International Institute of Information Technology, Hyderabad

Advance Problem Solving

Project Report on
Set Data Structure

Course Instructors
Dr. Venkatesh Chopella
Dr. Vikram Pudi

Submitted By
Prabha Pandey
2018201053

Priya Upadhyay
2018202012

Mentor
Surbhi Goyal

Objective:
Implementing a data structure for sets on which operations like union, intersection and difference can be performed.

1. Introduction

A set is collection of similar type of elements. A set consists of distinct elements. It is an implementation of mathematical concept of finite set. Depending on whether the values are stored in order or not, the set is called *ordered set* or *unordered set*. Following binary operations can be performed on sets-

1. Union
2. Intersection
3. Difference

1.1. Union

Union is a binary operation which takes two sets. say *A* and *B*, as input and results in output set *C*, such that each element of set *C* either belongs to set *A* or set *B*.

$$A \cup B = \{x : x \in A \text{ or } x \in B\}$$

1.2 Intersection

Intersection is a binary operation which takes two sets, say *A* and *B,* as input and results in output set *C*, such that each element of set *C* belongs to both set *A* and set *B.*

$$A \cap B = \{x : x \in A \text{ and } x \in B\}.$$

1.3 Diffrence

Difference is a binary operation which takes two sets, say *A* and *B,* as input and results in output set *C,* such that for operation *B-A* (also represented as *B|A*) the output set *C* contains elements of set *B* that do not belong to set *A.*

$$B \setminus A = \{x \in B \mid x \notin A\}.$$

2. Various implementations

A set data structure can be implemented in multiple ways.

- Tree based approach(ordered set)
- Hash based approach(unordered set)

2.1 Tree based approach

Since tree based approach implements ordered set, binary search trees are used for implementation (inorder traversal of BST gives elements in sorted order).
Worst case time complexity of an operation can go upto O(n), therefore Balanced Binary Search Trees such as *AVL Tree* and *Red Black Tree* are used, every operation takes O(logn) time in a balanced binary tree.

Choosing a good implementation depends on knowing how we will use the data structure. Since operations like union, intersection and difference can be performed efficiently on sorted list of elements, this project is implemented using Tree based approach.

2.2 Hash based approach

A set can also be implemented using hash tables. If good hash function is used and size of table is right(load factor), O(1) time complexity can be expected for insertion and deletion. In worst case look up operation can be as worse as O(n). Hash table based implementation is not compact, i.e, load factor is almost never one.

3. AVL Tree Implementation of Set

AVL trees are a form of *balanced* binary search trees (BBST). BBSTs augment the binary search tree invariant to require that the heights of the left and right subtrees at every node differ by at most one ("height" is the length of the longest path from the root to a leaf).

Balance Factor = (height of left subtree) – (height of right subtree)

Range of balance factor = {-1, 0, 1}

Since it is balanced, therefore operations like insertion, deletion and search take O(log n) time in worst case.

Pseudo-code for Insertion and Deletion in AVL Tree

Insert(root, x):

```
1.      if (root == NULL):
2.                root = new_node(x)
3.      else:
4.                if( root->data > x):
5.                     insert(root->left,x)
6.                     check if there is unbalanced nodes
7.                     perform appropriate rotations
8.                else if(root->data < x):
9.                     insert(root->right,x)
10.                         check if there is unbalanced nodes
11.                         perform appropriate rotations
12.                    else:
13.                         node already exists therefore continue
14.
15.          return root
```

Delete (root, x):

```
1.      if ( root == NULL ) :
2.                return NULL
3.      else if ( root->data > x ):
4.                delete(root->left , x)
5.      else if( root->data < x ):
6.                delete(root->right, x)
7.      else:
8.           if ( root is leaf node):
9.                delete(root)
10.               return NULL
11.          else if ( root has one child):
12.               root->data=child->data
13.               remove child
14.               return root
15.          else if ( root has two children):
16.                find inorder successor of root
17.                root->data=successor->data
18.                root->right=delete(root->right,successor->data)
19.     Check for unbalanced node
20.     if node is unbalanced:
21.         perform appropriate rotaions
22.     return root
```

4. Red Black Tree Implementation of Set

AVL Trees have a good performance for search operation, but when it comes to insertion and deletion, Red black tree has less number of rotations in comparison to AVL tree. Due to this reason Red Black trees are preferred for implementation of set data structure. C++ implementation of *STL set* is also based on Red Black Tree. A red black tree should follow these four properties-

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

Since red black tree is a self balancing tree, every operation takes O(logn) time.

