

HOMEWORK 04

INTRODUCTION:

A Binary Search Tree is a hierarchical data structure that is used to store and organize data. A binary search tree has a maximum of two children for each node: the left child and the right child. The main feature of a binary search tree is that, for every node, every other node in the left subtree has a value that is greater than the node's value, and every other node in the right subtree has a value that is less than the node's value. This feature enables efficient insertion, deletion, and searching procedures. These are flexible data structures that have many uses in computer science and other domains. They form the foundation of symbol tables in computer programming, enabling the effective mapping of identifiers to values or memory locations in interpreters, compilers, and programming language processing tools. Furthermore, they are widely used in database systems to support critical operations in the database like insertion, deletion, and searching. They also power indexing and searching operations, allowing quick data retrieval based on keys. They are essential to file systems because they help arrange directories and files, making it possible to perform effective operations like insertion and deletion as well as fast searches for file paths.

An algorithm for sorting data based on the ideas of Binary Search Trees is called Binary Search Tree Sort. This sorting method involves inserting elements into a BST first, followed by an in-order traversal of the BST.

The working of the Binary Search Tree Sort proceeds in few following steps:

- Insertion: To begin, insert each input array element into a Binary Search Tree. Each element is inserted into the Binary Search Tree into the left subtree if its value is smaller than the present node's value, and into the right subtree if its value is greater. The pseudo-code for the insertion is given below:

TREE-INSERT (*T*, *z*)

y = *NIL*

x = *T.root*

while *x* \neq *NIL*

y = *x*

 if *z.key* < *x.key*

x = *x.left*

 else *x* = *x.right*

z.p = *y*

if *y* == *NIL*

T.root = *z*

elseif *z.key* < *y.key*

y.left = *z*

else *y.right* = *z*

[1]

- In-order Traversal: Once every element has been inserted, go through the BST in order. The left subtree is accessed first in an in-order traversal, followed by the current node and the right subtree. Because of a BST's characteristics, when this traversal is applied to one, the nodes are visited in ascending order. The pseudo-code for the traversal into the tree is given below:

ITERATIVE-TREE SEARCH (x, k)

while $x \neq \text{NIL}$ and $k \neq x.\text{key}$

 if $k < x.\text{key}$

$x = x.\text{left}$

 else $x = x.\text{right}$

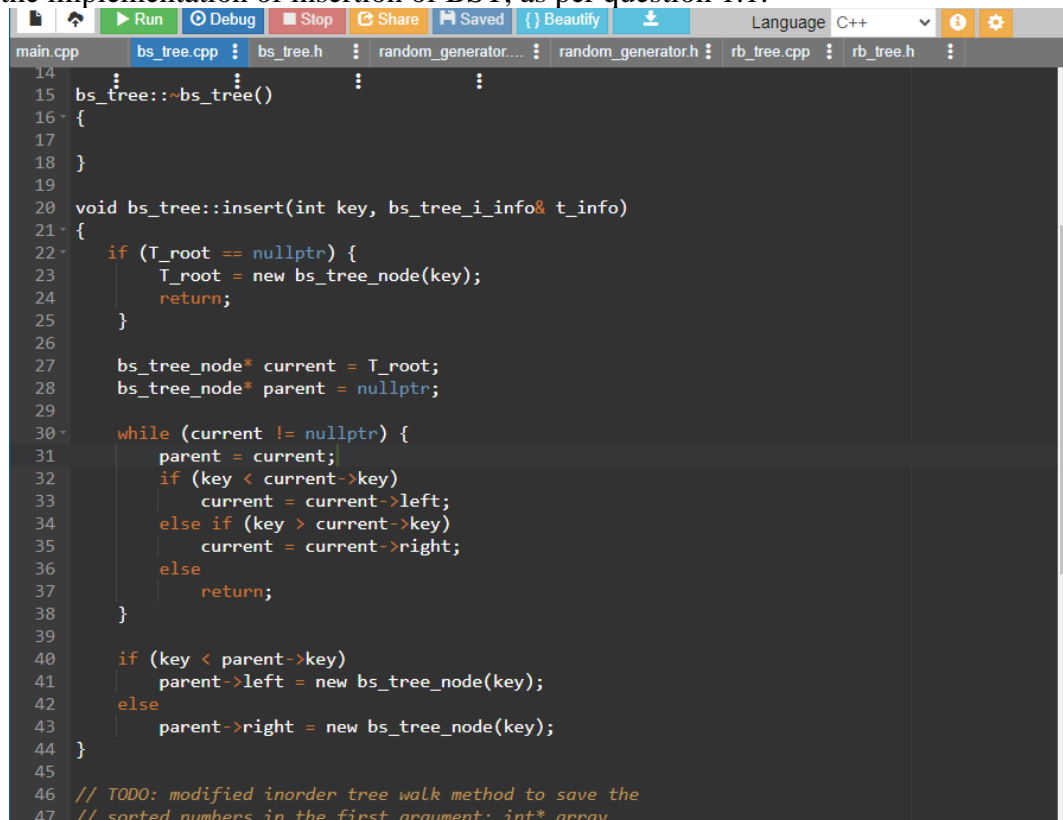
return x

[1]

- Output: Print the value of each visited node during the in-order traversal. The elements are sorted in this order because the nodes are visited in ascending order which is the characteristic of Binary Search Tree.

Implementation:

This is the implementation of insertion of BST, as per question 1.1:



```
14
15 bs_tree::~bs_tree()
16 {
17 }
18
19
20 void bs_tree::insert(int key, bs_tree_i_info& t_info)
21 {
22     if (T_root == nullptr) {
23         T_root = new bs_tree_node(key);
24         return;
25     }
26
27     bs_tree_node* current = T_root;
28     bs_tree_node* parent = nullptr;
29
30     while (current != nullptr) {
31         parent = current;
32         if (key < current->key)
33             current = current->left;
34         else if (key > current->key)
35             current = current->right;
36         else
37             return;
38     }
39
40     if (key < parent->key)
41         parent->left = new bs_tree_node(key);
42     else
43         parent->right = new bs_tree_node(key);
44 }
45
46 // TODO: modified inorder tree walk method to save the
47 // sorted numbers in the first argument: int* array.
```

Fig 1. Insertion to a Binary Search Tree

Explanation:

