Binary Search Trees and Order-statistic Trees

Darshan Singh (2441307)

Part (a)

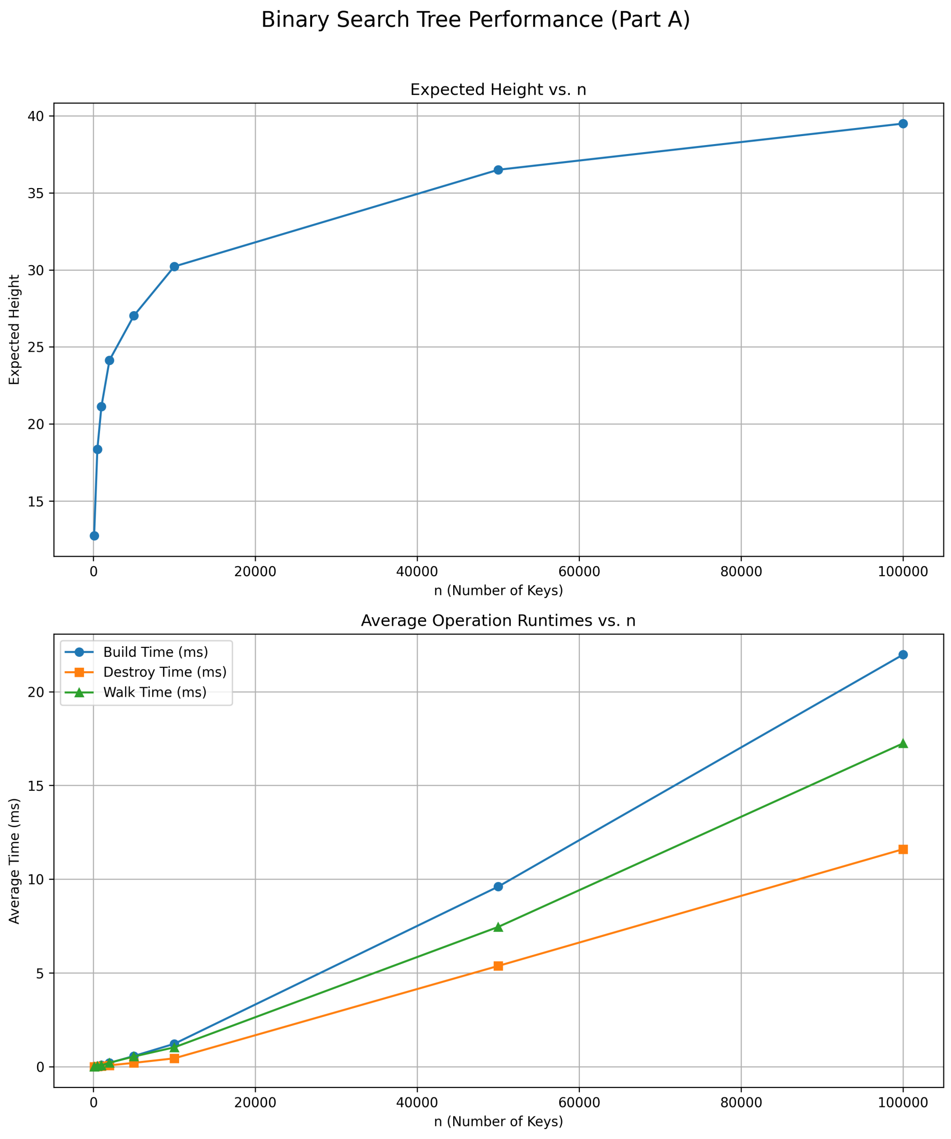


Figure 1: Performance analysis of a randomly built Binary Search Tree. The top graph shows the expected height, while the bottom graph displays the average runtimes for the ‘Build’, ‘Destroy’, and ‘Walk’ operations as a function of n.

These runtimes were obtained by choosing a range of problem sizes ‘n’ from 100 to 10000. Each of these problem sizes were staggered and then run under a parameter of `n_trials=50` times representing the amount of trees built for each size. These runtimes were then averaged in order to produce more accurate results. Keys used were from range 0 to n-1 and thereafter shuffled randomly (Fisher-Yates shuffle).