Pseudo code for insertion in Red Black Tree

```
Insert(root, x)
1.      if tree is empty
2.           root = createNode(x, BLACK)
3.      else
4.           insertAt = find(root, x)
5.           if insertAt->data==x
6.                key with value x already exists in tree. Return.
7.           else
8.                newNode = createNode(x,RED)
9.                if insertAt->data > x
10.                    insertAt->right = newNode
11.               else
12.                    insertAt->left = newNode
13.               RedRedCorrection(root, newNode)
```

RedRedCorrection() corrects the violation of property no. 3.

Let the new node be x, its parent be p, its grandparent be g and its uncle be u. Following cases are handled by RedRedCorection() :

1. If p is red and and u is also red then, simply change the color of p and u to black and make thier parent's color as red. Now recursivly call red-red correction on g.

2. If p is the left child of g and x is also the left child of p, then its a case of LL problem. Swap the colors of g and p, then perform Right Rotation(at g).

3. If p is the left child of g and x is the right child of p, then its a case of LR problem. Do a left rotation(at p) to convert the problem into LL problem. Swap the colors of g and x, then perform right rotation(at g).

4. If p is the right child of g and x is also the right child of p, then its a case of RR problem. Swap the colors of g and p, then perform Left Rotation(at g).

5. If p is the right child of g and x is the left child of p, then its a case of RL problem. Do a right rotation(at p) to convert the problem into RR problem. Swap the colors at g and x, then perform Left Rotation(at g).

```
Delete(root, x)
1.      if tree is empty
2.                return
3.      else
4.                deleteNode = find(root, x)//find node with key x
5.                replacementNode = findInorderSuccessor(deleteNode)
6.                if deleteNode is leaf node
7.                     if deleteNode is BLACK
8.                          fixDoubleBlack(root,x)
9.                     if parent->left == deleteNode
10.                              parent->left = NULL
11.                         else
12.                              parent->right = NULL
13.                         delete deleteNode
14.
15.                    if deleteNode has one child
16.                         if deleteNode==root
17.                              swap(replacementNode, root)
18.                         else
19.                              swap(replacementNode, deleteNode)
20.                              replacementNode->parent = parent
21.                         delete deleteNode
22.                    if deleteNode has two childeren
23.                         swap(replacementNode, deleteNode)
24.                         delete(root, replacementNode)
```

When a black node is to be deleted and its replacement node is also black, then after deletion, one of the paths from root to leaf node will have one less black node. Which violates the property no. 4. **fixDoubleBlack()** is the function which handles this problem.

```
fixDoubleBlack(root, deleteNode)
1.      if root==deleteNode
2.                return
3.      if sibling == NULL
4.                fixDoubleBlack(root, parent)
5.      else if sibling(deleteNode)->color == BLACK
6.           if leftChild(sibling) is RED
7.                if parent->left == sibling
8.                     sibling->left->color = sibling->color
9.                     sibling->color = parent->color
10.                    RightRotate(root, parent)
11.               if parent->right == sibling
12.                    sibling->left->color=parent->color
13.                    RightRotate(root, sibling)
14.                    LeftRotate(root, parent)
15.               parent->color = BLACK
16.          if rightChild(sibling) is RED
17.               same as 7-15 with right and left exchanged
18.       if sibling is RED
19.          parent->color = RED
20.          sibling->color = BLACK
21.          if parent->left == sibling
22.               RightRotate(root, parent)
23.          else
24.               LeftRotate(root, parent)
25.          fixDoubleBlack(root, deleteNode)
```

5. Set Operations using Balanced Binary Trees

Union, intersection and difference of two sets is basically, union, intersection and difference of two trees. Inorder traversal of the two trees gives us two sorted lists. Modified versions of merge algorithm are used to perform the mentioned operations. The result after merging is inserted into a new tree, which is the resultant set.

