Comparing the Performance of Skip List, Red Black Trees & Java's Tree Set

Group Members –Shariq Ali, Abhigyan Sinha, Navanil Sengupta, Enakshi Mandal

Problem definition:

In this Project we will be studying the execution time of Skip List, Red Black Trees and Java's Tree Set for add, remove and contain operations over large sizes of input such as 4M, 16M, 64M, etc. We will observe their respective execution time and figure out which data structure is beneficial to use with respect to the input size and the operation at hand.

Introduction:

To understand about the three data structures and their underlying concepts, let us go through them in a perfunctory manner.

1) Skip – List : Skip List was invented to benefit from the properties of both a Linked List and Binary Search where one did not have to traverse the whole List in a linear manner to insert, remove or check for an element. To reduce the operation time from O(n) to O(log n) average time, skip list was introduced which still had all the properties of a singly Linked List. So, in a skip List basically we have multiple level of Singly Linked Lists starting from L_0------ L_n, where L0 is the Linked List containing all the elements in the original Linked List. Each L_{i+1} after L_0 is a subset of L_0 . As the height of the Skip List increases the Linked Lists on the higher levels tend to be sparser than the ones at the lower level. Now, in order to search an element, skip list uses the property of Binary Search and if the value of next node is greater than the element to be searched or inserted it gets into the lower level and again follows the same procedure, where it checks the condition in the subsequent levels until it finds a point where it can either insert the element or it finds the element. It uses the same procedure for searching an element and to remove it.

2) Red Black Trees – Red Black trees are an advanced form of Binary Search Trees, where the height of RBL is always O(log n). This is to avoid the fact that a skewed binary tree can make the time complexity to O(n) and therefore, RBL ensures that the height of the Binary Tree is always O(log n). This is because after every insertion rotations and re-coloring are performed to ensure that the properties of a RBL are satisfied and the tree is consistent with them. Following are the properties of a RBL :

   a) Every node has a color either red or black.
   b) Root of tree is always black.
   c) There are no two adjacent red nodes (A red node cannot have a red parent or red child).
   d) Every path from a node (including root) to any of its descendant NULL node has the same number of black nodes.

   RBL are preferred over AVL when we have frequent insertion and deletion operations and AVL is preferred when searching is the main motive.

Algorithms:

1) Skip List : Below is the algorithm for inserting an element(elem) in a Skip List

```
define insert(elem, root, height, level):
  if right of root < elem:
    -- If right isn't "overshot" (i.e. we are going to long), we go right.
    return insert(elem, right of root, height, level)
  else:
    if level = 0:
      -- We're at bottom level and the right node is overshot, hence
      -- we've reached our goal, so we insert the node in between root
      -- and the node next to root.
      old ← right of root
      right of root ← elem
      right of elem ← old
    else:
      if level ≤ height:
        -- Our level is below the height, hence we need to insert a link before we go on.
        old ← right of root
        right of root ← elem
        right of elem ← old
       -- Go a level down.
      return insert(elem, below root, height, level - 1)
```

Below is the algorithm for searching an element in a Skip List :

```
define search(skip_list, needle):
  -- Initialize to the first node at the highest level.
  level ← max_level
  current_node ← root of skip_list

  loop:
    -- Go right until we overshoot.
    while level'th shortcut of current_node < needle:
      current_node ← level'th shortcut of current_node

    if level = 0:
      -- We hit our target.
      return current_node
    else:
```

-- Decrement the level.
level ← level – 1

2) **RBL – Tree insertion Algorithm :**

```
Node* Insert(Node* root, Node* n) {
 // Insert new Node into the current tree.
 InsertRecurse(root, n);
 // Repair the tree in case any of the red-black properties have been violated.
 InsertRepairTree(n);
 // Find the new root to return.
 root = n;
 while (GetParent(root) != nullptr)
{
  root = GetParent(root);
}
 return root;

}

void InsertRecurse(Node* root, Node* n) {
 // Recursively descend the tree until a leaf is found.
 if (root != nullptr)
 {
  if (n->key < root->key) {
   if (root->left != nullptr) {
    InsertRecurse(root->left, n);
    return;
    }
else {
    root->left = n;
    }
  }
else {
 // n->key >= root->key
   if (root->right != nullptr)
{
    InsertRecurse(root->right, n);
    return;
 }
else {
    root->right = n;
    }
 }
```

```
}
```

