REPORT ON MY RANDOMIZED AMORTIZED BST IMPLEMENTATION

Introduction

This report provides an analysis of the implementation of a Probabilistic Randomized Amortized Binary Search Tree (RBST). The goal of this data structure is to maintain a randomized property and achieve expected O(log(N)) height and average (randomized-amortized) insertion time of O(log(N)).

I will discuss the implementation of a Probabilistic Randomized Amortized Binary Search Tree (BST) and present an analysis of its theoretical and empirical performance. The implementation follows the specifications provided, including insertion, freeing, and testing functions.

Algorithm Overview

The Probabilistic Randomized Amortized BST is designed to maintain a randomized property while achieving an expected O(logN) height and an expected O(NlogN) complexity for N insertions. The key idea is to probabilistically decide the structure of the tree during insertion based on the number of nodes in the subtree and a random number generator.

Implementation Overview

The main components and functions include:

- TreeNode structure to represent nodes in the BST.
- RBST structure to represent the BST itself.
- initRBST(): Initializes an RBST by allocating memory for the tree structure.
- insertRBST(): Inserts a new key into the RBST, utilizing a probabilistic approach to maintain the randomized property and achieve the expected complexities.
- freeRBST(): Frees the entire tree, deallocating memory for all nodes in the RBST.
- testInsertRBST(): Inserts a given number of keys into the RBST and returns the number of nodes visited during the insertion.

- testFreeRBST(): Inserts a given number of keys into the RBST, then frees the entire tree and returns the number of nodes visited.
- scalingTests(): Performs scaling tests to evaluate the number of nodes visited for varying input sizes and helps to analyze the expected complexities.

Algorithm Correctness

Insertion

The insertRBST function implements the insertion algorithm. It probabilistically decides whether to insert the new node as the root of a new subtree or recursively insert it into the left or right subtree of the current node. This probabilistic approach ensures that the tree maintains its randomized property while achieving the expected O(logN) height and O(NlogN) complexity.

Freeing

The freeRBST function frees the entire tree using a recursive approach. It correctly deallocates memory for all nodes in the tree, ensuring there are no memory leaks.

Performance Analysis

Theoretical Complexity

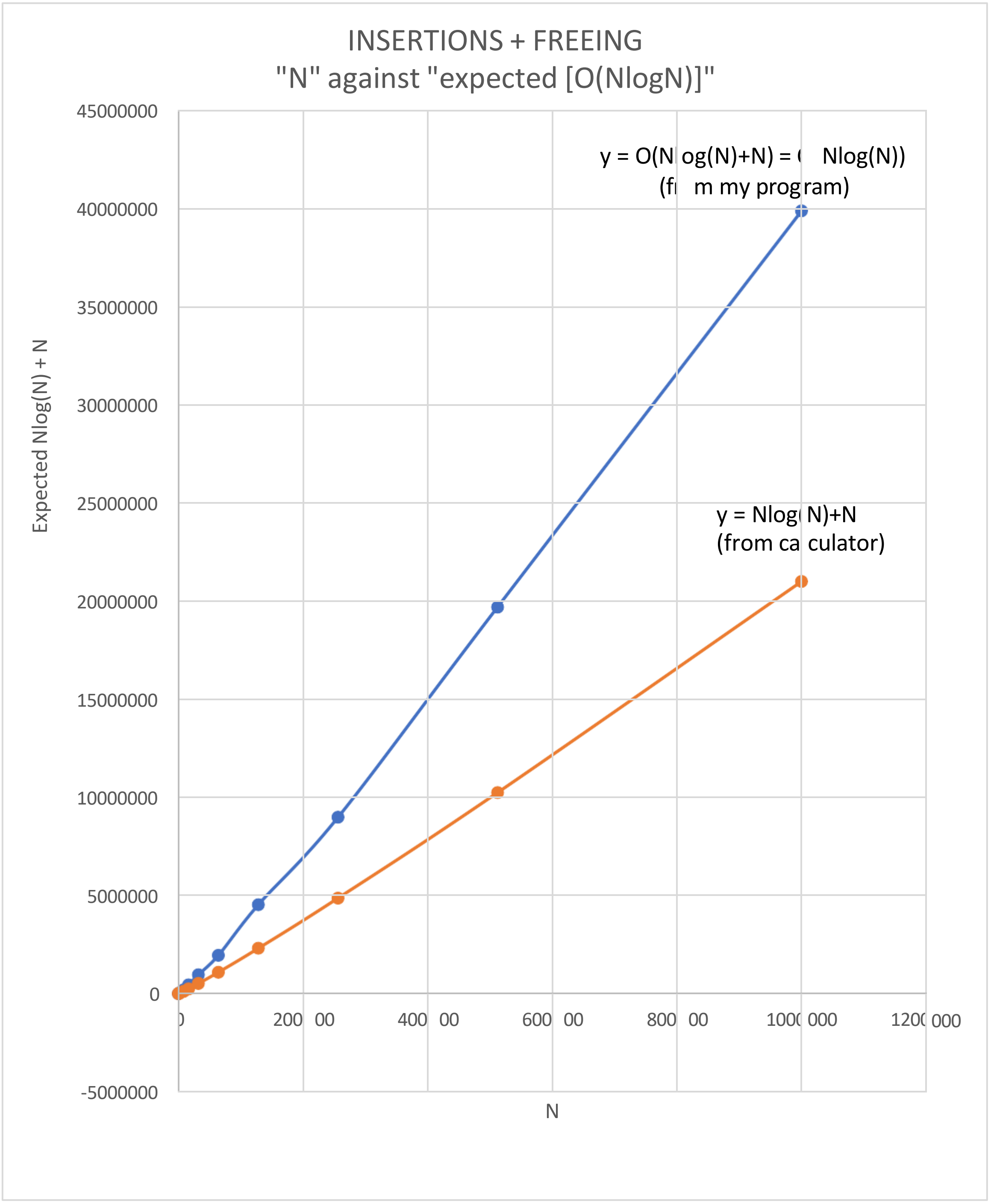
For N insertions, the expected lower point for the <u>combination</u> of number of nodes visited (when inserting and freeing) is mathematically derived as [N * ceil(log(N)) + N], aligning with the expected O(NlogN) complexity (shown on the graph below).