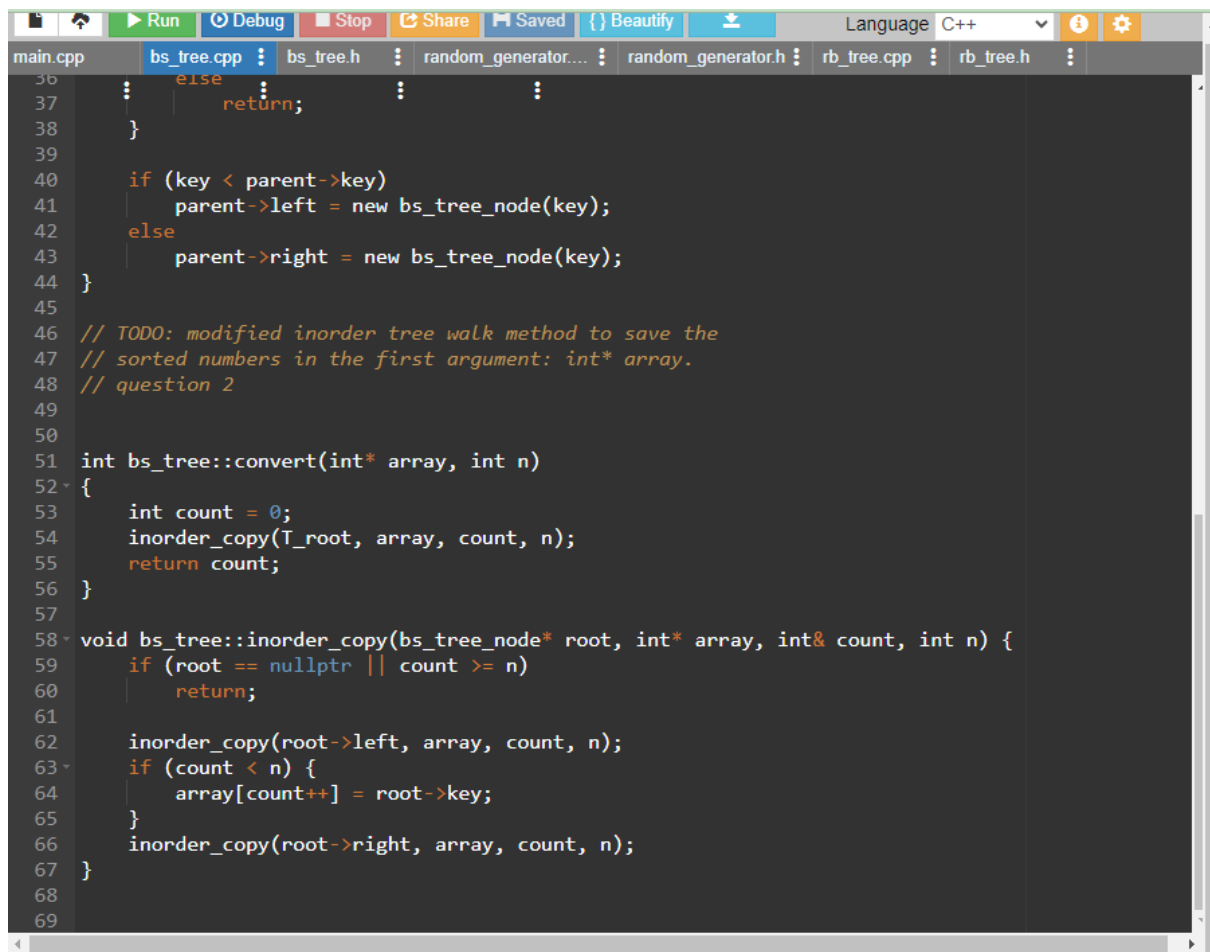
Let me break down the code and explain,

1. The insert function takes two parameters: a key and t_info. The key represents the value to be inserted into the BST.
2. Then diving into the function, check if T_root is empty. If it is true, then the incoming / new node would be the root node. If it is false, then it continues.
3. The function searches downward the tree to determine the ideal location for the new key to be inserted if it is not empty. It begins at the base of the tree and works its way down iteratively until it finds an appropriate position. It keeps track of two pointers while traversing: parent, which indicates the parent of current, and current, which points to the node that is currently being examined.
4. The function checks the key against the key of the current node (current->key) inside the while loop. The node relocates to the left subtree (current = current->left) if the key

is less than the key of the current node. It advances to the right subtree (current = current->right) if the key is larger.

5. The function just returns without taking any further action if the key that needs to be inserted is already in the tree. When the current becomes nullptr, which is an appropriate location for insertion, the new node with the specified key is created and connected to the tree. By comparing the value of the key in the parent node's key, it is decided which way to put the new node—as the parent's left or right child.

Here is the implementation of the INORDER-TREE-WALK algorithm for binary search as per question 1.2:



```
36
37     else
38         return;
39 }
40
41 if (key < parent->key)
42     parent->left = new bs_tree_node(key);
43 else
44     parent->right = new bs_tree_node(key);
45 }
46
47 // TODO: modified inorder tree walk method to save the
48 // sorted numbers in the first argument: int* array.
49 // question 2
50
51 int bs_tree::convert(int* array, int n)
52 {
53     int count = 0;
54     inorder_copy(T_root, array, count, n);
55     return count;
56 }
57
58 void bs_tree::inorder_copy(bs_tree_node* root, int* array, int& count, int n) {
59     if (root == nullptr || count >= n)
60         return;
61
62     inorder_copy(root->left, array, count, n);
63     if (count < n) {
64         array[count++] = root->key;
65     }
66     inorder_copy(root->right, array, count, n);
67 }
68
69
```

Fig 2. In-order tree walk into a Binary Search Tree

Explanation:

Let me break down the code and explain,

1. The first function of bs_tree::convert(int* array, int n), takes an integer array and its size n as the two parameters.
2. The count variable is initialized to zero and it keeps track of the number of elements that are copied into the array. Later, it calls the inorder_copy function, which is the helper function, that helps in traversal. Once it is done, it returns the number of elements that are copied to the array.

- More about, the helper function `inorder_copy`, helps in traversing into the BST through its root and copies the key into the array. The count variable keeps note of the number of elements copied into the array.

