Fenwick Trees, Tries and Splay Trees

Group Number: 11

Team Members:

1) Rajneesh Pandey (106119100)
2) Divakar PS (106119030)
3) Satyarth Pandey (106119112)

Introduction:

The report made is based on a detailed comparison of three data structures that are related / seemingly similar, in terms of structure (Trees) which are:

- Splay Trees
- Tries
- Fenwick Tree

on different functionality(operations) that can be performed by each one of them. The report starts by a detailed review of each of the data structure with the applications, algorithm and examples for each operation and then is followed by an extensive graphical analysis of each of the operation among all the three data structures. Then we conclude by a brief note on the performance and functionality of all the three data structures.

SPLAY TREES

INTRODUCTION

A splay tree is a self-balancing tree, but AVL and Red-Black trees are also self-balancing trees then. But what makes the splay tree unique among these two trees? It has one extra property that makes it unique is splaying.

A splay tree contains the same operations as a Binary search tree, i.e., Insertion, deletion and searching, but it also contains one more operation, i.e., splaying. So, all the operations in the splay tree are followed by splaying.

Splaying means that the node which has been recently visited (irrespective of the operation) will become as the root of the tree so that, next time when that node is being searched, we get it at a lesser time as compared to previous time, which actually makes more sense, because in real life situations we find a lot of places where a set of data is being referred most of the times. Thus, it makes more sense to keep those mostly referred data at the root, which is performed by splaying operation

Splay trees are not strictly balanced trees, but they are roughly balanced trees.

We will explain more about splaying in upcoming section.

REAL TIME APPLICATION:

1) A good example is a network router. A network router receives network packets at a high rate from incoming connections and must quickly decide on which outgoing wire to send each packet, based on the IP address in the packet. The router needs a big table (a map) that can be used to look up an IP address and find out which outgoing connection to use. If an IP address has been used once, it is likely to be used again, perhaps many times. So, we can build the above look up table using splay tree. if we have look up for a IP address once it will come on top of tree, so next time that IP address come again, we can find its entry on top and don’t have to go till bottom.

2) Splay trees are typically used in the implementation of caches, memory allocators, garbage collectors, data compression, ropes (replacement of string used for long text strings), in Windows NT (in the virtual memory, networking, and file system code) etc.

3) A example of a domain in which splay trees are used are Intrusion detection systems (IDS)which are an essential part of the security infrastructure. They are used to detect, identify and stop intruders. The administrators can rely on them to find out successful attacks and prevent a future use of known exploits. IDS are also considered as a complementary solution to firewall technology by recognizing attacks against the network that are missed by the firewall. Nevertheless, IDS are plagued with false positive alerts, letting security professionals to be overwhelmed by the analysis tasks. Therefore, IDS employ several techniques in order to increase the detection probability of suspect threats while reducing the risk of false positives. This leads to the construction of a decision tree such as the splay tree used to verify the traffic. The proposed strategy to detect intrusions using protocol analysis consists in extracting data from each received packet and then traversing a pre-built decision tree.

4) Splay tree are used in the GCC compiler and GNU C++ library, the sed string editor, the most popular implementation of Unix malloc, Linux loadable kernel modules, and in much other software.

REQUIREMENTS:

There is no requirement as such, and the data can be in any order, but the knowledge of Binary Search Tree is required, as the insertion, searching and deletion operations are same as that of Binary Search Tree.

Advantages: In the splay tree, we do not need to store the extra information. In contrast, in AVL trees, we need to store the balance factor of each node that requires extra space, and Red-Black trees also require to store one extra bit of information that denotes the color of the node, either Red or Black.

Drawbacks: The major drawback of the splay tree would be that trees are not strictly balanced, i.e., they are roughly balanced. Sometimes the splay trees are linear, so it will take O(n) time complexity.

OPERATIONS:

1) SPLAYING:

Explanation:

Splaying means that the operation that we are performing on any element should become the root node after performing some rearrangements.

The rearrangement of the tree will be done through the rotations.

Rotations

There are six types of rotations used for splaying:

1. Zig rotation (Right rotation)
2. Zag rotation (Left rotation)
3. Zig zag (Zig followed by zag)
4. Zag zig (Zag followed by zig)
5. Zig zig (two right rotations)
6. Zag zag (two left rotations)

Factors required for selecting a type of rotation

The following are the factors used for selecting a type of rotation:

- Does the node which we are trying to rotate have a grandparent?
- Is the node left or right child of the parent?
- Is the node left or right child of the grandparent?

Cases for the Rotations

Case 1: If the node does not have a grand-parent, and if it is the right child of the parent, then we carry out the left rotation; otherwise, the right rotation is performed.

Case 2: If the node has a grandparent, then based on the following scenarios. the rotation would be performed:

Scenario 1: If the node is the right of the parent and the parent is also right of its parent, then *zig zig right right rotation* is performed.

Scenario 2: If the node is left of a parent, but the parent is right of its parent, then *zig zag right left rotation* is performed.

Scenario 3: If the node is right of the parent and the parent is right of its parent, then *zig zig left left rotation* is performed.

Scenario 4: If the node is right of a parent, but the parent is left of its parent, then *zig zag right-left rotation* is performed.

Algorithms:

Algorithm of left rotation:

```
  left.rotation(T, x)
  y=x->right
  x->right = y->left
  y->left = x
  return y
```

Algorithm of right rotation:

