Data structures for sets - Union, Intersection, Difference with Applications

By,
Tarun Mohandas (2018201008)
Shreyas H N (2018201059)

GIT Repository Link : https://github.com/tarun0409/set-operations.git

1. Abstract

A set is a collection of distinct objects of a single type and are considered as objects by their own right. Sets are fundamental way of representing a collection of data that allows the user to perform various operations on the collection like union, intersection and difference. There is a need to optimize these operations especially for those applications where these operations are performed in bulk regularly. This project aims at:

1. Providing efficient data structures for representing set and performing fundamental set operations
2. Comparison of the data structure used with other contemporary data structures and visualizing the differences
3. Analysis of performance and justification for our choice of data structure
4. Using sets in a real application and understand how it solves some complex problems in that application

2. Data Structure requirements for Set

Requirements for data structure for set from a computational perspective:

1. The data stored should be of uniform type (insertion of any other type should not be allowed)
2. The data should be ordered (This will help in performing various set operations in an efficient manner).
3. Insertion of duplicate data should not be allowed.
4. The data structure should support various set operations like union, intersection and difference and should be able to perform these operations effectively.
5. The API provided for data structure should have maximum abstraction (the users should be left out of the nitty gritty details of how it is working in backend)

3. Abstract Data Type (ADT) for Set

A set can be represented as an abstract data type with its behaviour defined by a set of values and set of operations. This behaviour is also sometimes called API. We can define the API for set as follows:

Let “Set” be the ADT under consideration and let “T” be the type of elements in the set. In general a set containing elements of type T can be represented by:

Set<T>

If we want to represent all the elements in a set, we can do so by listing them (comma separated) in curly braces such as follows:

Set<T> : {T x1, T x2, T x3, ...} such that xi is an element in Set

The set of operations on our ADT Set can be defined as follows:

1. Set<T> **insert(T x)** : If a value of type T is provided, this operation should insert that element into the set such that we can maintain the order of the elements intended. This operation should also not allow duplicate elements to be added.
2. <size> **get_size()** : This operation will return the cardinality (ie. the number of elements) of th set.
3. <boolean> **is_empty()** : This operation will return “true” if the set is an empty set. Otherwise it will return false.
4. T **find(T x)** : If a value of type T is provided, this operation should check if that element exists in the set and return the element if it does exist. The condition when the element does not exist should be handled during implementation as it differs according to the way set is being used in the application.
5. Set<T> **remove(T x)** : This is a compound operation. It involves first finding the element and then removing it from the set. If an element of type T is provided, this operation would first call the find(T x) and then if found remove the element from the set. Once again, according to the application the implementation should handle the case when the element is not found.
6. T **find_kth_element()** : This operation should find the kth element in the set when its elements are listed in the order intended by the user. Usually the order

will be ascending order for integers and lexicographical order for strings. So this operation will reduce to “find the kth smallest element in the set.”

7. Set<T> **union(Set<T> s)** : This operation will take as an input another set “s” and perform a set union on the original set. This means, the result will be another set, that contains the elements of both of these sets.
8. Set<T> **intersection(Set<T> s)** : This operation will take as an input another set “s” and perform a set intersection on the original set. This means the result will be another set, that contains the elements that are only common between these two sets.
9. Set<T> **difference(Set<T> s)** : This operation will take as input another set “s” and perform a set difference on the original set. This means the result will be another set, the contains only elements that are present in original set and not present in set “s”.

4. Data Structure for Set

To meet the requirements of the Set mentioned by us earlier, a binary search tree will be a good choice. This decision is justified for the following reasons:

1. BST enforces its elements to be of same type.
2. BST maintains an order among the elements, which will be helpful while performing various operations in efficient manner.
3. Insert and find operations are easy to implement in search trees because they inherently follow order among elements.
4. Typically inserting duplicate data in BST is not done, although we can change the code to make it do so.

Ordinary Binary Search Trees suffer from a major disadvantage : if the height of the tree is not balanced, almost all operations take **O(n) time** complexity if n is the number of nodes in the tree. In such cases we would rather be justified in using an array instead of BST merely for its simplicity. But we can improve upon the time complexity of the operations by using a balanced binary search trees.