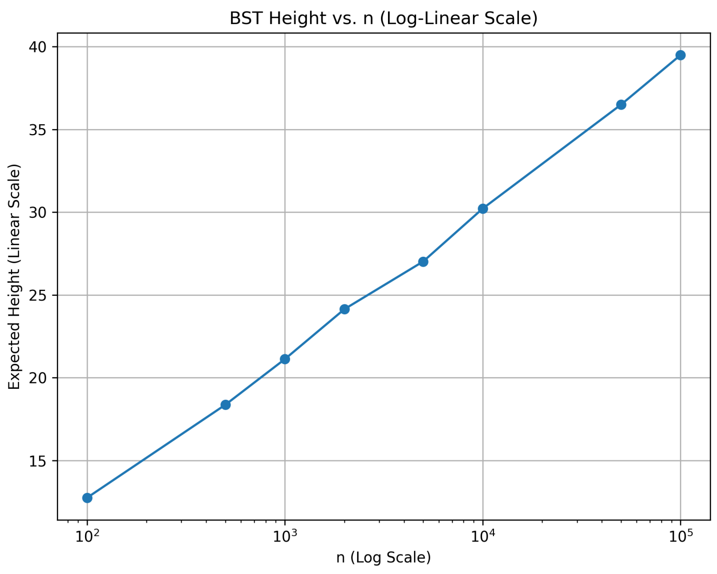


Figure 2: The expected height of a randomly built BST plotted against n on a log-linear scale. The nearly straight-line trend supports the theoretical O(lg n) complexity stated in Theorem 12.4.

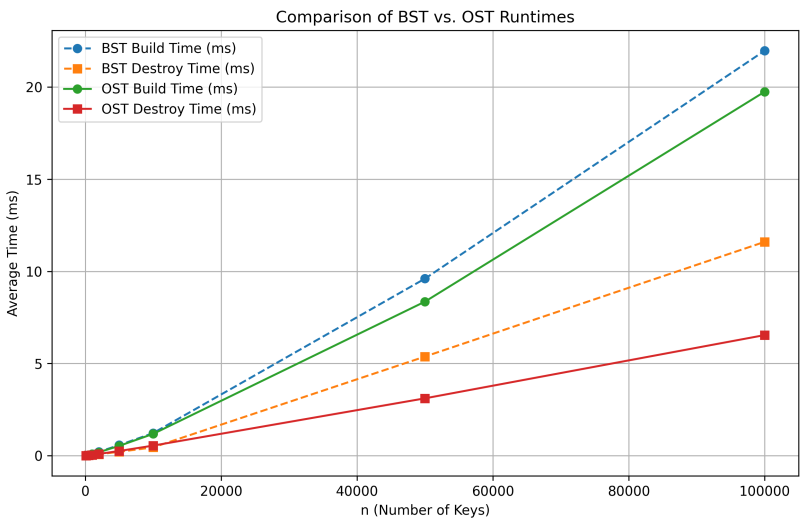
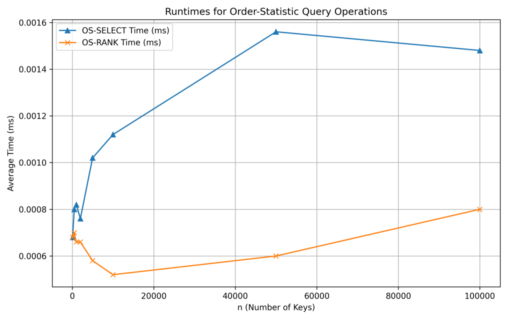
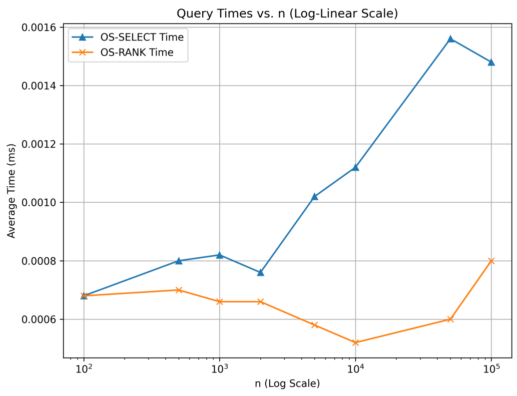


Figure 3: A comparison of the runtimes for a standard BST versus an Order-Statistic Tree (OST). The graph shows the overhead introduced by maintaining the 'size' attribute during the 'Build' and 'Destroy' operations.