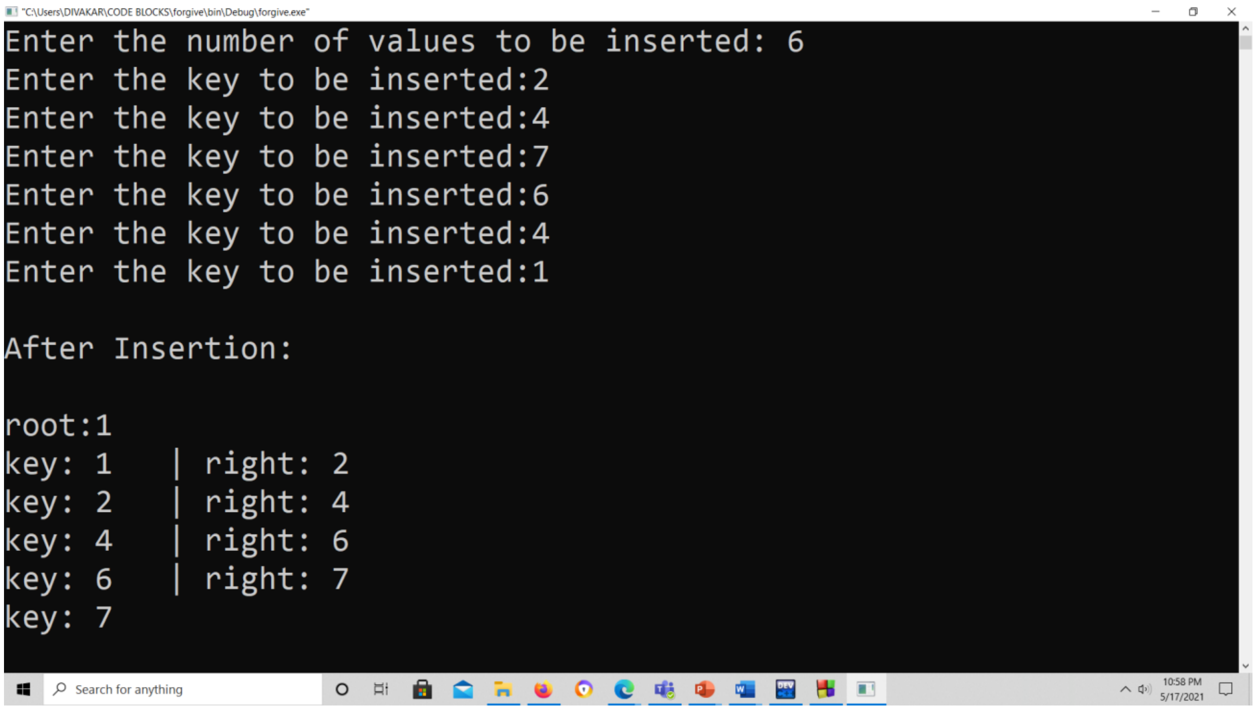
```
  right.rotation(T, x)
  y= x->left
  x->left=y->right
  y->right=x
  return y
```

Algorithm of splaying:

```
Splay(T, N)
 while(n->parent !=Null)
 {
    if(n->parent==T->root)
       if(n==n->parent->left)
          right_rotation(T, n->parent)
       else
          left_rotation(T, n->parent)
    else
         p= n->parent
         g = p->parent
       if(n=n->parent->left && p=p->parent->left)
           right.rotation(T, g), right.rotation(T, p)
       else if(n=n->parent->right && p=p->parent->right)
           left.rotation(T, g), left.rotation(T, p)
       else if(n=n->parent->left && p=p->parent->right)
           right.rotation(T, p), left.rotation(T, g)
       else
           left.rotation(T, p), right.rotation(T, g)
 }
```

Example:

six values are inserted into the tree and the tree is displayed:

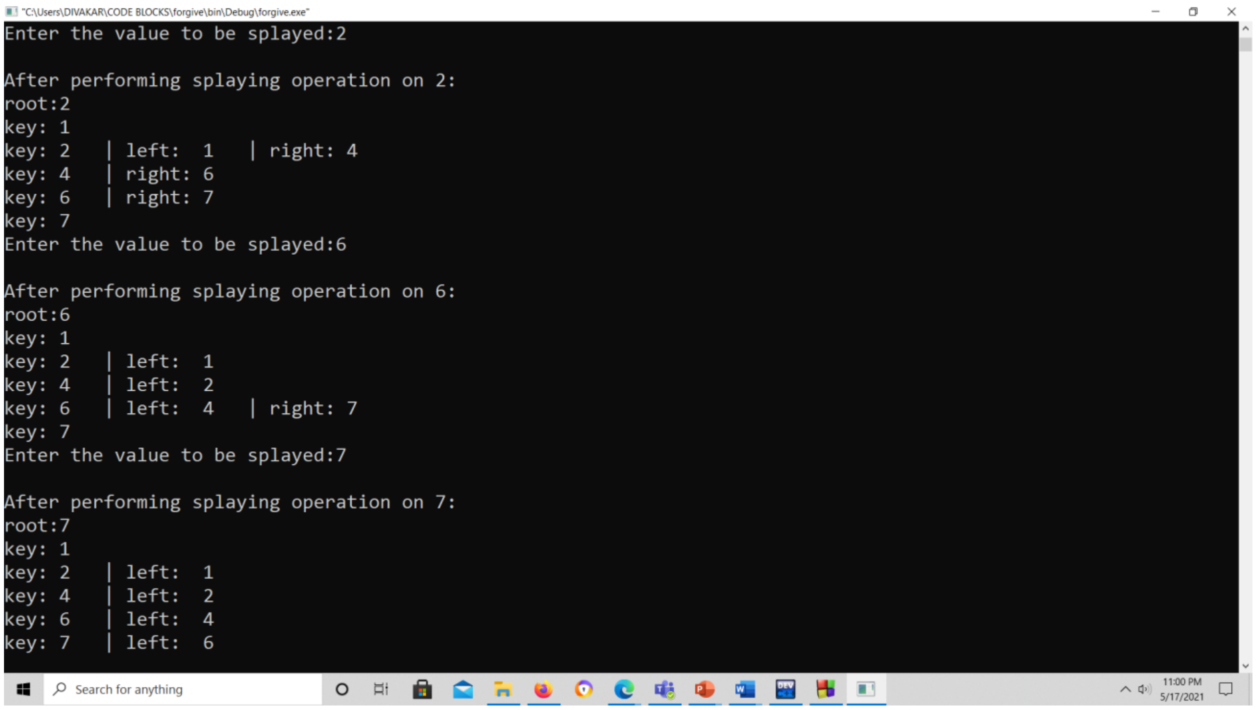


```
Enter the number of values to be inserted: 6
Enter the key to be inserted:2
Enter the key to be inserted:4
Enter the key to be inserted:7
Enter the key to be inserted:6
Enter the key to be inserted:4
Enter the key to be inserted:1

After Insertion:

root:1
key: 1    | right: 2
key: 2    | right: 4
key: 4    | right: 6
key: 6    | right: 7
key: 7
```

let us perform splaying on 3 keys : 2,6,7



```
Enter the value to be splayed:2

After performing splaying operation on 2:
root:2
key: 1
key: 2    | left:  1    | right: 4
key: 4    | right: 6
key: 6    | right: 7
key: 7
Enter the value to be splayed:6

After performing splaying operation on 6:
root:6
key: 1
key: 2    | left:  1
key: 4    | left:  2
key: 6    | left:  4    | right: 7
key: 7
Enter the value to be splayed:7

After performing splaying operation on 7:
root:7
key: 1
key: 2    | left:  1
key: 4    | left:  2
key: 6    | left:  4
key: 7    | left:  6
```

We can see that after performing the splaying operation the splayed element becomes the root !

2) SEARCHING:

Explanation:

We get the element to be searched as input from the user.

The searching happens in the same way as that of the Binary Search Tree, where we move to the left subtree if the current node value is greater than the value to be searched or we move to the right subtree if the current node value is lesser than the node to be searched and we return true if the current node value is equal to the value to be searched.

If the search is successful , then we perform splaying on the desired node,

Else if the search is unsuccessful , then we perform splaying on the last node encountered .