Balanced Binary Search trees are height balanced trees in which the height of the entire tree does not go beyond **O(logn)** if n is the number of nodes in the tree. For analysing this we define “balance factor” of a node to be the difference between height of left subtree of a node and right subtree of the node. We call a tree “balanced” if the balance

factors of all its nodes does not exceed 1. Although there are various forms of balanced binary search trees, we have used AVL Tree in our set implementation.

4.1. AVL Tree

AVL Trees are self balancing Binary Search Trees (BST) where the balance factor of every node in the tree does not exceed 1. By the term “self balancing”, we mean that the tree is designed in such a way that, no external operation is needed to balance the tree and it does some internal operations to self balance itself whenever a change occurs in its structure (insertion or deletion of an element).

4.2. AVL Tree operations

4.2.1. Insertion into AVL Tree

AVL Tree insertion is same as inserting into binary search tree, the only difference being upon insertion, imbalance needs to checked and accordingly rotations need to be performed as mentioned above to make the tree balanced.

1. Start with current_node as root and new_node as node to be inserted
2. Check if value of new_node is less than value of current_node
    a. If true then check if current_node has left sub tree
        i. If true then repeat step 1 with left_node as root of left subtree
        ii. If false then insert new_node into left subtree of current_node.
    b. If false then insert new_node into left_subtree of current_node
3. If value of new_node is greater than value of current_node
    a. Check if current_node has right subtree
        i. If true then repeat step 1 with right_node as root of right subtree
        ii. If false then insert new_node into right subtree of current_node
    b. If false then insert new_node into right_subtree of current_node
4. If value of new_node is equal to value of current_node, then return error.
5. Check for imbalance on the ancestor
6. If imbalanced perform the corresponding rotations and make the resultant tree balanced

4.2.2. Finding value in AVL Tree / Balanced Binary Tree

1. Start with root as current_node and val as value to be searched

2. If val is equal to value of current_node, return value
3. If val is lesser than value of current_node, check if current_node has left subtree
    a. If true, then repeat from step 2 with current_node as root of left subtree
    b. If false, then return that element is not found / error.
4. If val is greater than value of current_node, check if current_node has right subtree
    a. If true, then repeat from step 2 with current_node as root of right subtree
    b. If false, then return that element is not found / error.

4.2.3. Deletion from AVL Tree

Deletion from AVL Tree is same as deletion from BST, except that after deletion, imbalance needs to be checked and corresponding rotations need to be made to make the tree balanced. Sometimes the imbalance can propagate throughout the parent nodes until root node reducing the height of entire tree by one.

1. Find the value in AVL Tree using the procedure described above
    a. If value is not present, return error
2. Find the inorder predecessor of the node to be deleted
3. Swap the values of current node with its inorder predecessor
4. Delete inorder predecessor node
5. Continue the following steps until root node is reached
    a. Check for imbalance in parent
    b. If imbalanced, the perform corresponding rotations to make it balanced

5. Set Operations on AVL Tree

5.1. Building a Balanced Binary Tree from Sorted Array

This is a helper function for all the set operations, that will convert a sorted function to a balanced binary search tree which can be used to represent an AVL Tree, which in-turn can be used to represent a set.

1. Start with an array : sorted_array that is sorted
2. Find the middle element of the array and make it the current root of the sub tree
3. If the sorted_array has a left half, repeat from step 1 with left half of sorted_array as new sorted_array for the left subtree of current subtree
4. If sorted_array has a right half, repeat from step 1 with right half of sorted_array as new sorted_array for the right subtree of the current subtree