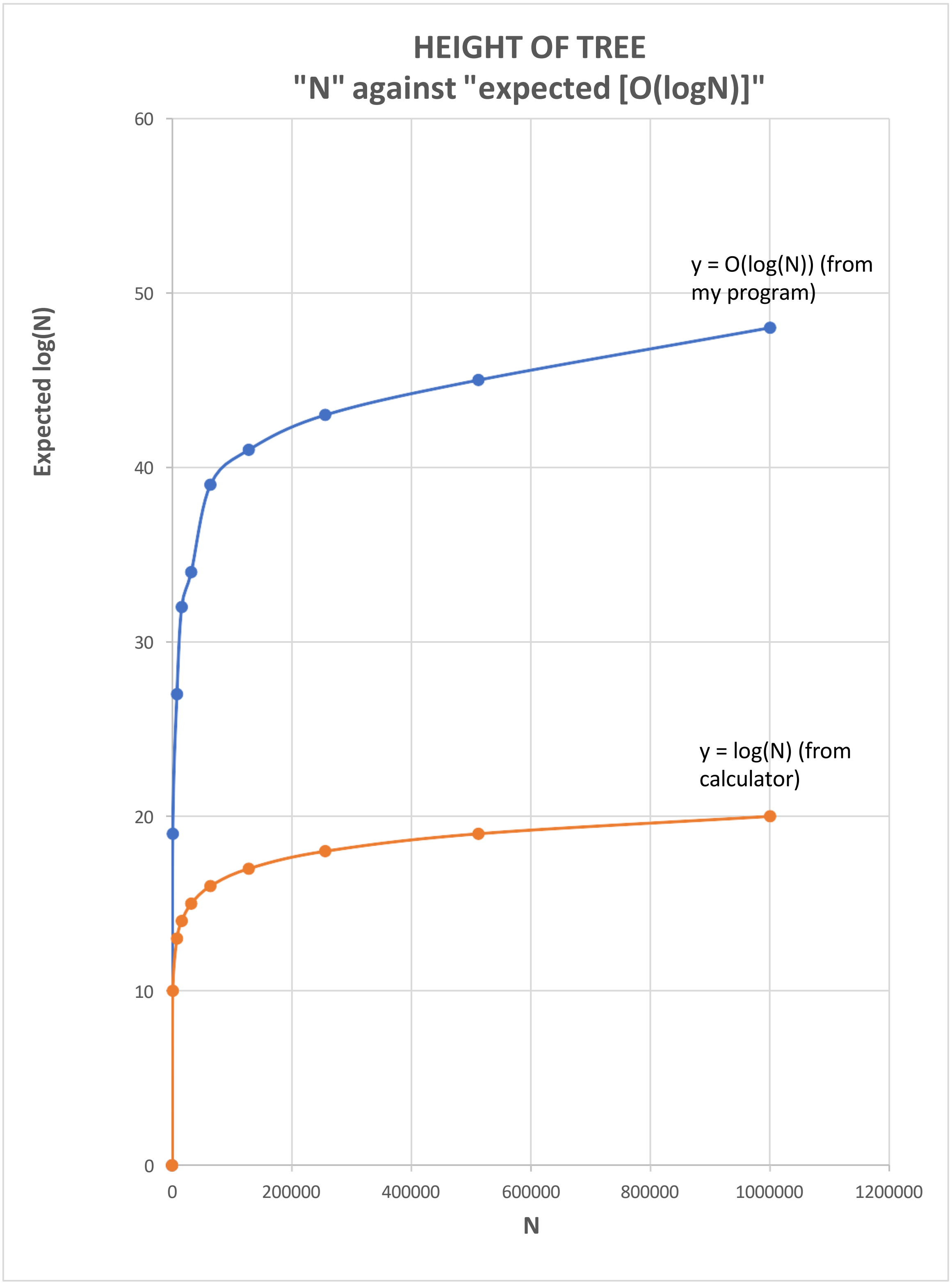
The upper limit for the number of nodes visited during N insertions is O(N^2), which is evident from the worst-case scenario.