RBL Tree Removal Algorithm

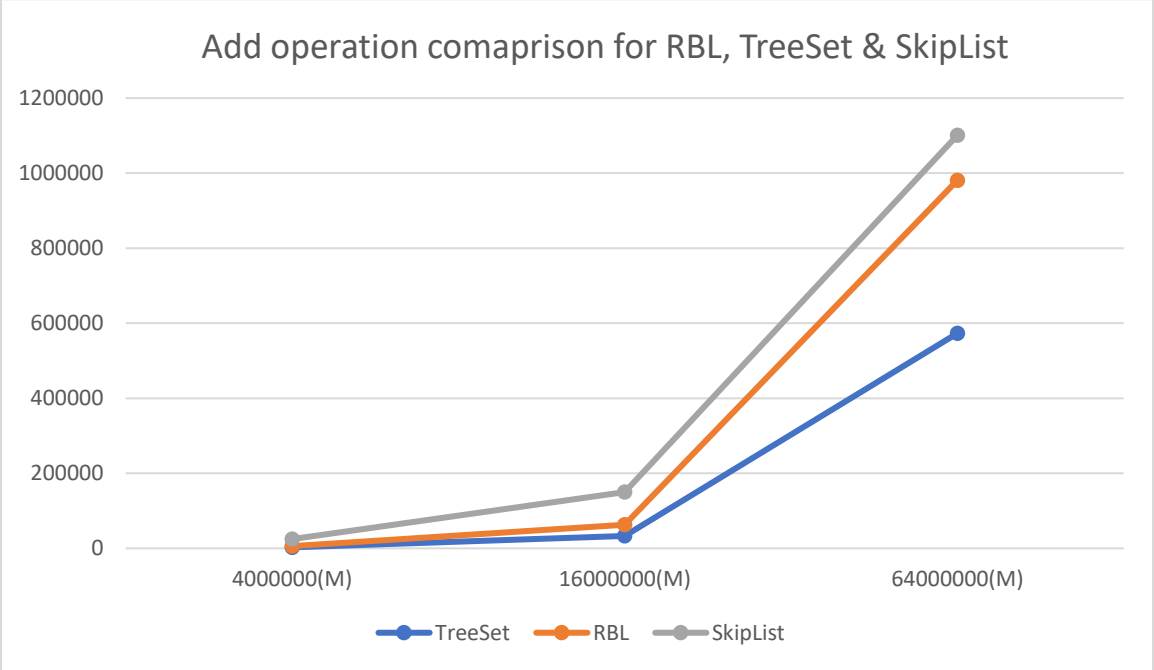
```
void ReplaceNode(Node* n, Node* child)
{
child->parent = n->parent;
 if (n == n->parent->left) {
   n->parent->left = child;
  }
else {
   n->parent->right = child;
  }
}

void DeleteOneChild(Node* n) {
  // Precondition: n has at most one non-leaf child.
  Node* child = (n->right == nullptr) ? n->left : n->right;
  assert(child);

  ReplaceNode(n, child);
  if (n->color == BLACK) {
    if (child->color == RED) {
      child->color = BLACK;
    } else {
      DeleteCase1(child);
    }
  }
  free(n);
}
```

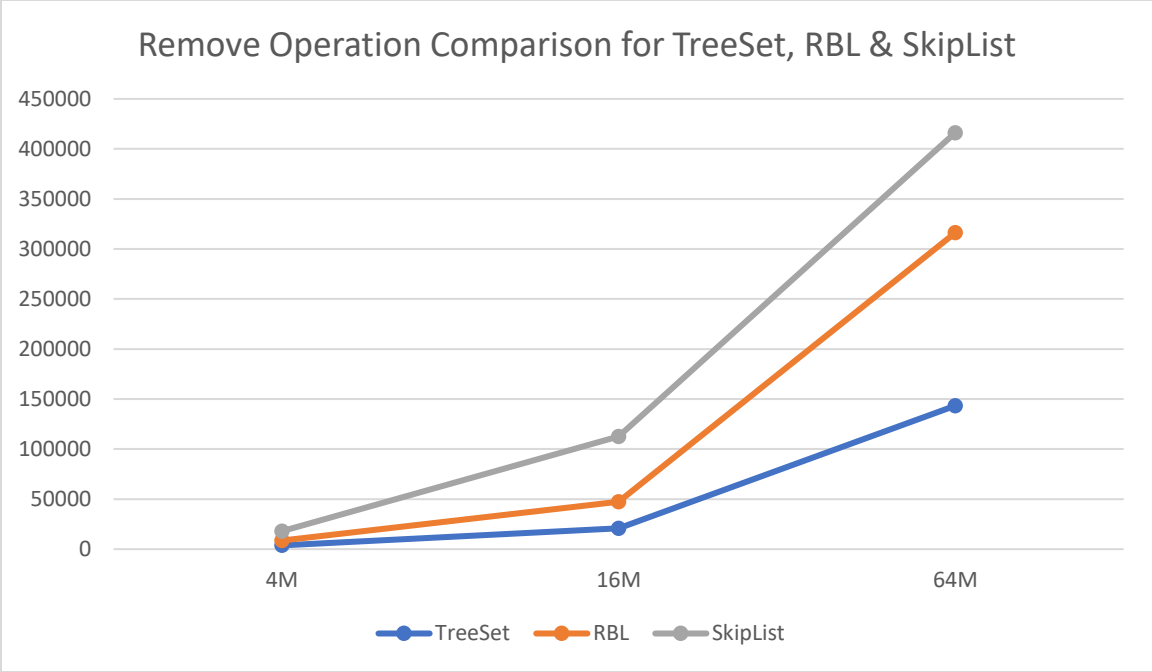
Comparison:

Now let us compare the execution time for Add, Find and Remove operations.



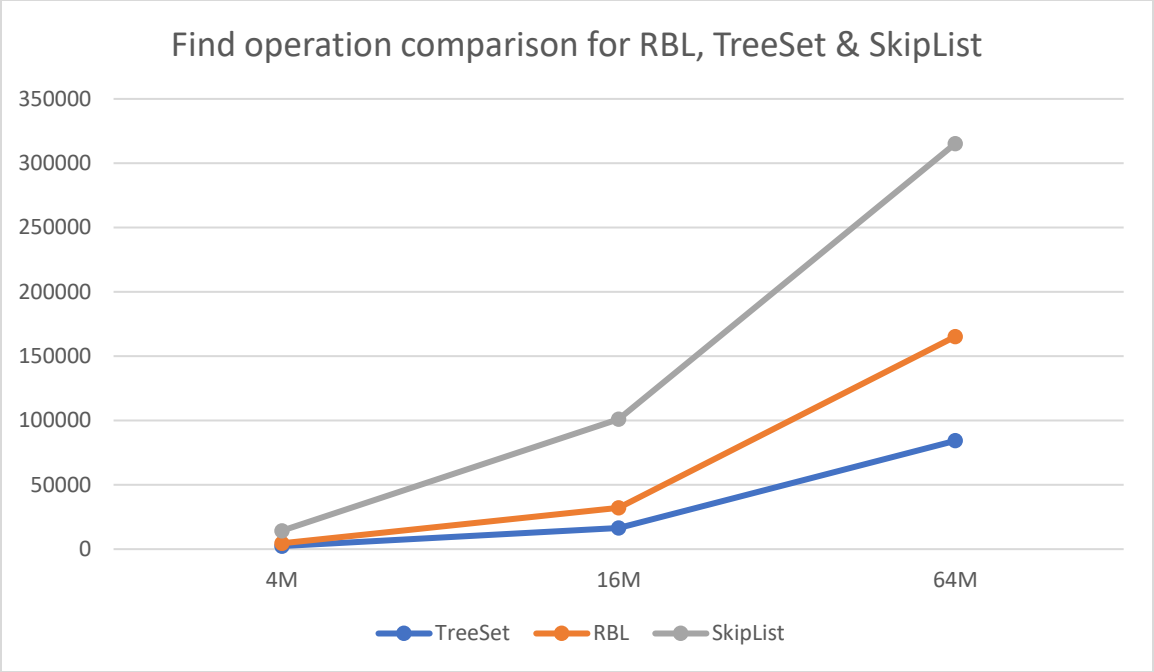
Execution Time for Add Operation for respective data structures.

Input Size	TreeSet	RBL	SkipList
4 Million	2727	3188	18747
16 Million	32940	29600	87423
64 Million	573123	407817	Memory Out
256 Million	Memory Out	Memory Out	Memory Out



Execution Time for Remove Operation for respective data structures.

Input Size	**TreeSet**	**RBL**	**SkipList**
4 Million	3758	4850	9323
16 Million	20944	26364	65297
64 Million	143333	172907	Memory Out
256 Million	Memory Out	Memory Out	Memory Out



Execution Time for Find Operation for respective data structures.

Input Size	**TreeSet**	**RBL**	**SkipList**
4 Million	2370	2171	9776
16 Million	16382	15671	69024
64 Million	84223	81032	Memory Out
256 Million	Memory Out	Memory Out	Memory Out