5. Return the root of the new tree

5.2. Union

1. Begin with two sets (represented by AVL Trees) as set 1 and set 2
2. Do inorder traversal of set 1 and get the elements in sorted order into vector1
3. Do inorder traversal of set 2 and get the elements in sorted order into vector2
4. Merge the two vectors into one with the following algorithm and store into res_vector:
    a. Start with i pointing to first element of vector1 and j pointing to first element of vector2. Let res_vector[last_element] represent the last element that was added to res_vector
    b. Repeat the following until i or j crosses its respective array size limits:
        i. If res_vector[last_element] equals vector1[i] : increment i
        ii. If res_vector[last_element] equals vector2[j] : increment j
        iii. Check if vector1[i] <= vector2[j]
            1. If true, add vector1[i] to res_vector. Increment i
            2. If false, check if vector2[j] < vector1[i]
                a. If true, add vector2[j] to res_vector, increment j
    c. If i is within vector1 limits, add remaining elements of vector1 to res_vector
    d. If j is within vector2 limits, add remaining elements of vector2 to res_vector
5. Build a balanced binary tree using the above mentioned recursive procedure using res_vector as sorted_array
6. Return the resultant tree

5.3. Intersection

1. Begin with two sets (represented as AVL Trees) as set 1 and set 2
2. Do inorder traversal of set 1 and get the elements into sorted order into vector1
3. Do inorder traversal of set 2 and get the elements into sorted order into vector2
4. Find the common elements of two vectors using the following algorithm and store into res_vector:
    a. Start with i pointing to first element of vector1 and j pointing to first element of vector2
    b. Repeat the following until i or j crosses its respective vector size limits
    c. Check if vector1[i] < vector2[j]
        i. If true, increment i
        ii. If false, check if vector1[i] > vector2[j]

            1. If true, increment j
            2. If false, add vector1[i] into res_vector. Increment i and j
5. Build a balanced binary tree using above mentioned recursive procedure using res_vector and sorted_array
6. Return the resultant tree

5.4. Difference

1. Begin with two sets (represented as AVL Trees) as set 1 and set 2
2. Do inorder traversal of set 1 and get the elements into sorted order into vector1
3. Do inorder traversal of set 2 and get the elements into sorted order into vector2
4. Find the common elements of two vectors using the following algorithm and store into res_vector:
    a. Start with i pointing to first element of vector1 and j pointing to first element of vector2
    b. Repeat the following until i or j crosses its respective vector size limits
    c. Check if vector1[i] < vector2[j]
        i. If true, increment i. Add vector1[i] to res_vector
        ii. If false, check if vector1[i] > vector2[j]
            1. If true, increment j
            2. If false, increment i and j
5. Build a balanced binary tree using the above mentioned recursive procedure using res_vector and sorted_array
6. Return the resultant tree

6. Time complexity analysis

6.1. Insertion:

Insertion can be split into three major actions that needs attention

1. Search for the right spot
2. Insert the element
3. Balancing the tree

Since it is a balanced tree, the height of the tree is O(logn). Each node we are checking, we are either moving to the left of the node or right of the node (ie. only moving down the height of the tree). Therefore searching for the right spot takes only O(logn) time. Inserting just involves placing the node on one of its child spots. So it takes only O(1) time. For insertion once we resolve the balance one time, we need to check if any other

node is balanced. And the rotations are only a few constant time operations and so for insertion, balancing the tree takes O(1) operations. Let T(n) be the time complexity for insertion.

T(n) = O(logn) + O(1) + O(1) = O(logn)

6.2. Deletion:

Deletion can be split into four major actions that need attention

1. Searching for the element
2. Search for inorder predecessor and swapping
3. Removing the element
4. Balancing the tree

Since it is a balanced tree and we only search along the height of the tree, searching for an element is O(logn) time. Searching for inorder predecessor is also O(logn) time. We always remove from leaf, so removing the element is O(1) time. Balancing takes constant amount of operations. But in deletion, the imbalance can propagate upto the root, making us check along the height of the tree. So balancing the tree takes O(logn) time. Let T(n) be the time complexity for deletion.