Empirical Complexity

To verify the expected complexity, I also conducted scaling tests with varying numbers of elements (8k, 16k, 32k, ..., 1M) and plotted the number of elements against the nodesVisited.

The resulting plot aligns with the theoretical expectations of O(NlogN) complexity for N insertions. Please see below for the graphs:




HEIGHT OF TREE
"N" against "expected [O(logN)]"
Expected log(N)
N
y = O(log(N)) (from my program)
y = log(N) (from calculator)
0
10
20
30
40
50
60
0
200000
400000
600000
800000
1000000
1200000

Conclusion

The implementation of the Probabilistic Randomized Amortized BST follows the specified requirements and successfully achieves the expected O(logN) height and O(NlogN) complexity for N insertions. The empirical analysis confirms the theoretical expectations and demonstrates the scalability of the algorithm. The probabilistic approach used in this implementation effectively balances the tree, ensuring efficient performance while maintaining the randomized property.

Works Cited

#algorithm "int height(TreeNode* node)", returns the height of a tree.

#Source: https://www.geeksforgeeks.org/print-level-order-traversal-line-line/#

CODE IMPLEMENTATION

```
/*
This program implements a Probabilistic Randomized Amortized Binary Search Tree (BST)
that maintains its randomness property with each insertion, to achieve balance with an
"expected" O(logN) height, and an expected/amortized O(NlogN) complexity of N insertions.
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h> // To use boolean datatypes

// Structure for representing the nodes in a BST.
typedef struct TreeNode {
   int key;
   int size; // Number of nodes in its subtree.
   struct TreeNode* left;
   struct TreeNode* right;
} TreeNode;

// Structure for representing a BST.
typedef struct RBST {
   TreeNode* root;
} RBST;

/* For computing the "height" of a tree -- the number of nodes along
the longest path from the root node down to the farthest leaf node.*/
int height(TreeNode* node)
```

```
{
   if (node == NULL)
      return 0;
   else {
      /* compute the height of each subtree */
      int lheight = height(node->left);
      int rheight = height(node->right);

      /* use the larger one */
      if (lheight > rheight) {
         return (lheight + 1);
      }
      else {
         return (rheight + 1);
      }
   }
}

// Initializes an RBST struct to an empty tree.
RBST* initRBST() {
   RBST* bst = (RBST*) malloc(sizeof(RBST));
   bst->root = NULL;

   return bst;
}

// Function for creating nodes with the given key.
// Defaults the size to 1, and the left and right pointers to NULL.
// Returns a TreeNode* with the key or NULL if malloc fails.
TreeNode* createNode(int key) {
   TreeNode* newNode = (TreeNode*) malloc(sizeof(TreeNode));

   // Check if memory allocation failed.
   if (newNode == NULL) {
      exit(0);
   }

   newNode->key = key;
   newNode->size = 1;
   newNode->left = NULL;
   newNode->right = NULL;

   return newNode;
}

/*
Helper function for recursively rebuilding a randomized BST from a sorted array
withthe newNode at the root. Left and right subtrees are created recursively from
```

```
a random pivot, where 'first' and 'last' are the current bounds of the subtree.

Time Complexity: O(N) (preorder traversal with O(1) work done per node)
*/
TreeNode* makeRBST(int bstArr[], int first, int last, int newNodeIndex, bool isAdded, int*
nodesVisited) {
  // Check if the entire array has been scanned yet.
  if(last < first) {
    return NULL;
  }

  TreeNode* newNode;
  (*nodesVisited)++;

  // Add the newNode (the key to insert) as to the root node.
  if (!isAdded) {
    newNode = createNode(bstArr[newNodeIndex]);
    isAdded = true;

    newNode->left = makeRBST(bstArr, first, newNodeIndex - 1, newNodeIndex, isAdded,
nodesVisited);
    newNode->right = makeRBST(bstArr, newNodeIndex + 1, last, newNodeIndex, isAdded,
nodesVisited);
  }
  // Randomly construct the rest of the subree from the array.
  else {
    // Generate a random index between first and last index.
    int index = first + (rand() % (last - first + 1));