Theoretically we would expect that overhead of maintaining the size for each node to specifically affect the `TREE-DELETE` function for the Order-Statistic tree. However using the exact same parameters from the baseline BST we see better runtimes for the OST in both cases.



(a) Runtimes for the ‘OS-SELECT’ and ‘OS-RANK’ query operations on an Order-Statistic Tree, showing logarithmic-time performance.



(b) Query times captured for OST algorithms.

Figure 4: Comparison of Order-Statistic Tree query performance across different algorithms.

The query performance of the Order-Statistic Tree is analysed in Figure 4. The results confirm the expected efficient runtimes for both the `OS-SELECT` and `OS-RANK` operations. In Figure 4b, the data plotted on a log-linear scale approximates a straight line, a trend which is indicative of a logarithmic time complexity, $O(\log n)$.

The variance observed in the `OS-SELECT` measurements can be attributed to the structural differences in the randomly generated trees used across the 50 trials. As the runtime of this function is proportional to the traversal depth required to find the median, minor variations in tree structure from one trial to the next result in the observed fluctuations.

A notable performance difference is observed between the two functions. The chosen test case for `OS-RANK` — finding the rank of the root node — is an $O(1)$ operation, as it only requires accessing the ‘size’ attribute of the root’s left child. This accounts for its faster and highly stable runtime. Conversely, the test for `OS-SELECT` requires finding the median element, necessitating an $O(\log n)$ traversal deep into the tree. This makes its runtime inherently longer and more susceptible to the structural variances between trials.

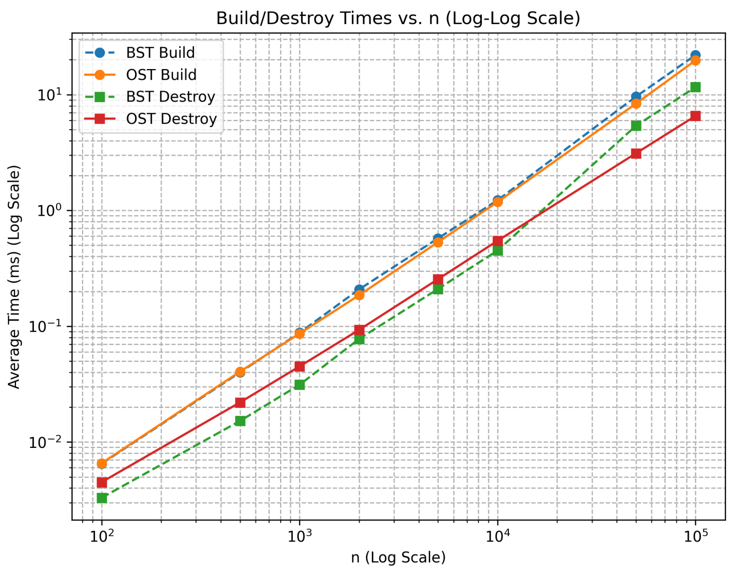


Figure 5: Log scaled comparison to reflect time complexities

The consistent set of parameters were also used for these OST queries. Notable for this experiment is what testing methods were selected for 'OS-SELECT' and 'OS-RANK'. For the Select, the median element's position was chosen. as the i-th smallest element. This, under assumption should give reasonable ranges depending how how varied the elements are randomly shuffled. For the Rank, the root was selected. In any tree this would be the size of it's left subtree. Given that the keys are randomly permuted, fairly varied results will come up in each of the 50 trials done. This provides a solid average-case for the function's performance.