T(n) = O(logn) + O(logn) + O(1) + O(logn) = O(logn)

6.3. Union

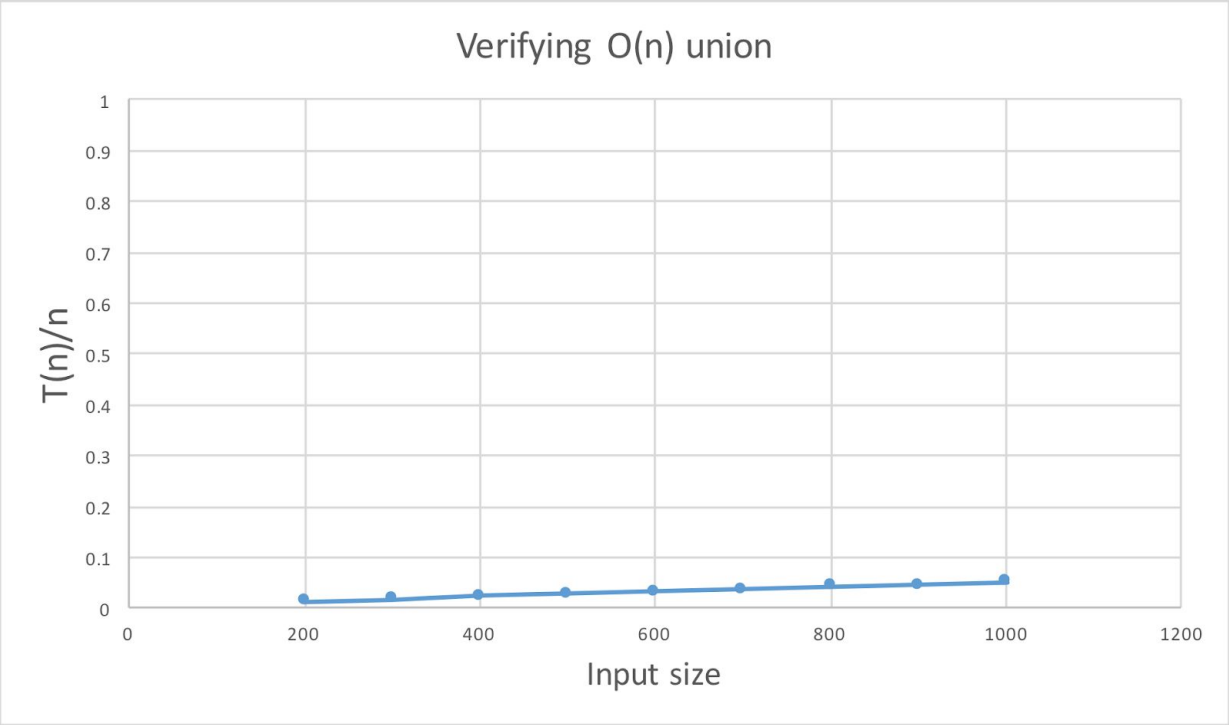
Union can be split into three major actions that need attention

1. Inorder traversals of two binary trees
2. Merging of two sorted arrays
3. Constructing BST

Inorder traversal has to visit all nodes no matter what. So it is O(n+m) if n is number of nodes in set 1 and m is number of nodes in set 2. From the merge sort procedure, we know that merging two sorted arrays is O(n+m). While constructing a BST, each node is visited upon insertion and only once. Hence it is also O(n+m). Let T(n+m) be the time complexity for union

T(n+m) = O(n+m) + O(n+m) + O(n+m) = O(n+m)

Therefore we perform union operation in linear time.



To verify that the union function written takes O(n), the time taken by the function for varying sizes of input was divided by n. It was seen that the resulting number was almost constant. The same has been plotted in the above graph.