The current node's left subtree is traversed by the function recursively (`root -> left`). The current node's key is copied into the array, and the count is increased if the count is less than `n`, which indicates that there is still space in the array. Next, it iteratively navigates through the current node's right subtree (`root->right`). In other words, this procedure traverses the BST in reverse order, adding keys to the array in sorted order until the tree is completely explored or the count reaches `n`, whichever comes first.

Here is the implementation concerning the duplicates, as mentioned in question 1.3:

```

1 #include "bs_tree.h"
2
3 bs_tree::bs_tree()
4 {
5     T_root = nullptr;
6     T_nil = nullptr;
7 }
8
9 bs_tree::~bs_tree()
10 {
11 }
12
13 void bs_tree::insert(int key, bs_tree_i_info& t_info)
14 {
15     if (T_root == nullptr) {
16         T_root = new bs_tree_node(key);
17         return;
18     }
19
20     bs_tree_node* current = T_root;
21     bs_tree_node* parent = nullptr;
22
23     while (current != nullptr) {
24         parent = current;
25         if (key < current->key)
26             current = current->left;
27         else if (key > current->key)
28             current = current->right;
29         else {
30             t_info.i_duplicate++;
31             return;
32         }
33     }
34 }
35

```

Timer (generate): 2ms real 0ms user 0ms sys
Running sort using BS trees:
Timer (sort): 19ms real 20ms user 0ms sys
New size: 49998
Duplicates: 2
The output is sorted!

...Program finished with exit code 0
Press ENTER to exit console.

Fig 3. Modification with respect to duplicates

Explanation:

Let me break down the code and explain,

- `bs_tree::bs_tree()` constructor: Initialization is made, the member pointers `T_root` and `T_nil` to `nullptr` in the constructor. This guarantees that when there is a new instance of `bs_tree` is created, the tree is initially empty.
- `int key, bs_tree_i_info& t_info, void bs_tree::insert()`: As stated in the function signature, it is modified that as insert function to accept `bs_tree_i_info` as a reference parameter. And it is modified the function to handle duplicates differently. An attempt was made earlier to access `dup_counter`, which wasn't declared in the scope at hand. Rather, when a duplicate key is found during insertion, I immediately increased the `i_duplicate` counter of the given `bs_tree_i_info` object.
- `int* array, int n): int bs_tree::convert` : In order to traverse the tree in order and copy the keys into the supplied array, added code that calls the `inorder_copy` helper function

with the necessary arguments. The quantity of elements copied into the array is returned by the function.

- `bs_tree::inorder_copy(root, int* array, int& count, int n),bs_tree_node*`:This helper function copies the keys into the array by recursively traversing the tree in order as required by the convert function. It is changed the function so that it will terminate the traversal when either the current node is nullptr or the count of copied elements reaches n.

RED_BLACK TREE:

Red-black trees are binary search trees that have an additional bit of storage for each node, that is, a node's color, which can be either BLACK or RED. Red-black trees make sure that no simple path from the root to a leaf is longer than twice as long as any other by limiting the node colors on that path. This keeps the tree roughly balanced. The attributes left, right, color, key, and p are now present in every node of the tree. A node's corresponding pointer attribute has the value NIL if either its parent or child does not exist

All instances of NIL are indicated by a single sentinel, denoted as T: nil, in Red-Black Tree (RBT) implementations. Boundary conditions are made simpler by replacing all pointers to NIL in the tree with this sentinel, whose color attribute is always set to BLACK. Although it shares characteristics with usual nodes, its particular values are unimportant and can be changed as needed for operational convenience. Internal nodes are usually the center of attention, and in graphical representations, leaves are frequently left out.

The working of the Red Black Tree Sort proceeds in few following steps:

- Build a RBT: Place every component in a Red-Black Tree from the input collection. The tree is modified as elements are added to preserve its red-black tree characteristics, such as color harmony and hierarchy. The pseudo-code for the build a RBT is given below:

RB-Insert(T,z)

Y = T.nil

X= T.root

while x != T.nil

y = x

if z.key < x.key

x = x.left

else x = x.right

z.p = y

if y == T.nil

T.root = z

elseif z.key < y.key

y.left = z

else y.right = z

z.left = T.nil

z.right = T.nil

z.color = RED

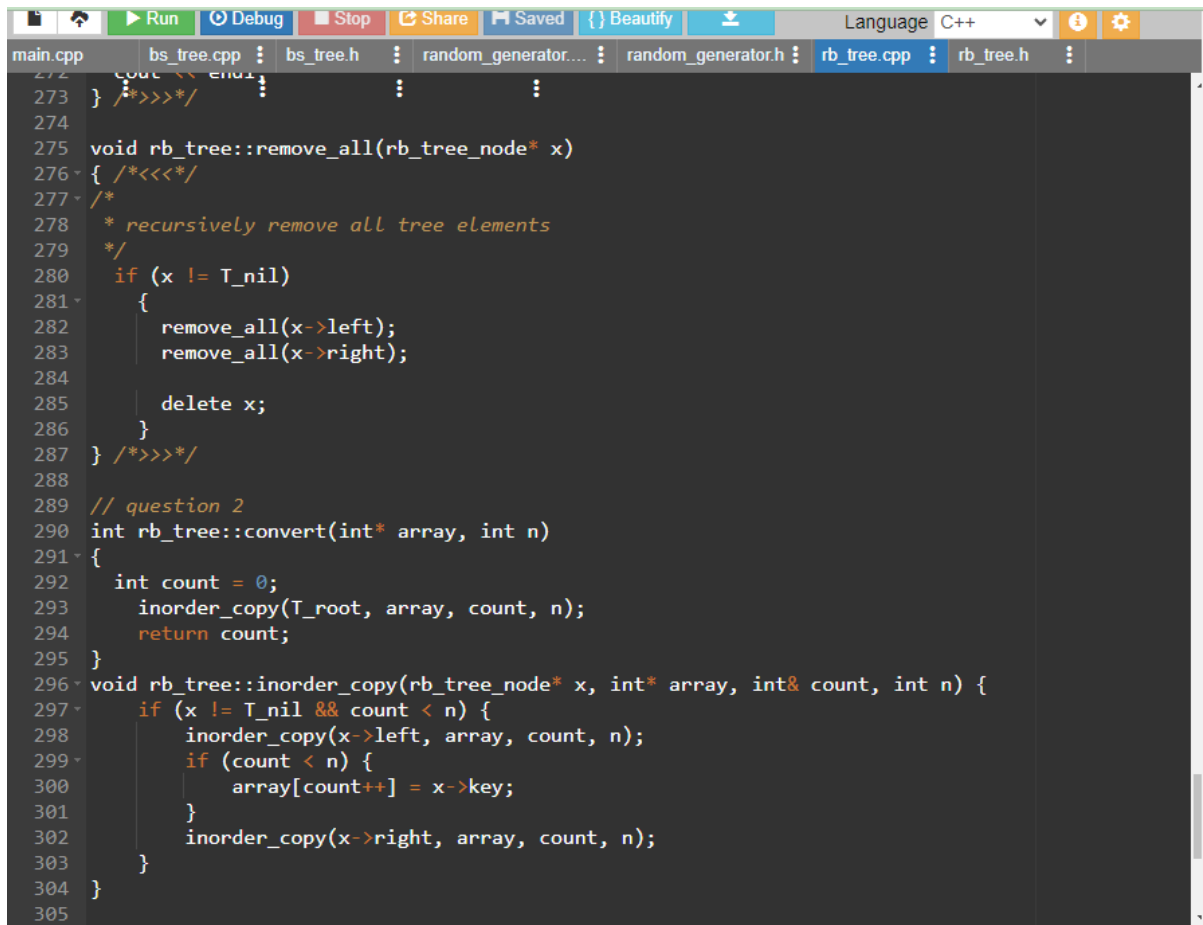
RB-Insert Fixup(T,z)

