectations that the height of a randomly built BST should be $\mathcal{O}(\log n)$ for large n, and the experiments confirm that this also applies to OSTs.

5.2 Average Time to Build and Destroy BST vs. OST

Figure 14 provides a comparative analysis of the time required to build and destroy both BSTs and OSTs for increasing numbers of keys.

5.2.1 Build Times

As shown in the figure 14, the time to build both types of trees increases as n grows, with the OST taking slightly longer to construct than the BST. This is expected because, in addition to performing regular BST operations, OSTs must also maintain the `‘size‘` attribute during insertion. The cost of updating the size attribute at each node causes the slight overhead observed in OST construction time.

5.2.2 Destroy Times

The destroy operation, which involves repeatedly deleting nodes, shows a more noticeable difference between the two structures. While the BST destroy time grows at a moderate rate, the OST destroy time increases significantly faster. This can be attributed to the additional complexity of maintaining the `‘size‘` attribute when deleting nodes, which leads to a greater computational cost in OSTs.

5.3 Conclusion

From these experiments, we can conclude that augmenting a BST into an OST has a minimal impact on the tree's height, as both data structures exhibit logarithmic height growth as the number of keys increases. However, there is a measurable impact on the time required to build and destroy the trees, with OSTs incurring a higher cost due to the need to maintain the `‘size‘` attribute during insertions and deletions. These results validate the theoretical expectations of BST height while highlighting the practical trade-offs in time complexity when augmenting trees for order-statistic operations.

6 Algorithm Changes for Part B

In an Order-Statistic Tree (OST), each node is augmented with a `‘size‘` attribute, which tracks the number of nodes in the subtree rooted at that node (including the node itself).

6.1 Changes to `TREE-INSERT`

When a node is inserted into an OST, the size of each ancestor of the newly inserted node must be updated to reflect the increased size of their respective subtrees.

Key Changes:

- **Incrementation of the `‘Size‘` field during insertion:** As we traverse down the tree to find the insertion point, we increment the size of every node encountered.

Pseudocode for `TREE-INSERT`:

Algorithm 1 TREE-INSERT(key)

```
1:  z ← new TreeNode(key)
2:  y ← null
3:  x ← root
4:  while x ≠ null do
5:      y ← x
6:      x.size ← x.size + 1                          ▷ Increment size of subtree
7:      if z.key < x.key then
8:          x ← x.left
9:      else
10:         x ← x.right
11:     end if
12: end while
13: z.parent ← y
14: if y = null then
15:     root ← z
16: else if z.key < y.key then
17:     y.left ← z
18: else
19:     y.right ← z
20: end if
```

Explanation:

- As we move down the tree to find the correct position for the new node, we increment the size of every node along the way by 1. This ensures that the size attribute remains consistent and reflects the correct number of nodes in the subtree rooted at each node.
- Once the new node is inserted, its size is initialised to 1, representing a single-node subtree.

6.2 Changes to `TREE-DELETE`

When a node is deleted, the size of its ancestors must be updated to reflect the decreased number of nodes in their respective subtrees. After the deletion, we propagate the changes in size back up to the root.

Key Changes:

- **Updating the size field after deletion:** Once a node is deleted, we update the size field for each ancestor of the deleted node to account for the removed node.

Pseudocode for Tree-Delete:

Algorithm 2 TREE-DELETE(z)

```
1:  affectedNode ← null
2:  if z.left = null then
3:      TRANSPLANT(z, z.right)
4:      affectedNode ← z.parent
5:  else if z.right = null then
6:      TRANSPLANT(z, z.left)
7:      affectedNode ← z.parent
8:  else
9:      y ← TREE-MINIMUM(z.right)
10:     affectedNode ← y.parent ≠ z ? y.parent : y
11:     if y.parent ≠ z then
12:         TRANSPLANT(y, y.right)
13:         y.right ← z.right
14:         y.right.parent ← y
15:     end if
16:     TRANSPLANT(z, y)
17:     y.left ← z.left
18:     y.left.parent ← y
19: end if
20: delete z
21: if affectedNode ≠ null then                ▷ Update sizes starting from the affected node
22:     UPDATE-SIZE(affectedNode)
23: end if
```

Explanation:

- The `TREE-DELETE` function handles the deletion of a node from the BST. It first determines which node will be affected by the deletion, as this node's size will need to be updated. If the node to be deleted (z) has no left child, it is replaced by its right child using the transplant function. If it has no right child, it is replaced by its left child. If the node has both children, the function finds the node's successor (the minimum node in its right subtree) and replaces the node with its successor. The successor's right child is then transplanted to the successor's position, and the successor's left and right children are updated accordingly.

Algorithm 3 UPDATE-SIZE(node)

```
1:  while node ≠ null do
2:      newSize ← 1                                  ▷ Start with 1 to count the node itself
3:      if node.left ≠ null then
4:          newSize ← newSize + node.left.size
5:      end if
6:      if node.right ≠ null then
7:          newSize ← newSize + node.right.size
8:      end if
9:      if newSize = node.size then
10:         break                                    ▷ No further updates needed if size hasn't changed
11:     end if
12:     node.size ← newSize                          ▷ Update size of the node
13:     node ← node.parent                           ▷ Move up to the parent
14: end while
```

Explanation:

- The `UPDATE-SIZE` function is responsible for updating the size of a node and its ancestors in the tree. It starts from a given node and moves up to the root, updating the size attribute at each step. The size is initialised to 1 to account for the node itself. If the node has a left child, the size of the left subtree is added to the new size. Similarly, if the node has a right child, the size of the right subtree is added. The function only updates the size if it has changed; otherwise, it breaks out of the loop, as no further updates are needed. This ensures that the size attribute remains accurate and avoids unnecessary updates.

References

Thomas H, Cormen et al. (2009). *Introduction to Algorithms Third Edition*.