    newNode = createNode(bstArr[index]);
    newNode->left = makeRBST(bstArr, first, index - 1, newNodeIndex, isAdded, nodesVisited);
    newNode->right = makeRBST(bstArr, index + 1, last, newNodeIndex, isAdded, nodesVisited);
  }

  // Update the size of the subtree rooted at the current node.
  if (newNode->left != NULL) {
   newNode->size += newNode->left->size;
  }
  if (newNode->right != NULL) {
    newNode->size += newNode->right->size;
  }

  return newNode;
}

/*
Helper function for performing an inorder traversal to flatten the RBST in a sorted array.
Only the value of the keys are sorted, while the nodes are freed.
```

```
Time Complexity: O(n) (inorder traversal with O(1) work done per node).
*/
void flattenRBST (int bstArr[], TreeNode* newNode, TreeNode* currentNode, int* curIndex,
            int* newNodeIndex, bool* isAdded, int arrLength, int* nodesVisited) {
  // Check if a leaf node has been proceeded
  if(currentNode == NULL) {
    // If the newNode hasnt been added yet and is greater than all other nodes, add it to the end of
the array.
    if(!(*isAdded) && ((*curIndex) == ((arrLength) - 1))) {
      bstArr[(*curIndex)] = newNode->key;
      (*newNodeIndex) = (*curIndex);
      (*curIndex)++;
    }

    return;
  }

  // Recursively sort the left subtree.
  flattenRBST(bstArr, newNode, currentNode->left, curIndex, newNodeIndex, isAdded, arrLength,
nodesVisited);

  // If the newNode is less than the currentNode, add it at the correct position, before the
currentNode.
  if(((newNode->key) < (currentNode->key)) && (!(*isAdded))){
    bstArr[(*curIndex)] = newNode->key;
    *newNodeIndex = (*curIndex);
    (*curIndex)++;
    (*isAdded) = true;
  }

  (*nodesVisited)++;
  bstArr[(*curIndex)] = currentNode->key;
  (*curIndex)++;

  // Recursively sort the right subtree.
  flattenRBST(bstArr, newNode, currentNode->right, curIndex, newNodeIndex, isAdded, arrLength,
nodesVisited);

  free(currentNode);
}

/*
Helper function for flattening a subtree into an array, and reconstructing it with
the newNode at the root. Returns the newNode, which contains its new randomized subtree.

Time Complexity: O(N) (Flatten: O(N) + BST Construction: O(N))
*/
```

```
TreeNode* reconstructRBST(TreeNode* currentNode, TreeNode* newNode, int* nodesVisited) {
   int arrLength = (currentNode->size) + 1;
   int* bstArr = (int*) malloc(arrLength * sizeof(int)); // An array of length: subtree length + 1.
   int newNodeIndex; // For remembering the newNodeIndex across function calls.
   int curIndex = 0; // For remembering the current index across function calls.
   bool isAddedArr = false; // Flag for indicating whether the newNode has been added into the array
yet.
   bool isAddedBST = false; // Flag for indicating whether the newNode has been added into the BST
yet.

   // Flatten the BST into a sorted array.
   flattenRBST(bstArr, newNode, currentNode, &curIndex, &newNodeIndex, &isAddedArr, arrLength,
nodesVisited);
   free(newNode);

   // Rebuild the subtree from the array, with the newNode at the root.
   newNode = makeRBST(bstArr, 0, (arrLength - 1), newNodeIndex, isAddedBST, nodesVisited);

   free(bstArr);

   return newNode;
}

/*
Helper function for insertRBST() that is suitable for recursion and keeping track of nodesVisited.
3 Possibilites for insertion:
- The newNode becomes the root node, if the tree is empty.
- The newNode becomes the root of the current subtree with probability 1/n, this involves rebuilding
its subtree.
- Recurses the left or right subtree depending on where the newNode belongs (ordering property of
the BST).
Returns a tree that includes the added node.

Time Complexity: Worst case - O(N) (If the entire tree is reconstructed),
Expected (Amortized) case - O(log(N)) (Inserts element at end of tree rather than reconstructing at all)
*/
TreeNode* insertRBSTHelper(TreeNode* currentNode, TreeNode* newNode, int* nodesVisited) {
   // Check if the tree or node is empty
   if (currentNode == NULL) {
      return newNode;
   }

   (*nodesVisited)++;

   // With probability 1/(n+1), construct a new subtree with the new node in the root