6.4. Intersection

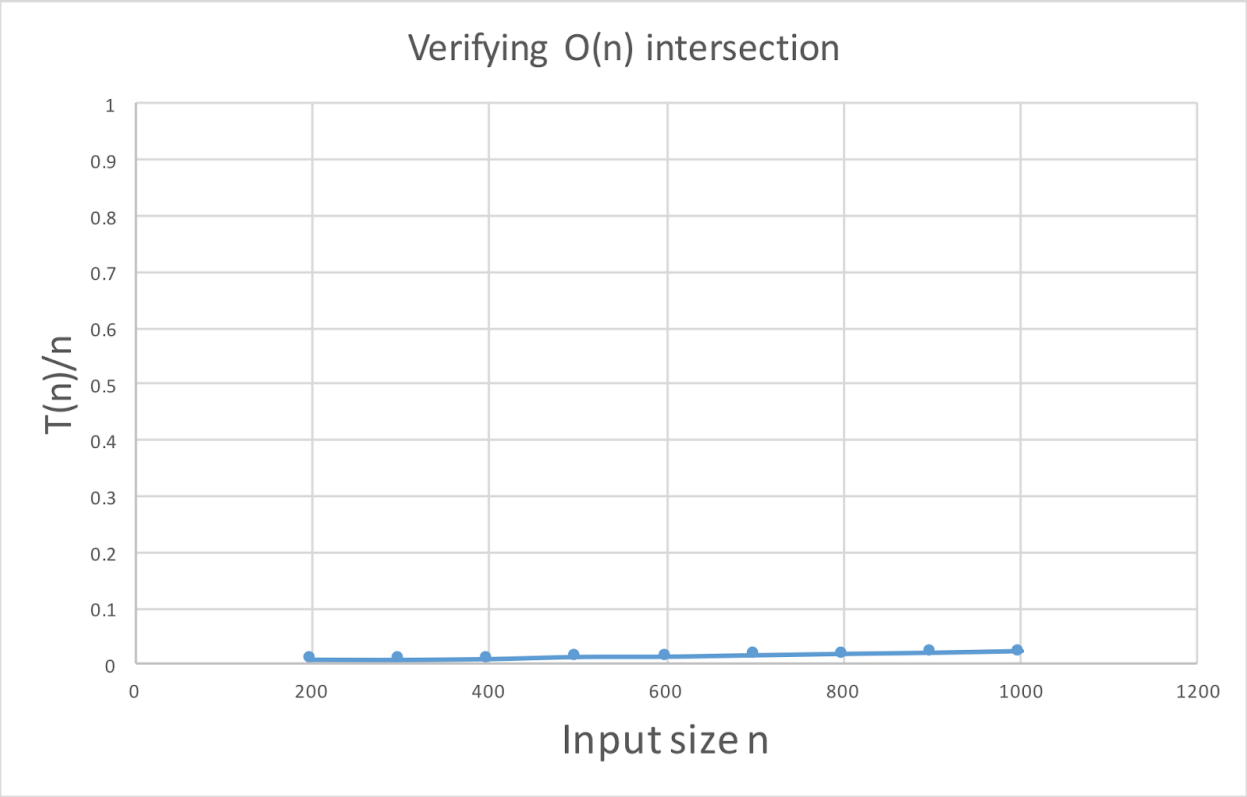
Intersection can be split into three major actions that need attention

1. Inorder traversals of two binary trees
2. Finding common elements of first array with second
3. Constructing BST

If n is number of inputs in set 1 and m is number of inputs in set 2, inorder traversals are going to take O(n+m). At the most while finding common elements, we only traverse the entire first array containing n elements. Therefore finding common elements takes O(n) time. We construct a BST from at most n elements since we only took elements from first set. Therefore constructing BST is also going take only O(n) time. Let T(n+m) be the time complexity for intersection.

T(n+m) = O(n+m) + O(n) + O(n) = O(n+m)

Therefore we perform intersection operation in linear time.



From the above graph it can be been that T(n)/n is a constant for varying sizes of input.

