Randomized Amortized Probabilistic Search Tree

In summary, the code leverages randomization to achieve an average-case complexity of O(NLogN) for RBST operations, such as insertion. However, in the worst case, the complexity can approach O(N^2) due to flattening and reinsertion operations. The chances of that taking place are infinitesimal.

Algorithm Overview:

`RBST initRBST()`

- Purpose: Initializes an RBST by creating an `Info` struct, allocating memory for it, and setting the root of the tree to NULL.
- Expected Complexity: O(1) - The initialization operation is constant time.

`void flattenRec(Node * node)`

- Purpose: Recursively flattens an RBST into a sorted linked list.
- Expected Complexity: O(N) - In the worst case, this function visits every node once, leading to linear time complexity.

`Node * insertRand(Node * node, Node * new, int size, int * visits)`

- Purpose: Randomly inserts a new node into the RBST. Depending on probability, it may flatten the tree into a linked list and then reinsert nodes.
- Expected Complexity: O(NLogN) - On average, each node is visited once for insertion. However, randomness ensures good average-case behavior.

`Node * insertNode(Node * node, Node * insert)`

- Purpose: Recursively inserts a new node into the RBST. Used for reinserting flattened subtrees.
- Expected Complexity: O(LogN) - In a balanced BST, the expected depth is logarithmic.

`Node * createNode(int key)`

- Purpose: Creates a new node with the specified key and initializes its left and right pointers to NULL.
- Expected Complexity: O(1) - Creating a node is a constant-time operation.

`int insertRBST(RBST bst, int key)`

- Purpose: Inserts a new key into the RBST using randomized insertion. It keeps track of the number of visits (comparisons) made during insertion.
- Expected Complexity: O(NLogN) on average - The expected depth of the tree is logarithmic, and each insertion may involve visiting a subset of nodes. The randomization improves average-case performance.

`void inOrderPrintBST(RBST tree)`

- Purpose: Provides a wrapper function to print the RBST in in-order traversal.

- Expected Complexity: O(N) - Printing all elements in the tree requires visiting every node once.

`void inOrderRec(Node * node)`

- Purpose: Recursively prints the keys of the RBST in in-order traversal.
- Expected Complexity: O(N) - In the worst case, it prints all nodes in the tree.

`int freeRBST(RBST bst)`

- Purpose: Frees the entire RBST and returns the number of nodes visited during the process.
- Expected Complexity: O(N) - In the worst case, this function visits every node once to deallocate memory.

`int freeTree(Node * tree, int * visits)`

- Purpose: A helper function to recursively delete nodes in the RBST.
- Expected Complexity: O(N) - In the worst case, it recursively visits every node to deallocate memory.

`int testInsertRBST(int n, int * array)`

- Purpose: Initializes an RBST and inserts a sequence of keys into it, tracking the total number of visits (comparisons) made during insertion.
- Expected Complexity: O(NLogN) on average - The randomization in insertion helps maintain good average-case performance.

`int testFreeRBST(int n, int * array)`

- Purpose: Tests the insertion and freeing of an RBST in a single function, returning the total number of visits made during insertion.
- Expected Complexity: O(NLogN) on average - Similar to `testInsertRBST`, the randomization helps achieve good average-case performance.

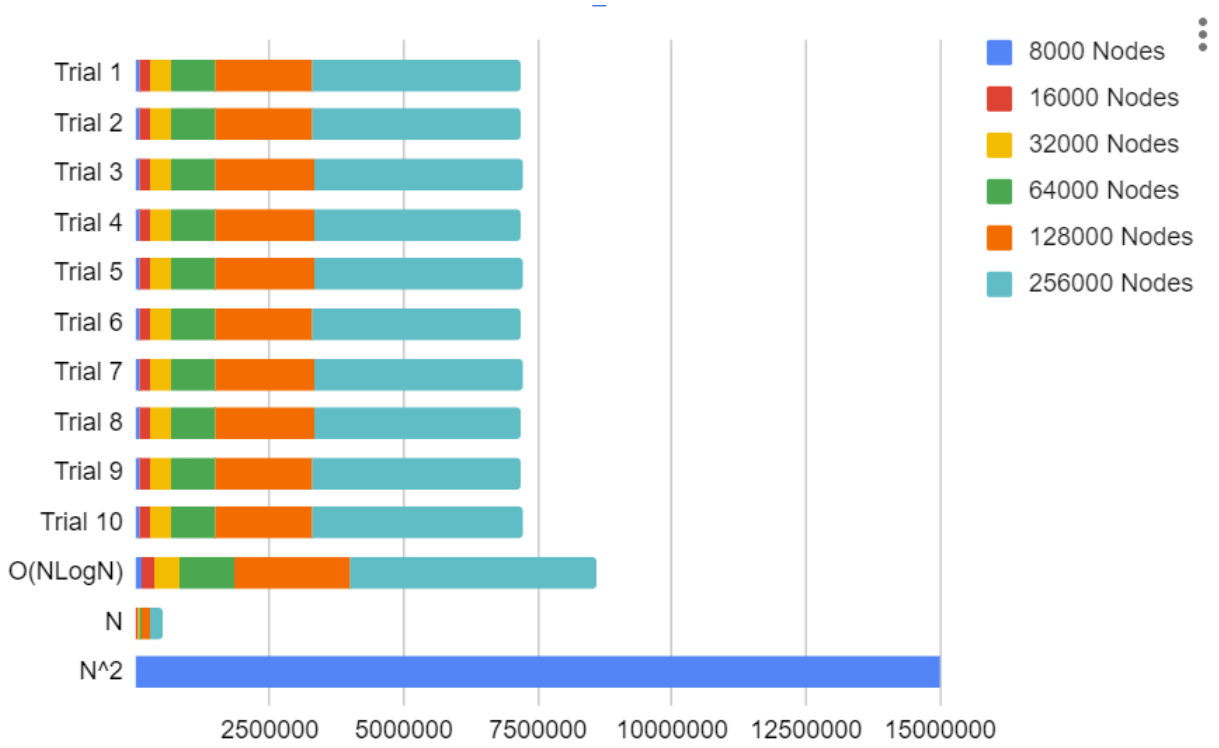
`int scalingTests(int tests_per)`

- Purpose: Conducts scaling tests for RBST insertion operations. It runs tests for different RBST sizes and compares the results to the expected complexity of O(NLogN).
- Expected Complexity: O(NLogN) on average - The tests involve inserting elements and comparing the average number of visits, which should grow logarithmically with input size.

Data Curated Via 'int scalingTests(int tests_per)':

This graph plots the number of visits resulting from 10 trials of insertion using varying sizes of data entered into the tree in sorted order. In a regular BST, entering values in sorted order would create a tree that is essentially the same as a linked list, with the values linked together via the right or left pointers, depending on how they are sorted. However, with the randomization portion of the insert function, each node insertion has a chance of becoming the root node of the current subtree of the insertion. This allows the nodes to be better dispersed, which promotes balance in the tree. This can be observed in the data returned. In a balanced tree an insertion should take around log(N), N being the size of the tree. So it makes sense that

insertion of N values should take around Nlog(N) visits, ideally a small amount below to account for the change in size of the tree. The graph compares the results with varying complexities and shows how it compares.



GPT-3.5 Credits:

This report was created using text from a report generated by ChatGPT, a language model based on GPT-3.5 architecture, developed by OpenAI. GPT-3.5 is a powerful natural language processing model capable of generating human-like text and providing information, explanations, and reports on various topics.