5.1 Union

```
Union ( Set s, Set s1):
1.      vector v1 = inoder traversal of s
2.      vector v2 = inorder traversal of s1
3.      i=0
```

```
4.      j=0
5.           Set temp
6.      while( i < v1.size() && j < v2.size()):
7.                if( v1[i] == v2[j]):
8.                     insert v1[i] in temp
9.                     i++
10.                         j++
11.                    else if( v1[i] < v2[j] ):
12.                         insert v1[i] in temp
13.                         i++
14.                    else if( v1[i] > v2[j] ):
15.                         insert v2[j] in temp
16.                         j++
17.          if(i < v1.size())
18.                    insert remaining elements of v1 into temp
19.               if(j < v2.size())
20.                    insert remaining elements of v2 into temp
21.
22.          return temp
```

5.2 Intersection

Intersection(Set s, Set s1):

```
1.      vector v1 = inorder traversal of s
2.      vector v2 = inorder traversal of s1
3.      i=0
4.           j=0
5.      Set temp
6.      while( i < v1.size() && j < v2.size()):
7.                if( v1[i] == v2[j]):
8.                     insert v1[i] in temp
9.                     i++
10.                         j++
11.                    else if ( v1[i] < v2[j]):
12.                         i++
13.                    else if ( v1[i] > v2[j] ):
14.                         j++
15.          return temp
```

5.3 Difference

Difference(Set s, Set s1):

```
1.      vector v1=inorder traversal of s
2.      vector v2=inorder traversal of s1
```

```
3.      i=0
4.      j=0
5.      Set temp
6.      while ( i < v1.size() && j < v2.size()):
7.                  if( v1[i]==v2[j]):
8.                        i++
9.                        j++
10.                       else if ( v1[i] < v2[j]):
11.                             insert v1[i] in temp
12.                             i++
13.                       else if ( v1[i] > v2[j]):
14.                             j++
15.           if( i < v1.size()):
16.                       insert remaining elements of v1 into temp
17.
18.           return temp
```

5.4 Complexity Analysis

Inorder traversal of two input sets take O(n) time, where n is the size of input set.

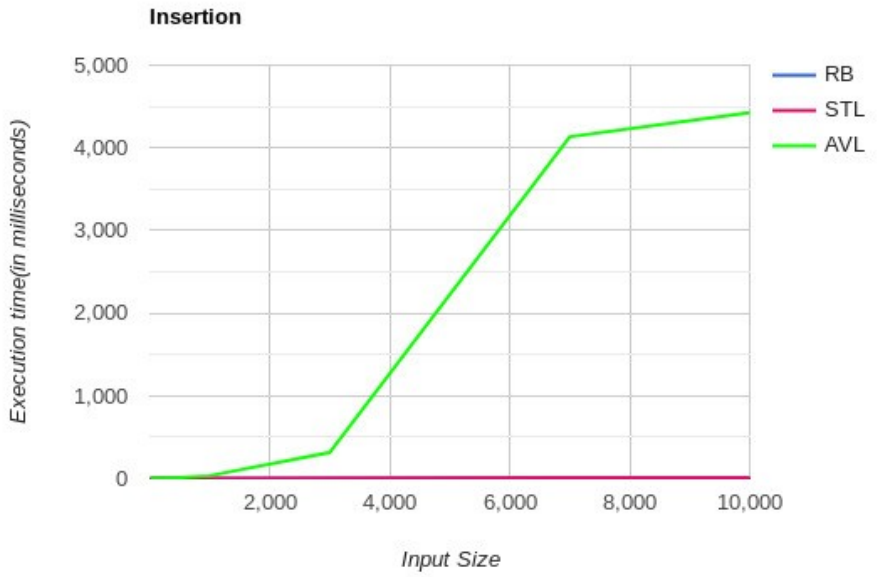
Merging the inorder traversals according to the operation(i.e, union, intersection or difference) takes O(n+n) time, where n is size of both the sets.