[1]

- In-order Traversal Proceed to traverse the Red-Black Tree in order. The nodes are visited by in-order traversal in a sorted order. Upon visiting every node, obtain the corresponding element and record it in an output list or array.
- Output: The elements in the output array are sorted orderly following the traversal's conclusion.

Implementation:

This is the implementation of inorder tree walk of RST, as per question 1.2:



```
272 } /*>>>*/
273 } /*>>>*/
274
275 void rb_tree::remove_all(rb_tree_node* x)
276 { /*<<<<*/
277 /*
278  * recursively remove all tree elements
279  */
280 if (x != T_nil)
281 {
282     remove_all(x->left);
283     remove_all(x->right);
284
285     delete x;
286 }
287 } /*>>>*/
288
289 // question 2
290 int rb_tree::convert(int* array, int n)
291 {
292     int count = 0;
293     inorder_copy(T_root, array, count, n);
294     return count;
295 }
296 void rb_tree::inorder_copy(rb_tree_node* x, int* array, int& count, int n) {
297     if (x != T_nil && count < n) {
298         inorder_copy(x->left, array, count, n);
299         if (count < n) {
300             array[count++] = x->key;
301         }
302         inorder_copy(x->right, array, count, n);
303     }
304 }
305
```

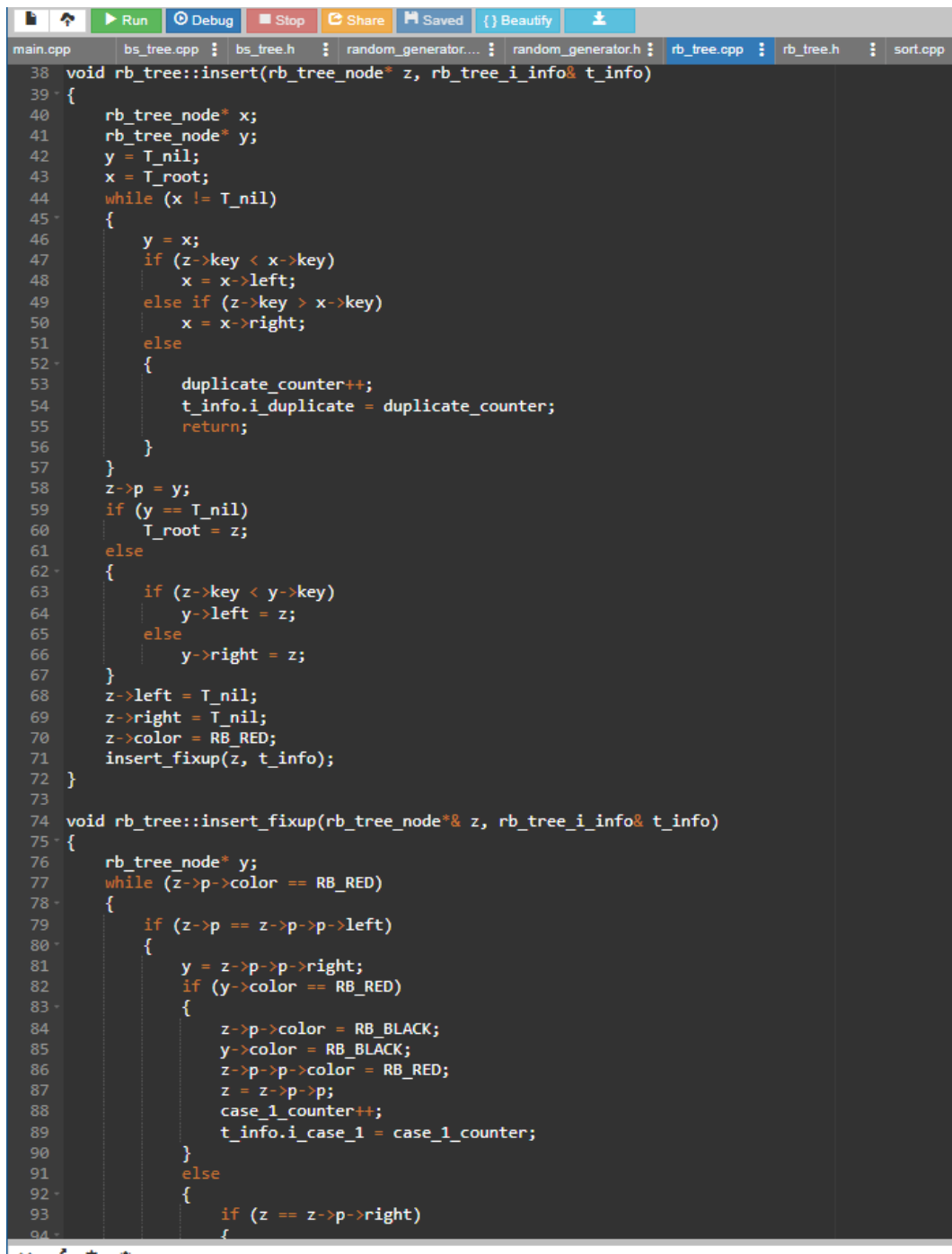
Fig 4. Inorder tree walk on RBT

Explanation:

Let me break down the code and explain,

- To keep track of how many elements are copied into the array, set the initial value of a counter variable set the count to 0.
- Start an in-order traversal of the red-black tree, beginning at node T_root.
- Go through the tree recursively and out of order. Verify that each node is not the sentinel node T_nil and that there is still space in the array (count < n). If all the requirements are satisfied, increment count and copy the current node's key into the array at the designated location.
- When count hits the maximum number of elements indicated by n, or when the current node is the sentinel node T_nil, the traversal should be stopped.
- Give back the total number of items that were copied into the array.