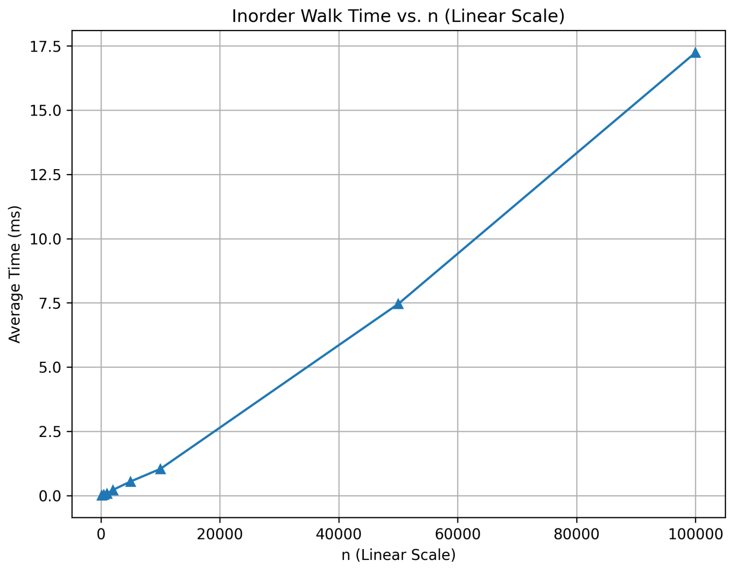


Figure 6: Linear plot of the `INORDER-TREE-WALK`
Similar parameters were used for this tree walk as it was run with the baseline BST. As seen the trend does confirm the experimental hypothesised runtime of $\Theta(n)$.

Part (b)

To transform the standard Binary Search Tree into an Order-Statistic Tree, the core data structure and its maintenance algorithms must be augmented. This is achieved by adding a `size` attribute to each node, which stores the total number of nodes in the subtree rooted at that node (including the node itself). Consequently, the `TREE-INSERT` and `TREE-DELETE` procedures must be modified to ensure this attribute is correctly maintained after any structural change to the tree.

Modifications to TREE-INSERT

The `TREE-INSERT` algorithm is modified to update the `size` attribute during its initial traversal. As the algorithm traverses down the tree to find the correct position for the new node, it increments the `size` of each node along this path. This ensures that all ancestors of the newly inserted node correctly reflect the increase in their subtree sizes. The new node itself is initialised with a size of 1.

Algorithm 1: Augmented TREE-INSERT

```
Input: A pointer to the tree's root $T$, a new node $z$
$y \leftarrow$ NIL;
$x \leftarrow T.root$;
while $x \neq NIL$ do
    $y \leftarrow x$;
    $y.size \leftarrow y.size + 1$;
    if $z.key < x.key$ then
        $x \leftarrow x.left$;
    end
    else
        $x \leftarrow x.right$;
    end
end
$z.parent \leftarrow y$;
if $y == NIL$ then
    $T.root \leftarrow z$;
end
else if $z.key < y.key$ then
    $y.left \leftarrow z$;
end
else
    $y.right \leftarrow z$;
end
```

Modifications to TREE-DELETE

Modifying `TREE-DELETE` is more complex, as the method for updating the `size` attribute depends on the structure of the node being removed. The core idea is to identify the parent of the node that is physically removed or moved from its original position and then traverse up to the root from that point, decrementing the `size` of each ancestor.

A helper procedure, `DECREMENT-ANCESTORS`, can be used to perform this upward traversal. The main `TREE-DELETE` logic then identifies the correct starting node for this procedure based on the following cases:

- **Case 1 & 2 (Node has 0 or 1 child):** The node z is removed and replaced by its single child or NIL. The update process starts from z's original parent, as its subtree is the first to be affected.
- **Case 3 (Node has 2 children):** The node z is replaced by its successor, y.
  - If the successor y is not a direct child of z, it is first moved from its original position to replace z. The size-update traversal must therefore begin from y's original parent.
  - If the successor y is a direct child of z, then y itself is the lowest point of structural change. After it replaces z and its new size is calculated from its children, the update traversal begins from y.

Algorithm 2: Augmented TREE-DELETE

```
Input: A pointer to the tree's root $T$, a node $z$ to delete
if $z.left == NIL$ then
    $startNode \leftarrow z.parent$;
    TRANSPLANT($T, z, z.right$);
end
else if $z.right == NIL$ then
    $startNode \leftarrow z.parent$;
    TRANSPLANT($T, z, z.left$);
end
else
    $y \leftarrow$ TREE-MINIMUM($z.right$);
    if $y.parent \neq z$ then
        $startNode \leftarrow y.parent$;
        TRANSPLANT($T, y, y.right$);
        $y.right \leftarrow z.right$;
        $y.right.parent \leftarrow y$;
    end
    else
        $startNode \leftarrow y$;
    end
    TRANSPLANT($T, z, y$);
    $y.left \leftarrow z.left$;
    $y.left.parent \leftarrow y$;
    $y.size \leftarrow (y.left.size) + (y.right.size) + 1$;
end
DECREMENT-ANCESTORS($startNode$);
```