Algorithm:

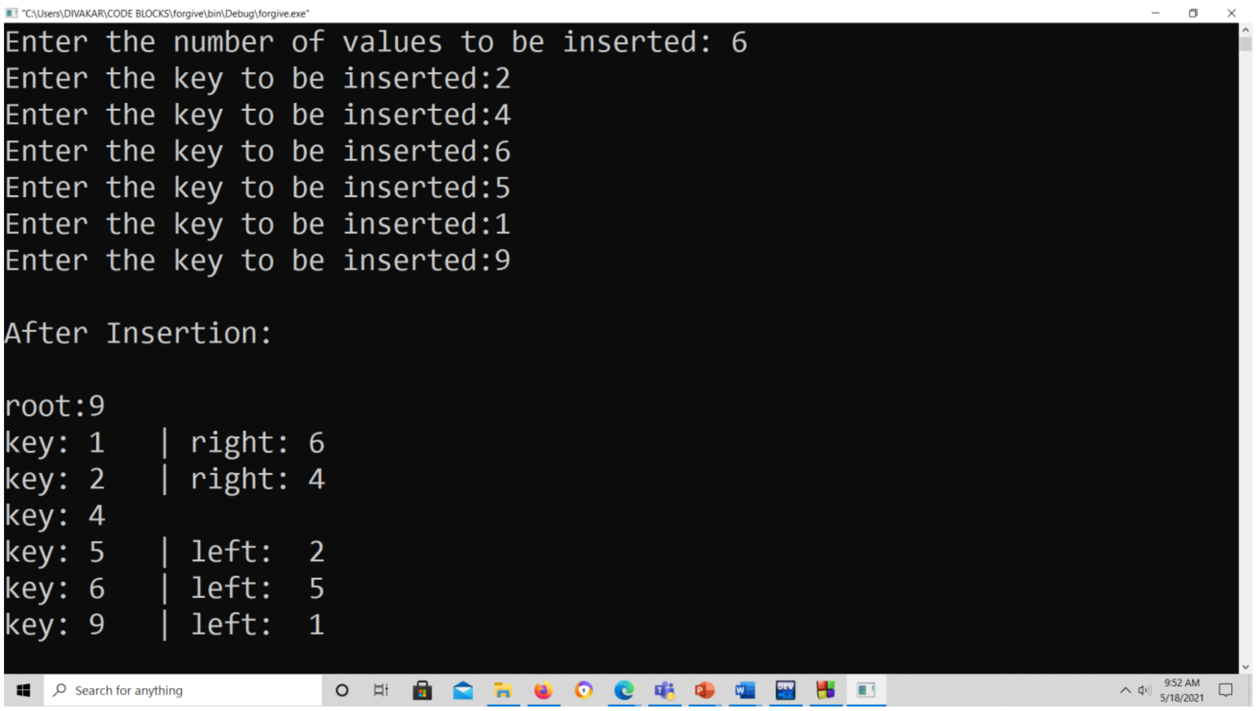
Let v be the value to be searched.

```
Node *P=root;
while(P)                    //iterative implementation of bst searching.
{
  if(P->value==v)
    break;
    if(v<(P->value))
    {
      if(P->left)
        P=P->left;
      else
        break;
    }
    else
    {
      if(P->right)
        P=P->right;
      else
        break;
    }
  }
  Splay(P);                  //splaying on last encountered node !

  if(P->value==v) return true; //return true , if the value =v
  else return false;                  //else return false
```

Examples:

Six values are inserted into the tree.

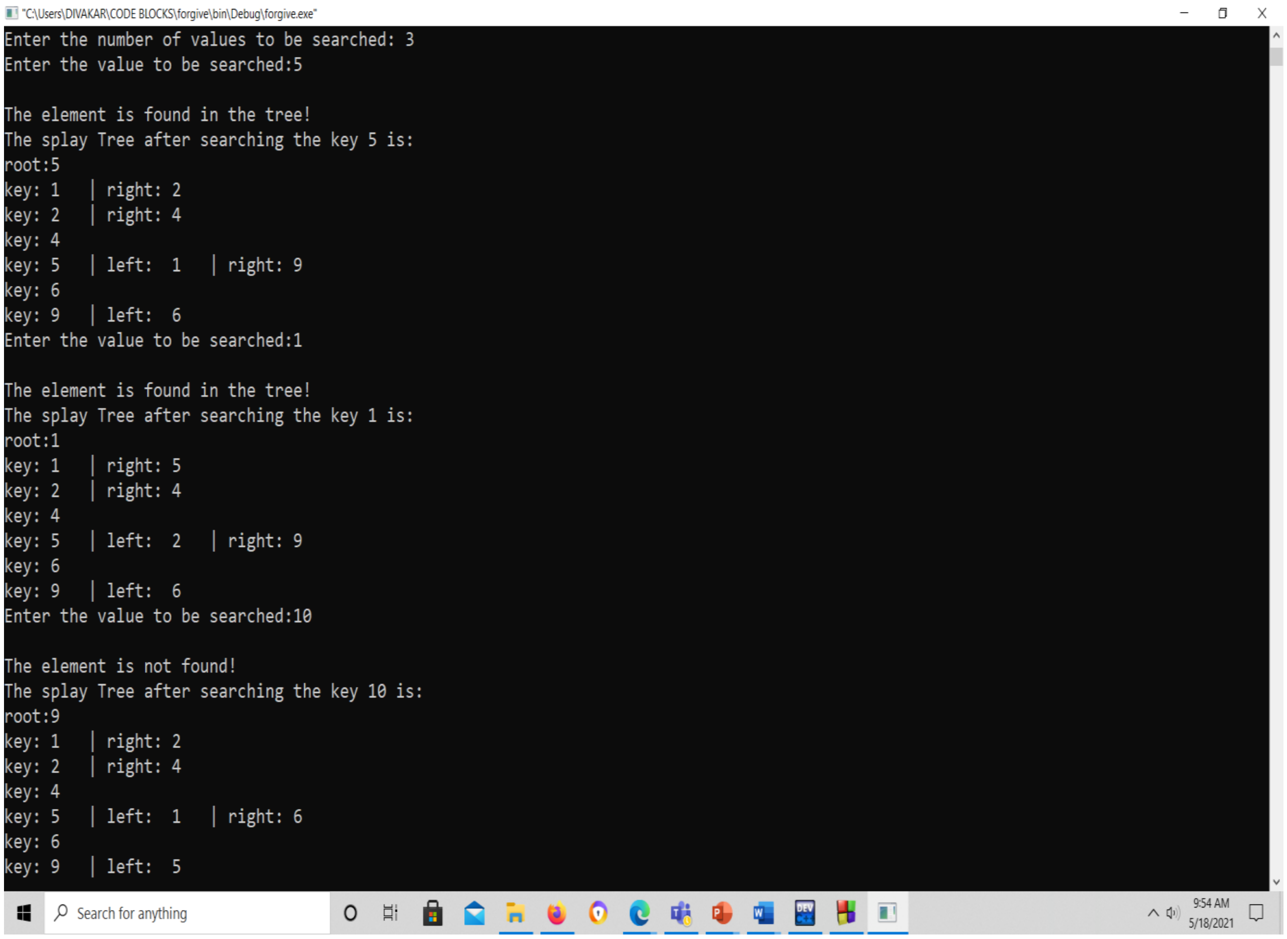


```
Enter the number of values to be inserted: 6
Enter the key to be inserted:2
Enter the key to be inserted:4
Enter the key to be inserted:6
Enter the key to be inserted:5
Enter the key to be inserted:1
Enter the key to be inserted:9

After Insertion:

root:9
key: 1    | right: 6
key: 2    | right: 4
key: 4
key: 5    | left:  2
key: 6    | left:  5
key: 9    | left:  1
```