Inserting the result of operation(intersection, union and difference) into a new set means inserting into a new tree. Each insertion takes O(logn) time, therefore, for 2n insertions it will take O(nlogn) time.

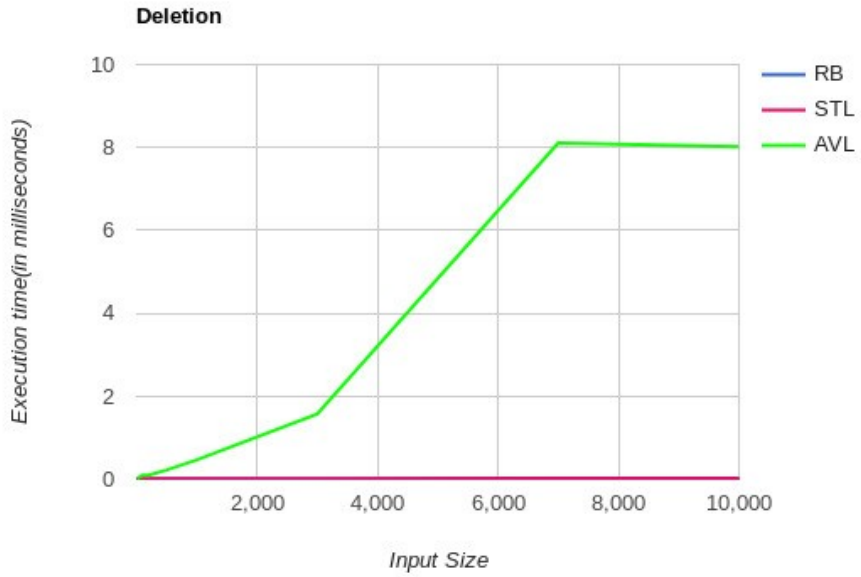
Overall Time and Space complexity of union, intersection and difference is O(nlogn) and O(n) respectively.