This is the implementation of duplicates with case 1, case & case 3, as per question 1.3:



```
38 void rb_tree::insert(rb_tree_node* z, rb_tree_i_info& t_info)
39 {
40     rb_tree_node* x;
41     rb_tree_node* y;
42     y = T_nil;
43     x = T_root;
44     while (x != T_nil)
45     {
46         y = x;
47         if (z->key < x->key)
48             x = x->left;
49         else if (z->key > x->key)
50             x = x->right;
51         else
52         {
53             duplicate_counter++;
54             t_info.i_duplicate = duplicate_counter;
55             return;
56         }
57     }
58     z->p = y;
59     if (y == T_nil)
60         T_root = z;
61     else
62     {
63         if (z->key < y->key)
64             y->left = z;
65         else
66             y->right = z;
67     }
68     z->left = T_nil;
69     z->right = T_nil;
70     z->color = RB_RED;
71     insert_fixup(z, t_info);
72 }
73
74 void rb_tree::insert_fixup(rb_tree_node*& z, rb_tree_i_info& t_info)
75 {
76     rb_tree_node* y;
77     while (z->p->color == RB_RED)
78     {
79         if (z->p == z->p->p->left)
80         {
81             y = z->p->p->right;
82             if (y->color == RB_RED)
83             {
84                 z->p->color = RB_BLACK;
85                 y->color = RB_BLACK;
86                 z->p->p->color = RB_RED;
87                 z = z->p->p;
88                 case_1_counter++;
89                 t_info.i_case_1 = case_1_counter;
90             }
91             else
92             {
93                 if (z == z->p->right)
94                 {
```

Fig 5. Duplicate counters are being used to know the count of duplicates

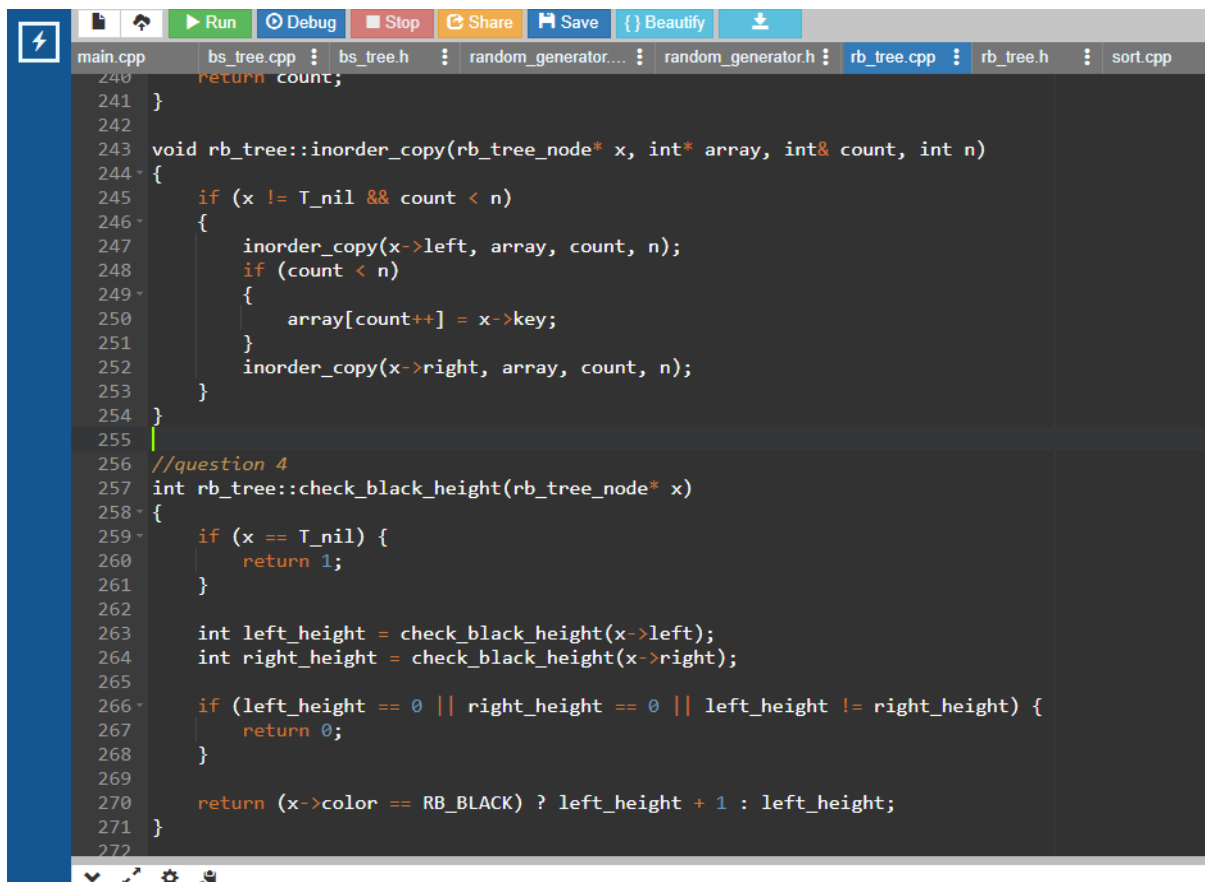
Explanation:

Let me break down the code and explain,

- The function enters a loop that keeps going as long as the newly inserted node z's parent (z->p) is red. Up to the root, this loop fixes violations recursively.
- Verifies whether Z's parent is its grandparent's left child. If this is accurate, then z belongs to its grandparent's left subtree.
- y is given the parent node z->p's sibling.