Let us perform searching on 3 values: 5 ,1 and 10



```
Enter the number of values to be searched: 3
Enter the value to be searched:5

The element is found in the tree!
The splay Tree after searching the key 5 is:
root:5
key: 1    | right: 2
key: 2    | right: 4
key: 4
key: 5    | left:  1    | right: 9
key: 6
key: 9    | left:  6
Enter the value to be searched:1

The element is found in the tree!
The splay Tree after searching the key 1 is:
root:1
key: 1    | right: 5
key: 2    | right: 4
key: 4
key: 5    | left:  2    | right: 9
key: 6
key: 9    | left:  6
Enter the value to be searched:10

The element is not found!
The splay Tree after searching the key 10 is:
root:9
key: 1    | right: 2
key: 2    | right: 4
key: 4
key: 5    | left:  1    | right: 6
key: 6
key: 9    | left:  5
```

We can observe that for the first two cases(5,1) where the search was successful , the searched node becomes the root node.

And for the third case(10) where the search was unsuccessful , the last encountered element(9) while searching has become the root, which ensures that the splaying operation has happened in both the cases.

3) INSERTION:

Explanation:

In the *insertion* operation, we first insert the element in the tree and then perform the splaying operation on the inserted element.

The insertion procedure is same as that of the Binary Search Tree, where we perform BST-searching until we reach the NULL, and then insert the new element according to the value to the last node.

After this we perform splaying on the inserted node.

Algorithm:

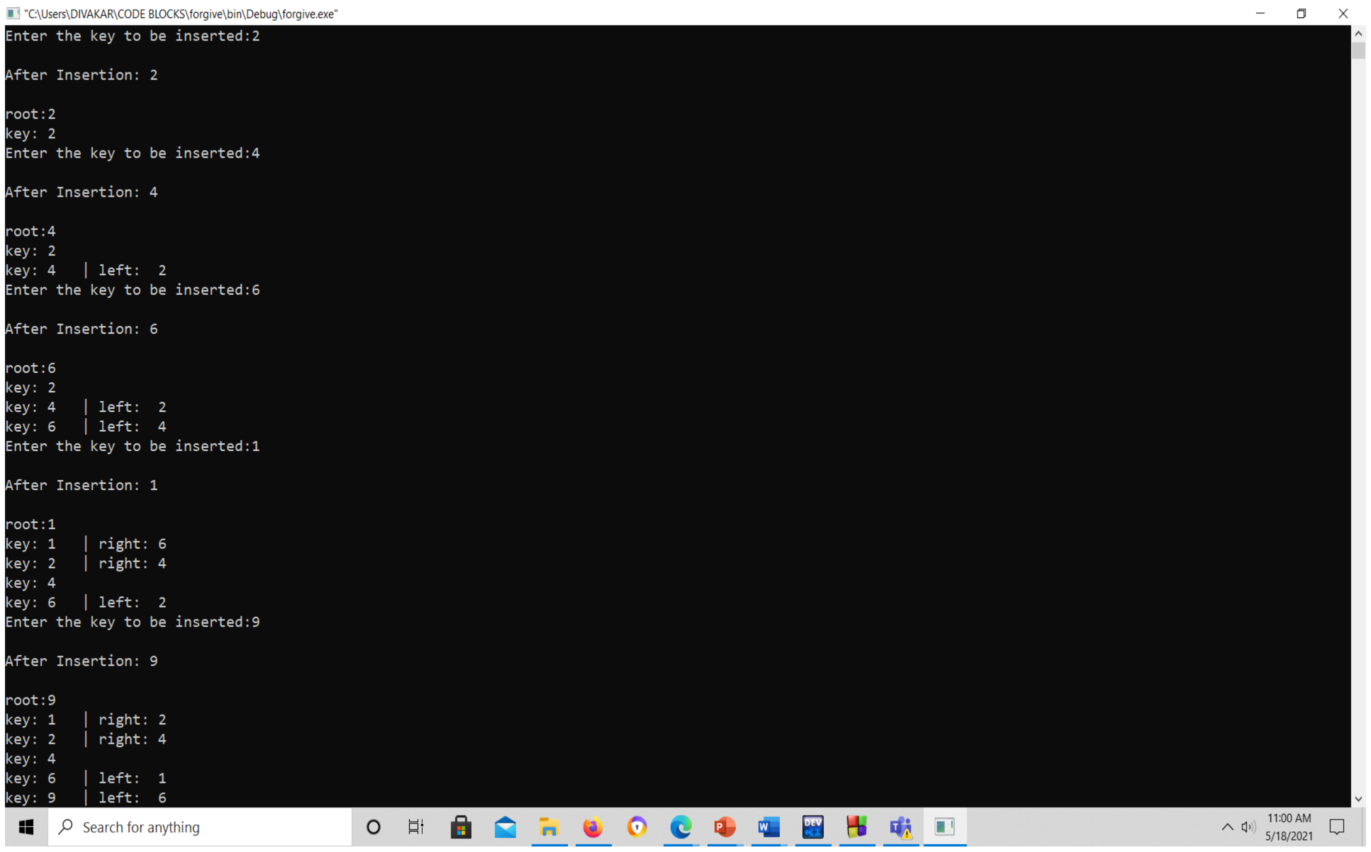
```
Insert(T, n)
     temp= T_root
     y=NULL
     while(temp!=NULL)
        y=temp
        if(n->data <temp->data)
           temp=temp->left
        else
           temp=temp->right
    n.parent= y
    if(y==NULL)
       T_root = n
    else if (n->data < y->data)
       y->left = n
    else
       y->right = n
   Splay(T, n)
```

In the above algorithm, T is the tree and n is the node which we want to insert. We have created a temp variable that contains the address of the root node. We will run the while loop until the value of temp becomes NULL.

Once the insertion is completed, splaying would be performed.

We can refer to the splaying section for the algorithm of splaying operation.