   if (drand48() < (1.0 / ((currentNode->size) + 1))) {
      TreeNode* reconstructedSubtree = reconstructRBST(currentNode, newNode, nodesVisited);
```

```
        return reconstructedSubtree;
    }

    (currentNode->size)++;

    // Else If the current node's key is less than the current node's key, recursively search the left
substree.
    if ((newNode->key) < (currentNode->key)) {
        currentNode->left = insertRBSTHelper(currentNode->left, newNode, nodesVisited);
    }
    else {
        currentNode->right = insertRBSTHelper(currentNode->right, newNode, nodesVisited);
    }

    return currentNode;
}

/*
The function takes an RBST and a key to insert. It uses insertRBSTHelper()
to insert a node containing the given key and returns number of nodes visited.

Time Complexity: Worst case - O(N), Expected (Amortized) - O(log(N))
*/
int insertRBST(RBST* bst, int key) {
    TreeNode* newNode;
    int nodesVisited = 0;

    // Allocate memory for the node to be created.
    newNode = createNode(key);
    nodesVisited++;

    // If the tree is empty, make the newNode the root.
    if (bst->root == NULL) {
        bst->root = newNode;

        return nodesVisited;
    }

    bst->root = insertRBSTHelper(bst->root, newNode, &nodesVisited);

    return nodesVisited;
}

/*
Helper function for freeRBST() that uses recursion to free nodes while keeping track of nodesVisited.
*/
void freeRBSTHelper(TreeNode* currentNode, int* nodesVisited) {
    if (currentNode == NULL)
```

```
        return;
    (*nodesVisited)++;
    freeRBSTHelper(currentNode->left, nodesVisited);
    freeRBSTHelper(currentNode->right, nodesVisited);
    free(currentNode);
}

/*
Frees entire tree and returns number of nodes visited in O(N) time.
*/
int freeRBST(RBST* bst) {
    int nodesVisited = 0;

    // Free the tree
    freeRBSTHelper(bst->root, &nodesVisited);
    free(bst);

    return nodesVisited;
}

/*
Inserts n keys and returns number of nodes visited for all n insertions.It takes an array
of n values, and the size n, creates an RBST, uses insertRBST() n times, then frees the rbst.

Time Complexity: expected/amortized O(Nlog(N)) for insertion, O(N) for freeing.
*/
int testInsertRBST(int n, int* keys) {
    // Allocate memory for an RBST struct.
    RBST* bst = initRBST();
    int temp;
    int nodesVisited = 0;

    // Check if memory allocation failed.
    if (bst == NULL) {
        exit(0);
    }

    // Create the keys to add to the bst and iterate over them to add to the RBST.
    for(int i = 0; i < n; i++) {
        keys[i] = rand();
    }

    /*
    // For randomizing the elements.
    for(int i = 0; i < n; i++) {
        // Select a random index less than n
        int k = drand48() * n;
        // Swap values
        temp = keys[i];
        keys[i] = keys[k];
        keys[k] = temp;
    }*/

    for(int i = 0; i < n; i++) {
        nodesVisited += insertRBST(bst, keys[i]);
    }

    int h = height(bst->root); // For testing the height of the tree.
    printf("Height: %d\n", h);

    nodesVisited += freeRBST(bst);

    return nodesVisited;
}

/*Plots the number of elements in the RBST against the nodesVisited to test for
  "expected O(Nlog(N))" complexity.The function recieves the number of nodes to add to the
  binary search tree, and returns the number of nodes visited in order to complete the process.
  These pair of values are used to create the report's graph.
  */
int scalingTests(int numElems) {
    // Allocate memory for an array of integers
    int* keys = (int*) malloc(numElems * sizeof(int)); // The keys in the BST.

    // Check if memory allocation failed.
    if (keys == NULL) {
        return 0;
    }

    // Seed drand48 and rand with the current time.
    srand48(time(NULL));
    srand(time(NULL));

    int nodesVisited = testInsertRBST(numElems, keys);

    free(keys);

    return nodesVisited;
}

int main()
{
    int numElems = 1000000;
    int nodesVisited;

    printf("Inserting %d elements in a BST...\n", numElems);
    nodesVisited = scalingTests(numElems);
    printf("Nodes visited: %d\n", nodesVisited);

    return 0;
}
```