6.5. Difference

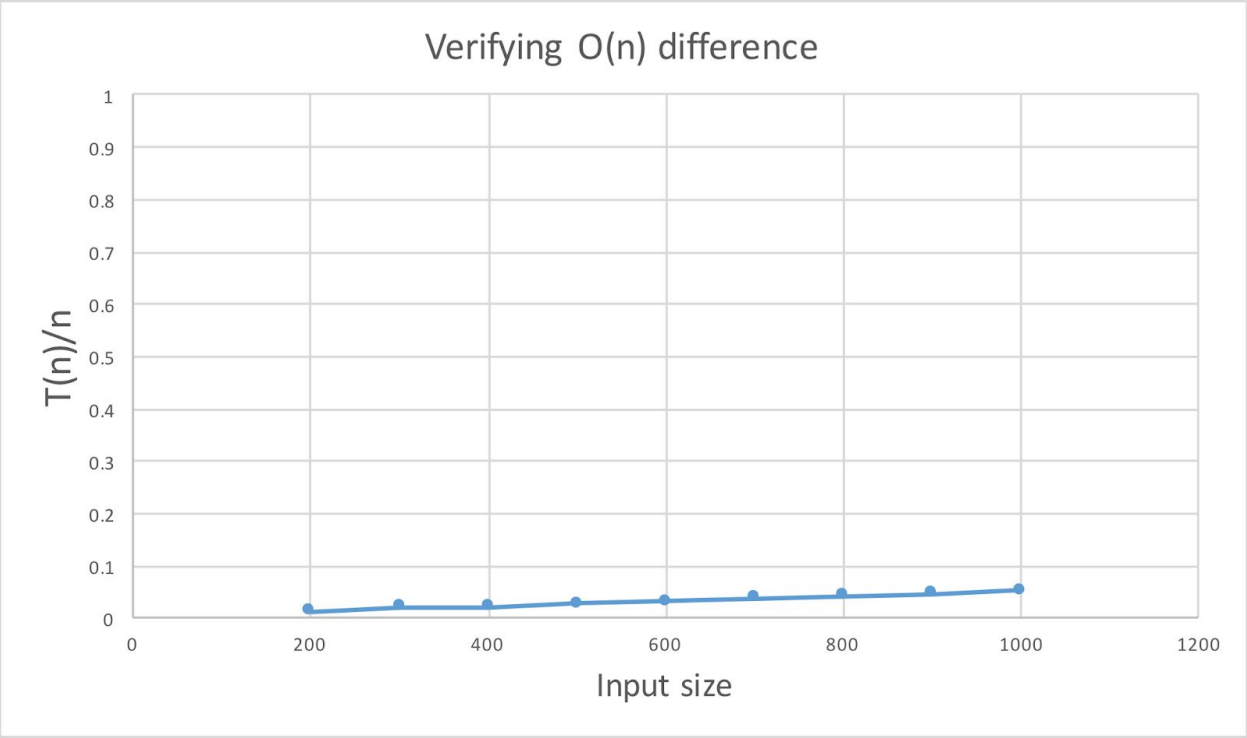
Difference can be split into three major actions that need attention

4. Inorder traversals of two binary trees
5. Finding exclusive elements of first array with second
6. Constructing BST

If n is number of inputs in set 1 and m is number of inputs in set 2, inorder traversals are going to take O(n+m). At the most while finding exclusive elements, we only traverse the entire first array containing n elements. Therefore exclusive common elements takes O(n) time. We construct a BST from at most n elements since we only took elements from first set. Therefore constructing BST is also going take only O(n) time. Let T(n+m) be the time complexity for intersection.

T(n+m) = O(n+m) + O(n) + O(n) = O(n+m)

Therefore we perform difference operation in linear time.



From the above graph it can be been that T(n)/n is a constant for varying sizes of input.

7. Comparison and analysis

In this section of the report a comparative analysis is made with respect to the STL set implementation. Random data is generated in increasing input size and the time taken by each implementation is recorded.

```
 Size = 18100
Our Implementation | Time taken : 9.23408 ms to insert 18100 elements
STL set function | Time taken : 11.0156 ms  to insert 18100 elements

 Size = 19100
Our Implementation | Time taken : 10.0243 ms to insert 19100 elements
STL set function | Time taken : 11.2175 ms  to insert 19100 elements

 Size = 20100
Our Implementation | Time taken : 10.9706 ms to insert 20100 elements
STL set function | Time taken : 12.2779 ms  to insert 20100 elements

 Size = 21100
```

Our Implementation | Time taken : 11.2762 ms to insert 21100 elements
STL set function | Time taken : 12.1594 ms to insert 21100 elements