Examples:



"C:\Users\DIVAKAR\CODE BLOCKS\forgive\bin\Debug\forgive.exe"

```
Enter the key to be inserted:2

After Insertion: 2

root:2
key: 2
Enter the key to be inserted:4

After Insertion: 4

root:4
key: 2
key: 4    | left:  2
Enter the key to be inserted:6

After Insertion: 6

root:6
key: 2
key: 4    | left:  2
key: 6    | left:  4
Enter the key to be inserted:1

After Insertion: 1

root:1
key: 1    | right: 6
key: 2    | right: 4
key: 4
key: 6    | left:  2
Enter the key to be inserted:9

After Insertion: 9

root:9
key: 1    | right: 2
key: 2    | right: 4
key: 4
key: 6    | left:  1
key: 9    | left:  6
```

In the above example, we have inserted 5 elements: 2,4,6,1,9

We can see that after every insertion the lastly inserted node becomes the root , this ensures that the splaying operation has happened after every insertion.

For example, if the node 9 is searched at once ,after insertion, we will be able to find it with a constant time complexity instead of going through a complete BST-search ,which takes O(log n) time complexity.

4) DELETION:

Explanation:

As we know that splay trees are the variants of the Binary search tree, so deletion operation in the splay tree would be similar to the BST, but the only difference is that the delete operation is followed in splay trees by the splaying operation.

Types of Deletions:

There are two types of deletions in the splay trees:

1. Bottom-up splaying
2. Top-down splaying

Bottom-up splaying

In bottom-up splaying, first we delete the element from the tree and then we perform the splaying on the parent of deleted node.

Top-down splaying

In top-down splaying, we first perform the splaying on which the deletion is to be performed and then delete the node from the tree. Once the element is deleted, we will perform the join operation.

If the element is not present in the splay tree, which is to be deleted, then splaying would be on the last accessed element before reaching the NULL.

In the code as well as algorithm we have implemented the top- down splaying.

Algorithm:

```
If(root==NULL)
return NULL
Splay(root, data)
  If data!= root->data
     Element is not present
  If root->left==NULL
     root=root->right
  else
     temp=root
Splay(root->left, data)
root1->right=root->right
free(temp)
return root
```

In the above algorithm, we first check whether the root is Null or not; if the root is NULL means that the tree is empty.

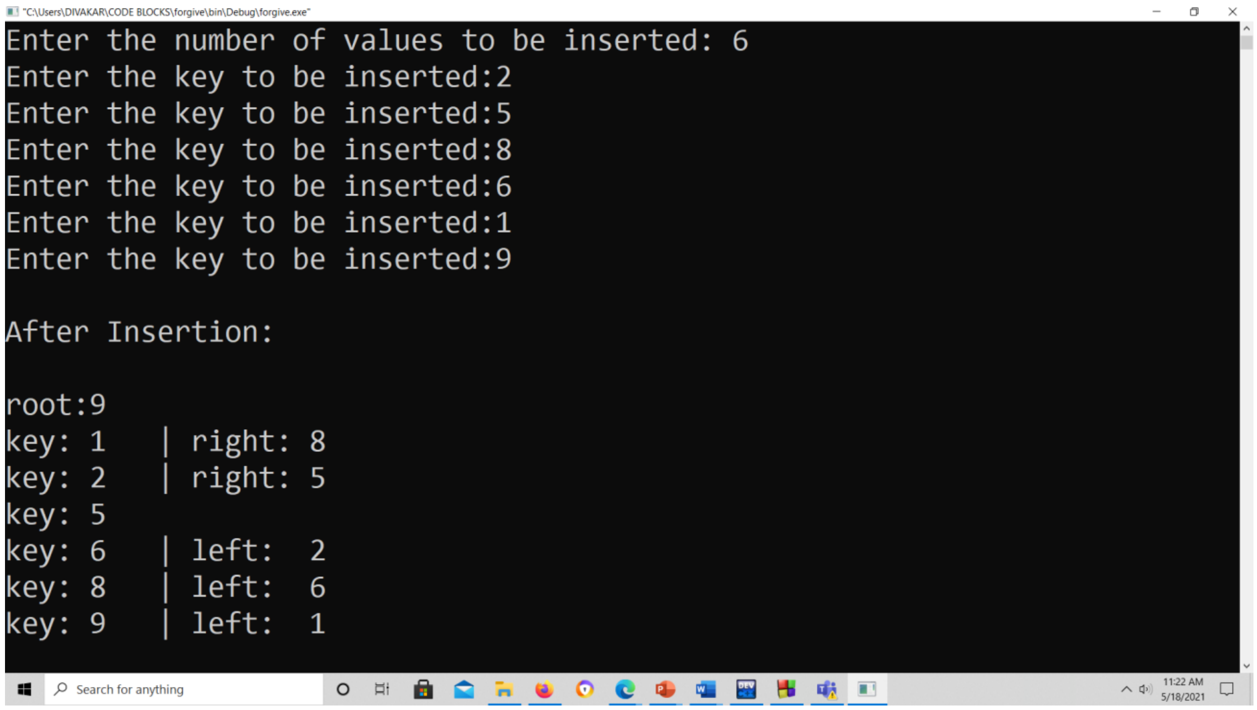
If the tree is not empty, we will perform the splaying operation on the element which is to be deleted.

Once the splaying operation is completed, we will compare the root data with the element which is to be deleted; if both are not equal means that the element is not present in the tree. If they are equal, then the following cases can occur:

Case 1: The left of the root is NULL, the right of the root becomes the root node.

Case 2: If both left and right exist, then we splay the maximum element in the left subtree. When the splaying is completed, the maximum element becomes the root of the left subtree. The right subtree would become the right child of the root of the left subtree.

Examples:


"C:\Users\DIVAKAR\CODE BLOCKS\forgive\bin\Debug\forgive.exe"
Search for anything
11:22 AM
5/18/2021