- Determines if z is the correct child of z->p. If verified, it suggests Situation 2. To fix the infraction, rotates around the parent z to the left. Left rotation increment counters and Case 2 increment counters (case_2_counter), which update the appropriate fields in t_info.
- Case 3 is implied if the condition of Case 2 is not satisfied. To fix the violation, it recolors the grandparent's node (z->p->p) to red and the corresponding parent node (z->p) to black. It then rotates the grandparent to the right. Updating the relevant field in t_info, Case 3's counter (case_3_counter) is increased.
- Makes sure the tree's root is always black after breaking out of the loop, upholding property 2 of a red-black tree.

This is the implementation of the test function of RST, as per question 1.4:



```

240     return count;
241 }
242
243 void rb_tree::inorder_copy(rb_tree_node* x, int* array, int& count, int n)
244 {
245     if (x != T_nil && count < n)
246     {
247         inorder_copy(x->left, array, count, n);
248         if (count < n)
249         {
250             array[count++] = x->key;
251         }
252         inorder_copy(x->right, array, count, n);
253     }
254 }
255
256 //question 4
257 int rb_tree::check_black_height(rb_tree_node* x)
258 {
259     if (x == T_nil) {
260         return 1;
261     }
262
263     int left_height = check_black_height(x->left);
264     int right_height = check_black_height(x->right);
265
266     if (left_height == 0 || right_height == 0 || left_height != right_height) {
267         return 0;
268     }
269
270     return (x->color == RB_BLACK) ? left_height + 1 : left_height;
271 }
272

```

Fig 6. Test function for RBT as to know the validation of property 5

Explanation:

This part is implemented to check whether the property of black height is satisfied or not. Let me break down the code and explain,

- When a pointer to node x in the red-black tree, this function returns an integer that indicates whether or not the black height property is upheld.
- If node x is currently the sentinel node T_nil, it indicates that it is a leaf (NIL), and all leaves are black by red-black tree property 3. As a result, it yields 1, denoting a black leaf.
- To find the black heights of the left and right children of node x, recursively calls the function for each child.

- Checks to see if the black heights of the both subtrees are not equal, or if any of the subtrees have a black height of 0 (signifying a property 5 violation). It returns 0 to indicate that the black height property is broken if any of these conditions are satisfied.
- The function goes back the black height of the current node x if it hasn't returned 0 i.e., informing no violations. It increases the black height by 1 if x is black; if not, it returns the original black height.

TIME COMPLEXITY:

When sorting n elements with an RBT, the total time complexity is $O(n \log n)$, which is the sum of the time complexity of the inorder traversal ($O(n)$) and insertion ($O(n \log n)$).

Measuring the run time performance of

a) **Binary Search Tree Sort:**

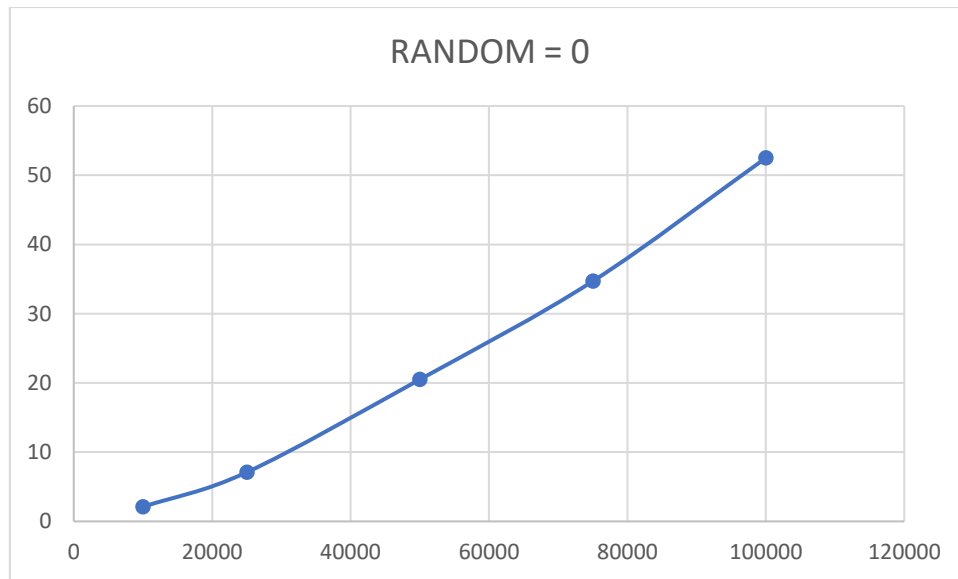
Table 1. Running time of Binary Search Tree Sort

<i>n / sort</i>	<i>RANDOM</i>	<i>ASCENDING</i>	<i>DESCENDING</i>	<i>DUPLICATES</i>
<i>10000</i>	2.1	187.4	170.3	0
<i>25000</i>	6.7	1241.8	1094.6	0.2
<i>50000</i>	24.5	5369.1	4859.6	0.5
<i>75000</i>	25.7	10703	10097.9	1.5
<i>100000</i>	52.5	19142.5	18833.5	2.72

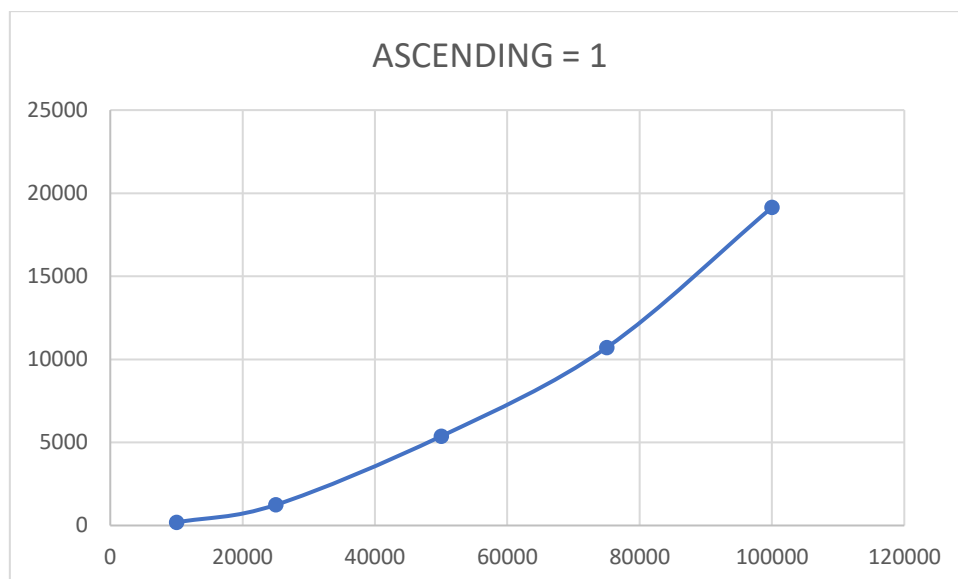
GRAPHS & INFERENCE:

For each case of

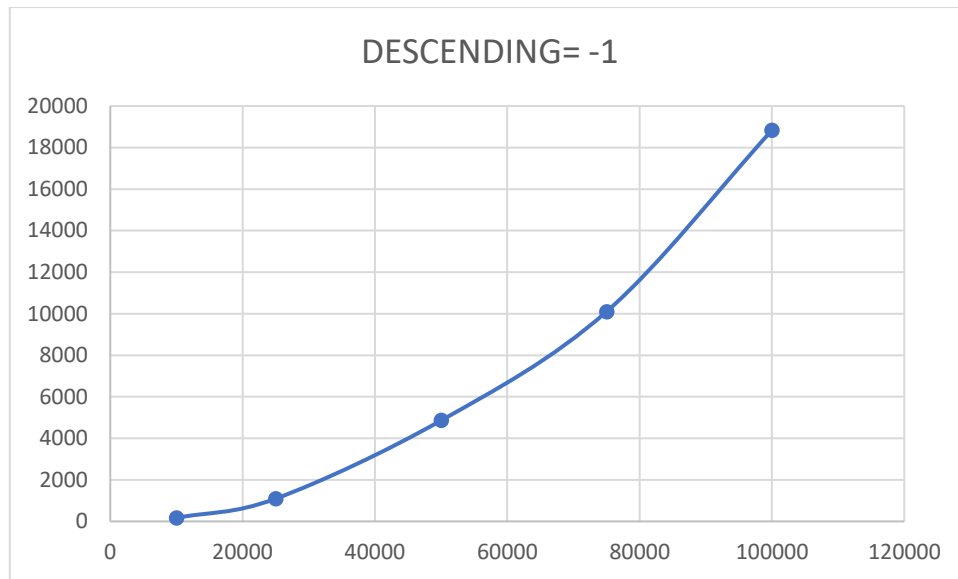
- Random: The binary search tree has a higher chance of being balanced when it receives random input, which involves adding elements to the tree in any order. As a result, each insertion operation has an average time complexity of $O(\log n)$, where n is the number of elements. It still takes $O(n)$ time to traverse the tree in an orderly and turn it into an array. The total time complexity is still, on average, $O(n)$ when input is random.



- b) Ascending: The binary search tree becomes unbalanced and resembles a linked list when elements are added in ascending order. In this instance, the height of the tree equals the number of elements, so each inserting operation takes $O(n)$ time. As a result, the total time complexity increases to $O(n^2)$ with increasing input.



- c) Descending: In the same way, the binary search tree turns unbalanced and takes on the appearance of a linked list when elements are added in descending order. $O(n)$ time is still required for each insertion operation. Even so, the inorder traversal requires $O(n)$ time. As a result, the total time complexity also becomes $O(n^2)$ with descending input.



The duplicates are tabulated above. The conclusion can be made that the number of duplicates increases as there is the increase in the value of n. For larger values of n, which refer to a larger data set.

b) Red Black Tree Sort:

GRAPHS & INFERENCE:

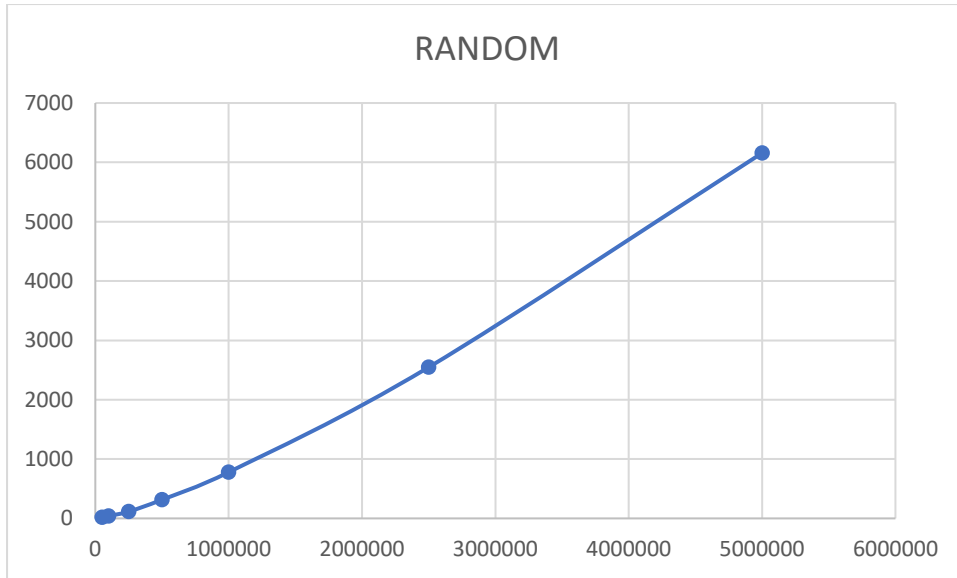
For each case of,

A) RANDOM Case:

Table 2. Random case tabulation for running time and counters

	<i>Running time</i>	<i>DUPLICATES</i>	<i>RIGHT ROTATE</i>	<i>LEFT ROTATE</i>	<i>CASE 3</i>	<i>CASE 2</i>	<i>CASE 1</i>
50000	18.7	0.4	14650.6	14522.66	19472.3	9701	25683.3
100000	41.1	3	29118.3	29056.33	38781.3	19393.3	51309
250000	114.5	13.57142857	72959	72816	97190	48585	128252
500000	315.6	60.4	145266	145108	193737	96637	256542
1000000	779.2	228	291231	290997	388171	194057	513613
2500000	2547.6	1445.6	728994	728647	971585	486056	1282977
5000000	6157.2	5724.8	1456579	1456109	1941327	971361	2564255

When dealing with random data, the average time complexity of search, insertion, and deletion operations is still $O(n \log n)$. This is because logarithmic time complexity is guaranteed for these operations. After all, a balanced tree is typically produced by random distribution.