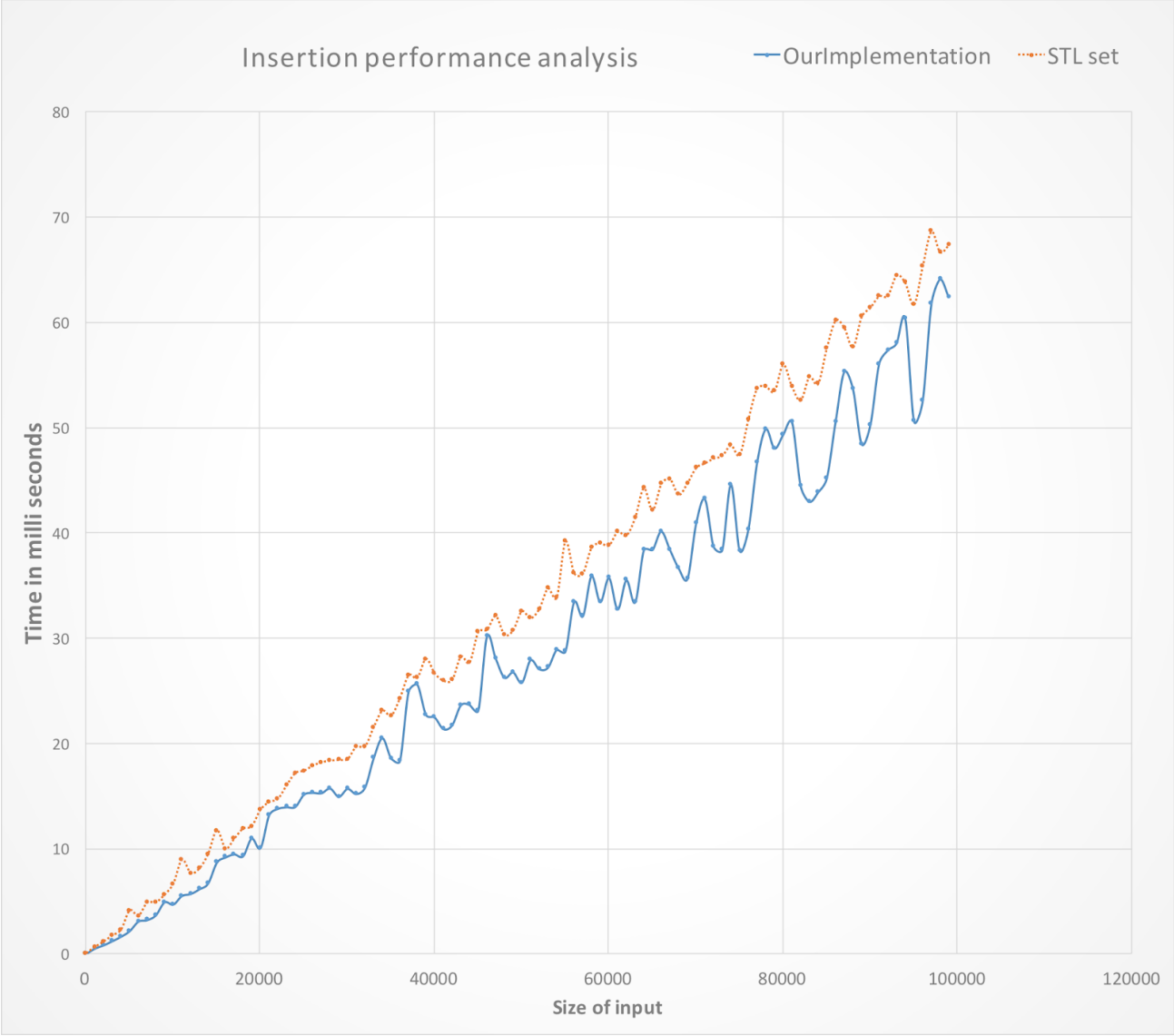
 Size = 22100
Our Implementation | Time taken : 11.6988 ms to insert 22100 elements
STL set function | Time taken : 13.7063 ms to insert 22100 elements

 Size = 23100
Our Implementation | Time taken : 12.4556 ms to insert 23100 elements
STL set function | Time taken : 13.9843 ms to insert 23100 elements

 Size = 24100
Our Implementation | Time taken : 14.2935 ms to insert 24100 elements
STL set function | Time taken : 15.1851 ms to insert 24100 elements

 Size = 25100
Our Implementation | Time taken : 15.3933 ms to insert 25100 elements
STL set function | Time taken : 17.0886 ms to insert 25100 elements

 Size = 26100
Our Implementation | Time taken : 15.2322 ms to insert 26100 elements
STL set function | Time taken : 17.2016 ms to insert 26100 elements



Input size varying from 0 to 100000 in steps of 100 is taken. The time taken in milliseconds is plotted in Y axis and the input size in plotted on the X axis.