6. Performance Graphs

6.1. Insertion



6.2. Deletion

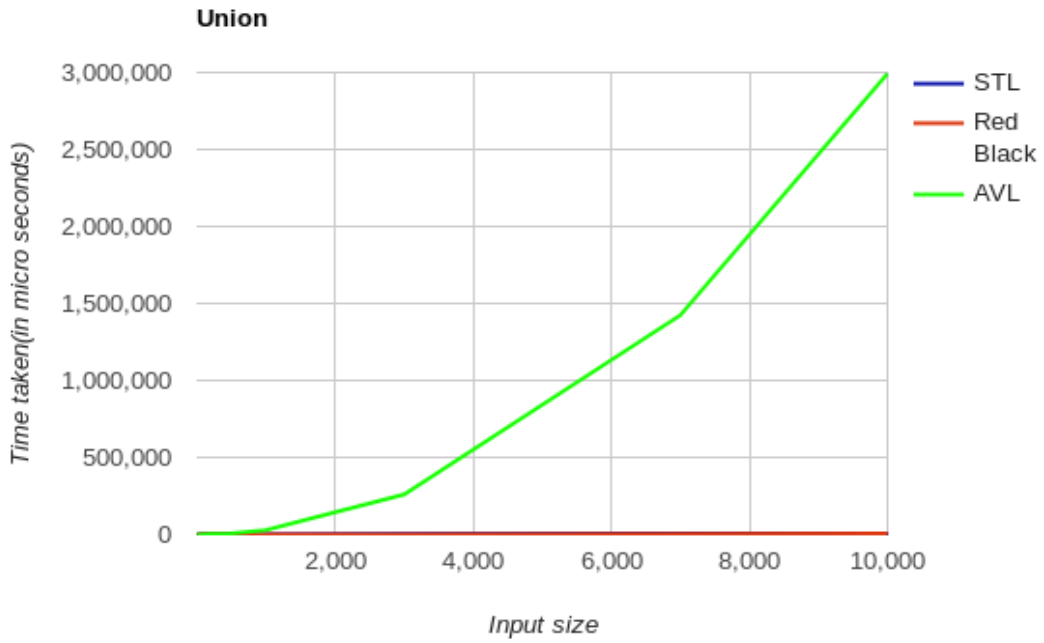


Inference from performance graphs of Insertion and Deletion:

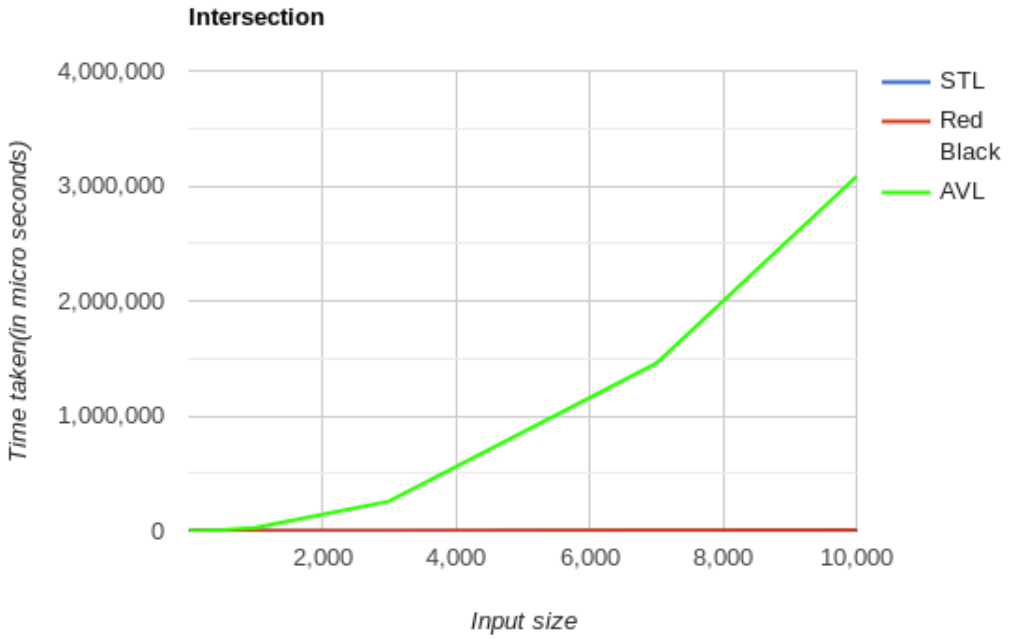
As it is evident from the graph, AVL tree implementation of set requires more time for insertion and deletion due to more number of rotations.

Whereas, performance of set implemented using RB tree is quite close to C++ STL(Standard Template Library) set. Graph of RB tree implementation and C++ STL overlap each other.