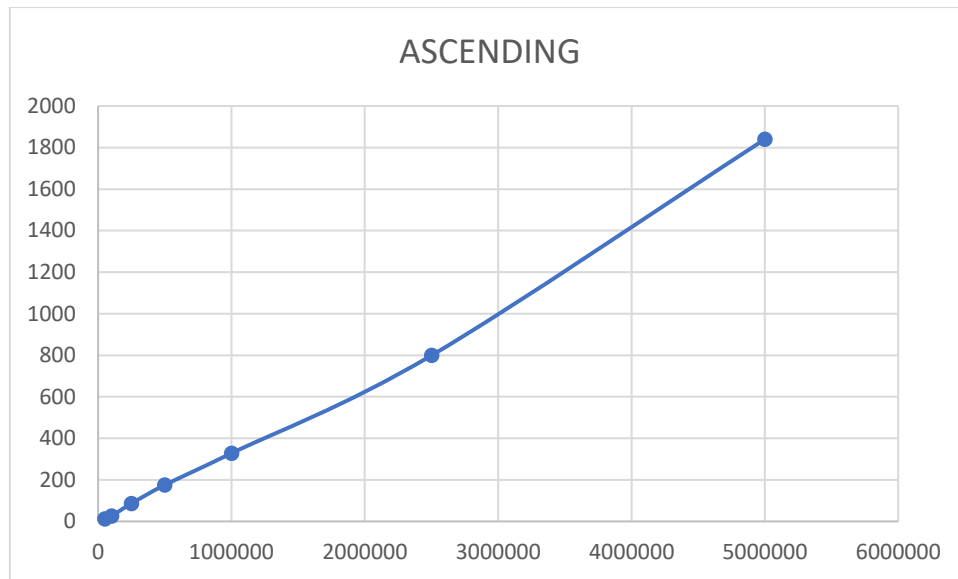


B) ASCENDING Case:

Table 3. Ascending case tabulation for running time and counters

	<i>Running time</i>	<i>DUPLICATES</i>	<i>RIGHT ROTATE</i>	<i>LEFT ROTATE</i>	<i>CASE 3</i>	<i>CASE 2</i>	<i>CASE 1</i>
50000	12.2	0	0	49971	49971	0	49966
100000	25.9	0	0	99969	99969	0	99964
250000	86.2	0	0	249967	249967	0	249961
500000	175.2	0	0	499965	499965	0	499959
1000000	328.1	0	0	999965	999965	0	999957
2500000	799.1	0	0	2499960	2499960	0	2499952
5000000	1839.8	0	0	4999958	4999958	0	4999950

The tree retains its balanced structure, and the running time complexity of RBT sorting remains $O(n \log n)$, regardless of the order (random, ascending, or descending).

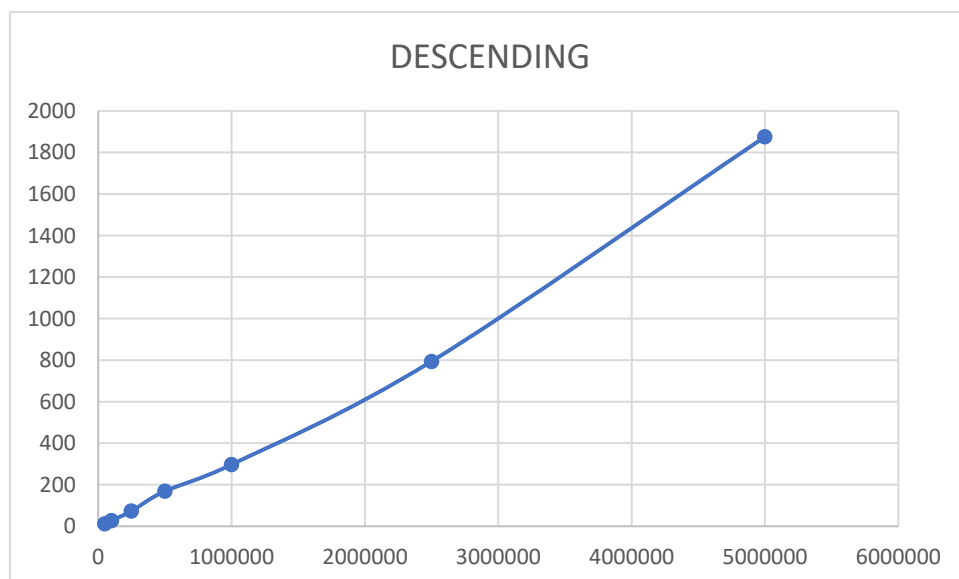


C) DESCENDING Case:

Table 4. Ascending case tabulation for running time and counters

	<i>Running time</i>	<i>DUPLICATES</i>	<i>RIGHT ROTATE</i>	<i>LEFT ROTATE</i>	<i>CASE 3</i>	<i>CASE 2</i>	<i>CASE 1</i>
50000	12.6	0	49971	0	49971	0	49966
100000	27.4	0	99969	0	99969	0	99964
250000	73.6	0	249967	0	249967	0	249961
500000	169.4	0	499965	0	499965	0	499959
1000000	297.1	0	999963	0	999963	0	999957
2500000	794.2	0	2499960	0	2499960	0	2499952
5000000	1875.3	0	4999958	0	4999958	0	4999950

The tree retains its balanced structure, and the running time complexity of RBT sorting remains $O(n \log n)$, regardless of the order (random, ascending, or descending).



CONCLUSION:

Ultimately, BST sort is a successful sorting technique, with a best-case time complexity of $O(n \log n)$. But its performance can drop to $O(n^2)$ in the worst-case scenario of sorted input data. As the value of the data nodes rises, so does the count of duplicates.

It is a dependable option for many applications because, in comparison to RBT sort, it provides effective sorting capabilities with $O(n \log n)$ time complexity in both best and worst-case scenarios. Moreover, even with sorted or partially sorted input data, RBT's self-balancing property guarantees optimal performance. The counters for the insertion cases (cases 1, 2, and 3) as well as the left and right rotation counters demonstrate how the structure of the Red-Black Tree varies as n increases.

REFERENCES:

- [1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009) *Introduction to Algorithms* (3rd ed.). The MIT Press.
- [2] Sedgewick, R., & Wayne, K. (2011). *Algorithms*(4th Edition). Addison-Wesley Professional.