It can be seen that AVL implementation of set outperforms the standard implementation of set in most cases.

8. Application of Sets : Friends management in Social Networks

The application of sets and its scope is vast ranging from probability theory to string theory, from finding shortest paths to finding clusters. Here we have taken once such application, that involves all of the fundamental operations on set we have discussed above. With the introduction of internet, people are becoming more connected than ever as time passes. It is not a wonder that social networking sites are also growing larger day by day as more people feel the need to connect to other people. We can see how sets can be used to solve some complicated problems in social network. Here we solve three such problems:

1. Getting friend suggestions
2. Finding mutual friends
3. Finding exclusive friends

8.1. Getting friend suggestions

Social networks can be precisely represented using a graph where each node is a person (profile of a person) and if there exists an edge between two nodes, it means they are connected as friends. There may be multiple heuristics involved while suggesting a friend to a profile, but for our application, we can keep it simple. We can suggest a profile to a person if they are direct friends to any of self's immediate friends. This can be done with our *set union operation* with the following algorithm:

1. Store friend profiles of each user in a set.
2. Get all the friend profiles of current user.
3. For each friend profile of current user, get all their friend profiles.
4. Use **union operation** on all of friends' friends set and suggest them to the user.

Example:
9 : 1
8 : 1
7 : 2
6 : 2
1 : 0 8 9
0 : 1 2 3
2 : 0 6 7
3 : 0 4 5
4 : 3
5 : 3

1. Create user
2. Add friend
3. View mutual friends
4. Get friend suggestions
5. View exclusive friends
6. View entire network
7. Exit
4

Enter the user : 0

4 5 6 7 8 9

In the above representation of friend network, a user_id : u1 u2 …. Represents all friends of user_id. So for user 0, we have immediate friends as 1, 2 and 3. So the friend suggestions will be union of friends set of all of user's friends. Therefore the output is set containing the friend suggestions excluding the user and user's immediate friends.

8.2. Finding Mutual Friends

Sometimes people may not know another person personally but would like to stay connected with them because the person has mutual friend(s) with that profile. For this, we use *set intersection* operation with the following algorithm.

1. Get friends set of current user as set1
2. Get friends set of profile as set2
3. Find **set intersection** between two sets
4. Return the resultant set as mutual friends list

Example:
4 : 0 1
3 : 0 1
2 : 0 1
0 : 2 3 4
1 : 2 3 4
1. Create user
2. Add friend
3. View mutual friends
4. Get friend suggestions

Choice : 3
Enter the user1: 0
Enter the user2: 1
Mutual friends : 2 3 4

8.3. Finding exclusive friends

Exclusive friends are friends that are just in our friends list and are not in mutual friends of any of our other friends. To find such kind of friends, we can use the *difference operation of sets* with the following algorithm.

1. Get friends of current user as set1.
2. For each user in current user friend list, do the following operations:
    a. Get friends list of current friend as set2
    b. Perform **set difference operation** of set1 and set2

Return the resultant set as set of exclusive friends.

Example:
5 : 0
4 : 0
3 : 0 2
2 : 0 1 3
0 : 1 2 3 4 5
1 : 0 2

1. Create user
2. Add friend
3. View mutual friends
4. Get friend suggestions
5. View exclusive friends
6. View entire network
7. Exit
Choice : 5

Enter the user : 0
4 5

9. Conclusion

We conclude with the above as supporting data, that with some optimizations in the operations, our implementation of sets using AVL Tree is performing better than the standard STL implementation of an ordered set in all of the operations for all sized inputs. By extension, due to its logarithmic time insertion and deletion and linear time set operations, it performs better than arrays and standard STL vectors.

Sets can be used to solve a wide range of problems and most of the fundamental operations of sets can be done in either logarithmic time, O(logn) or linear time O(n), O(n+m). Due to this, this data structure can be used in applications where insertion and removals are done in abundance and set operations are done in moderation.

GIT Repository Link : https://github.com/tarun0409/set-operations.git