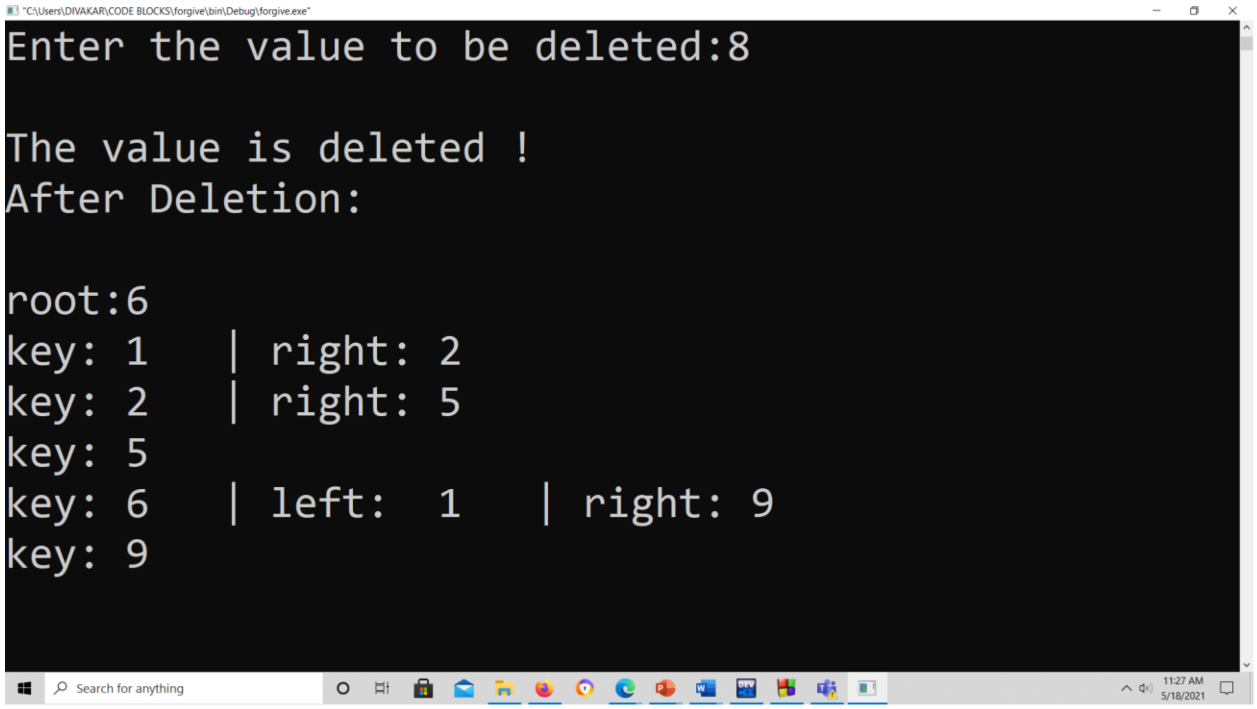
```
Enter the number of values to be inserted: 6
Enter the key to be inserted:2
Enter the key to be inserted:5
Enter the key to be inserted:8
Enter the key to be inserted:6
Enter the key to be inserted:1
Enter the key to be inserted:9

After Insertion:

root:9
key: 1    | right: 8
key: 2    | right: 5
key: 5
key: 6    | left:  2
key: 8    | left:  6
key: 9    | left:  1
```

We have inserted 6 elements into the: 2,5,8,6,1,9

Let us delete the element 8(which is present in the given tree)


"C:\Users\DIVAKAR\CODE BLOCKS\forgive\bin\Debug\forgive.exe"
Search for anything
11:27 AM
5/18/2021

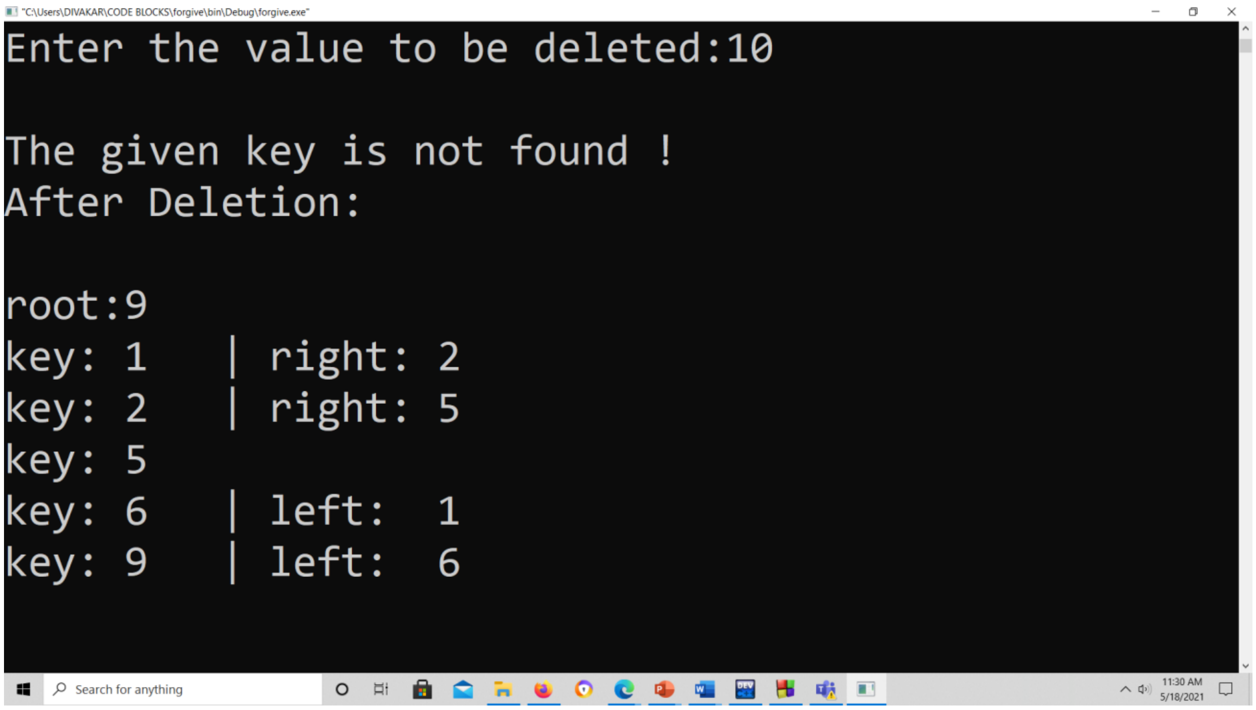
```
Enter the value to be deleted:8

The value is deleted !
After Deletion:

root:6
key: 1    | right: 2
key: 2    | right: 5
key: 5
key: 6    | left:  1    | right: 9
key: 9
```

We can see that the value has been deleted and the splaying operation has been performed according to the top- down approach.

Let us try to delete the element 10 (which is not present in the tree):


```
Enter the value to be deleted:10

The given key is not found !
After Deletion:

root:9
key: 1    | right: 2
key: 2    | right: 5
key: 5
key: 6    | left:  1
key: 9    | left:  6
```

We can see that the node is not present in the tree and it shows "the given key is not found!"

And thus, the splaying operation has been done on the last encountered node which should have been 9-in this case.

Thus, the deletion in the splay trees happen in the above- mentioned manner.

5) UPDATION:

Explanation:

The updation here actually means replacing a value with an -another value.

In the splay trees, which is actually a variation of Binary Search Tree , updation doesn't make a lot of sense as we have to take care of the BST-property as well.

Thus what we have done is, we have made a deletion operation on the old value and then we have inserted the given value , which on its own, will perform splaying operation on the updated value.