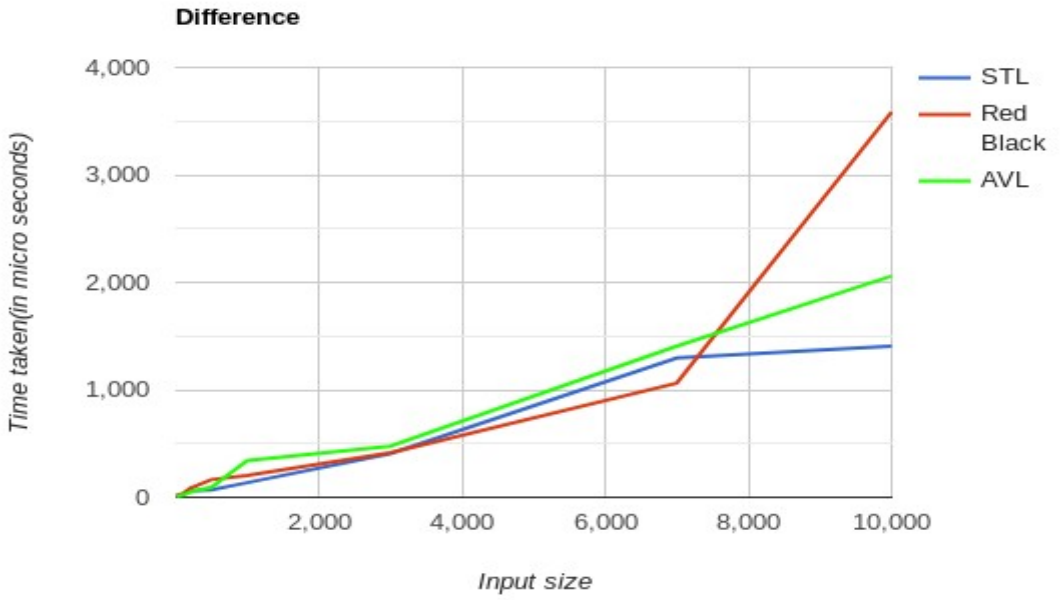
6.3. Union



6.4 Intersection



6.5 Difference



Inference from performance graphs of Union, Intersection and Difference:

Union, Intersection and Difference operations on set consitute of two parts, inorder traversal and inserting elements into new set after some processing of elements.
Since the main operation in insertion, graphs of these operations also depict the same behaviour as they did for insert operation.
Anomalus behaviour of Red Black tree implementation is observed in case of set difference operation when the input size reaches more than 7000 elements.

7. Application

As an application we have used set data structure (Both AVL and Red black) in **Dijkstra's algorithm** for finding the shortest path from the source vertex to all other vertices in the graph.

7.1. Algorithmic Steps

```
Dijkstra(graph g):
   1. set the vertices distance, from source vertex, as infinity
   2. set all vertices as unvisited
   3. take set s data structure (acting as min-priority queue)
```

```
4. s.insert({0,1}) //for source vertex {distance,vertex}
5. while(!s.empty()):
6.       pair p=s.begin() //min. weight edge vertex is first in the set
7.       s.erase(s.begin()) //extract min. weight edge vertex
8.       x=p.second //popped vertex
9.       weight=p.first
10.           if x is visited:
11.                continue
12.           else:
13.                visited[x]=true
14.                for every vertex i attached to x:
15.                     if distance[x]+weight < distance[g[x][i]]:
16.                          update distance[g[x][i]]
17.                           s.insert({distance[g[x][i]], i})
18.           return distance
```

7.2. Complexity Analysis

Using AVL or Red-Black Tree:

1. Every time the while loop executes, one vertex is extracted from the set. Assuming that there are V vertices in the graph, the set may contain O(V) vertices. Each pop operation takes O(log V) time. So the time required to execute the while loop itself is O(V log V).

2. A for loop is executed for each edge attached to the vertex being popped i.e, O(E) times, and the calculated distance of edges is inserted into the set if it is less than the old distance, which takes O(log V) time i.e, O(E log V) total time.

   From 1 and 2 total time complexity:

   $$O(V \log V) + O(E \log V) = \mathbf{O((E+V) \log V)}$$

7.3. Test Application of Set on the following link

https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/

7.4. Observation

AVL tree implementation of set gives time limit exceeded message when large input is provided to program of Dijkstra's on the above link for testing.

Whereas Red-Black tree implementation of set gives satisfactory performance for the same problem.

8. Conclusion

Our analysis showed us that ordered set data structure can be implemented using balanced binary search trees(BBST).
Amongst the BBST's that we used to implement the same, Red-Black tree showed better performance as compared AVL tree, reason being more number of rotations in AVL while performing insertion and deletion.
The actual set in C++ Standard template library also uses red-black tree for its implementation.

9. Link to Github repository

https://github.com/upadhyay-p/APS_Project.git

10. References

https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/