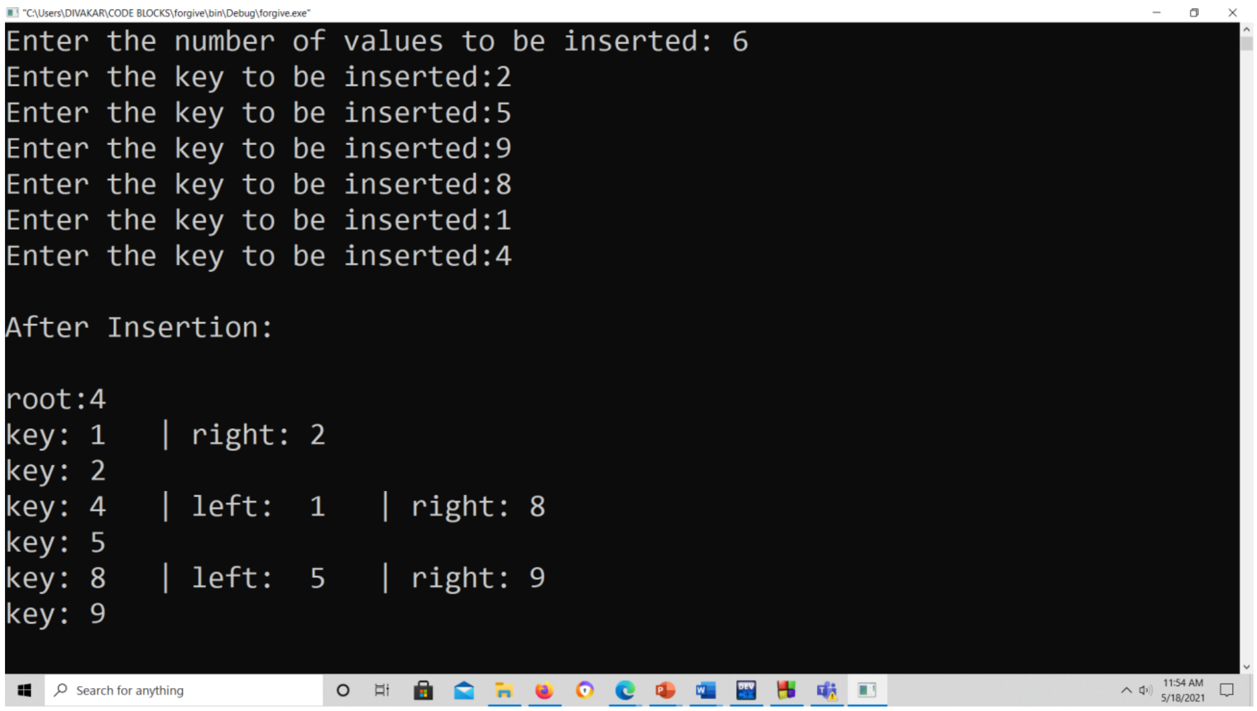
The above procedure has been done ,only in order to keep the BST-property of the tree on hold. We have tried a lot of possibilities , but this was found to be a better option.

Algorithm:

The value x has to be modified into y:

```
if(!Delete(x))    //deletes the element if it is present in the tree
        Print(The element is not found in the tree!)
        return;  //returns back is the node is not present in the tree
    Insert(y); //if x is present and it is deleted, this inserts y.
```

Examples:



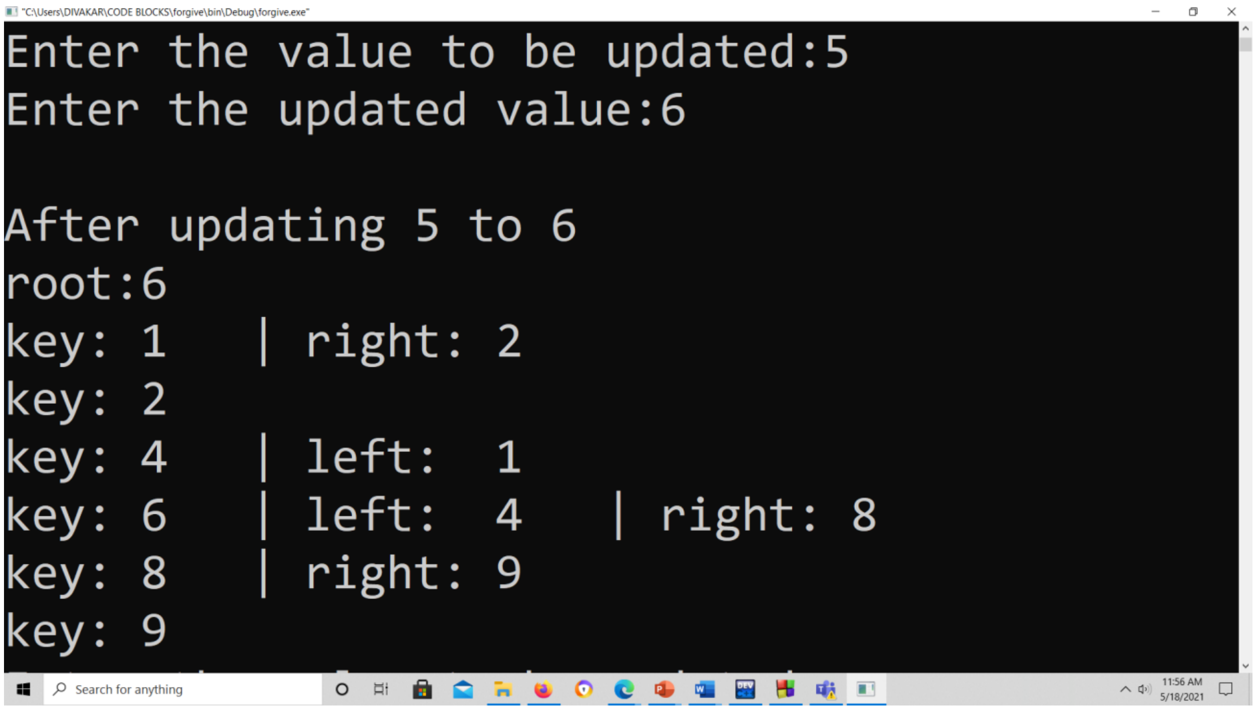
```
Enter the number of values to be inserted: 6
Enter the key to be inserted:2
Enter the key to be inserted:5
Enter the key to be inserted:9
Enter the key to be inserted:8
Enter the key to be inserted:1
Enter the key to be inserted:4

After Insertion:

root:4
key: 1   | right: 2
key: 2
key: 4   | left:  1   | right: 8
key: 5
key: 8   | left:  5   | right: 9
key: 9
```

Six elements have been inserted into the tree: 2,5,9,8,1,4

Let us try to update 5(which is present in the tree) by 6:

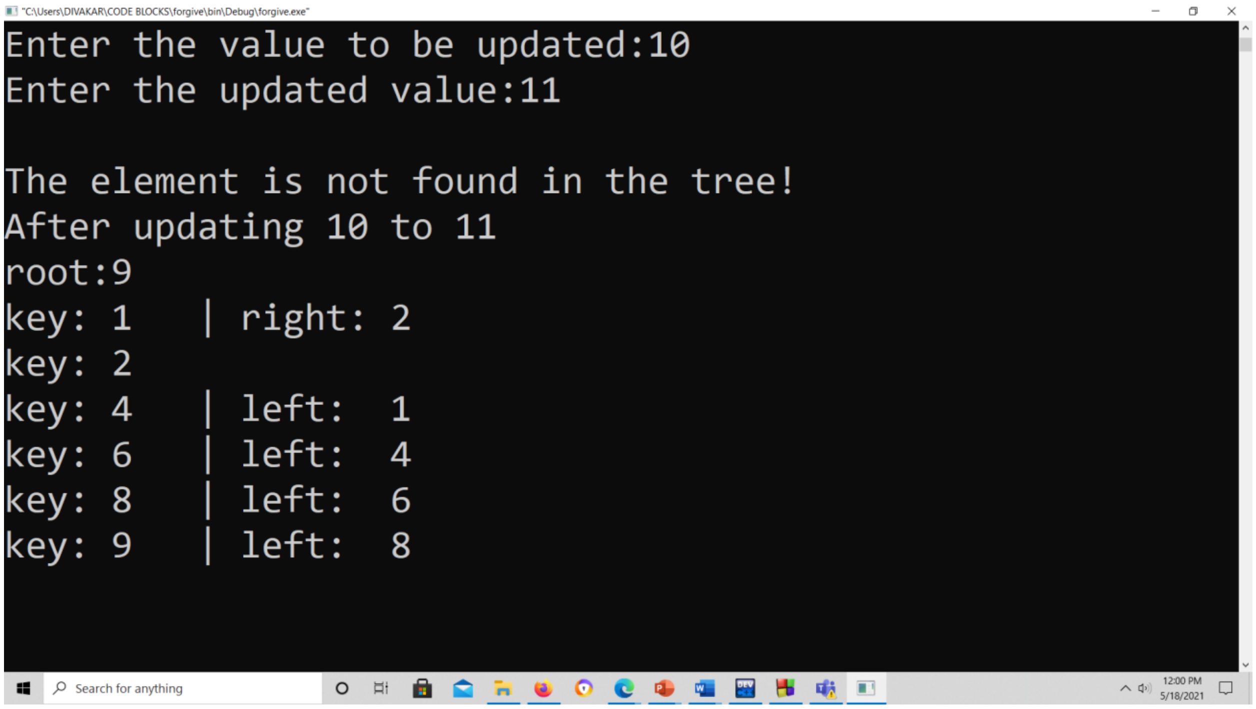


```
Enter the value to be updated:5
Enter the updated value:6

After updating 5 to 6
root:6
key: 1   | right: 2
key: 2
key: 4   | left:  1
key: 6   | left:  4   | right: 8
key: 8   | right: 9
key: 9
```

We can see that the 5 is being replaced by 6 and the splaying operation has been done on the element 6.

Now let us try to update the value 10 (which is not present in the tree) by 11:



```
Enter the value to be updated:10
Enter the updated value:11

The element is not found in the tree!
After updating 10 to 11
root:9
key: 1   | right: 2
key: 2
key: 4   | left:  1
key: 6   | left:  4
key: 8   | left:  6
key: 9   | left:  8
```

We can see that the value 10 is not present in the tree and it says the that "the element is not found in the tree".

But the splaying operation has been done on the value that would have been encountered last while searching for the element 10 (the last element would have been 9, which makes sense) .

TRIE

Description / Definition

The word trie is an infix of the word "retrieval" because the trie can find a single word in a dictionary with only a prefix of the word.

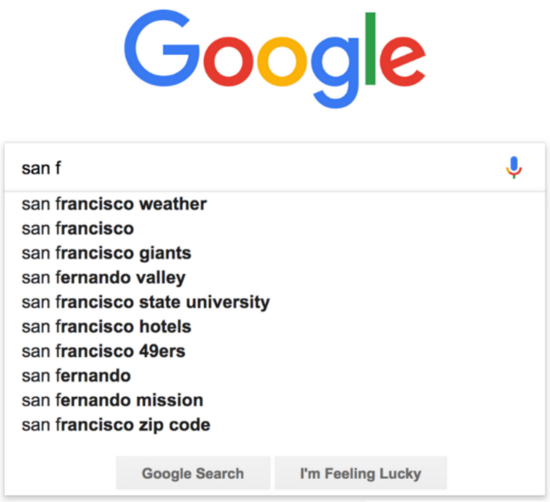
The main idea of the trie data structure consists of the following parts:

- The trie is a tree where each vertex represents a single word or a prefix.
- The root represents an empty string (""), the vertexes that are direct sons of the root represent prefixes of length 1, the vertexes that are 2 edges of distance from the root represent prefixes of length 2, the vertexes that are 3 edges of distance from the root represent prefixes of length 3 and so on. In other words, a vertex that are *k* edges of distance of the root have an associated prefix of length *k*.
- Let v and w be two vertexes of the trie, and assume that v is a direct father of w, then v must have an associated prefix of w.

Real Time Application / Where it is used ?

There are various applications of this very efficient data structure such as :

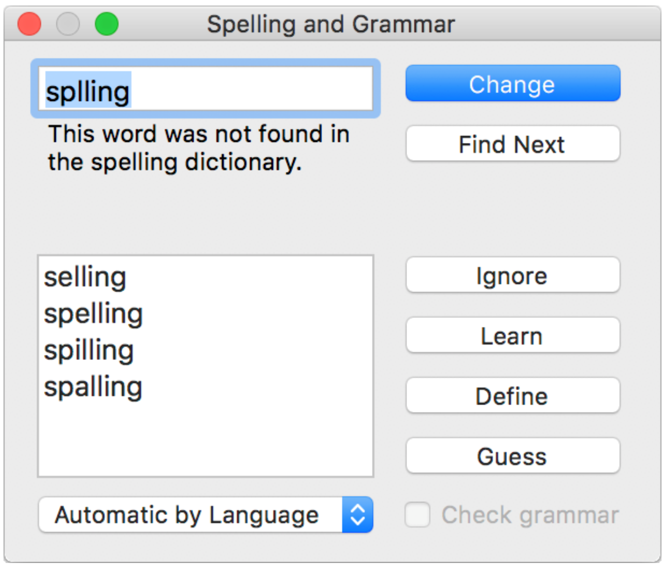
1. Auto Complete



san f

san **francisco weather**
san **francisco**
san **francisco giants**
san **fernando valley**
san **francisco state university**
san **francisco hotels**
san **francisco 49ers**
san **fernando**
san **fernando mission**
san **francisco zip code**

3. Spell Checker



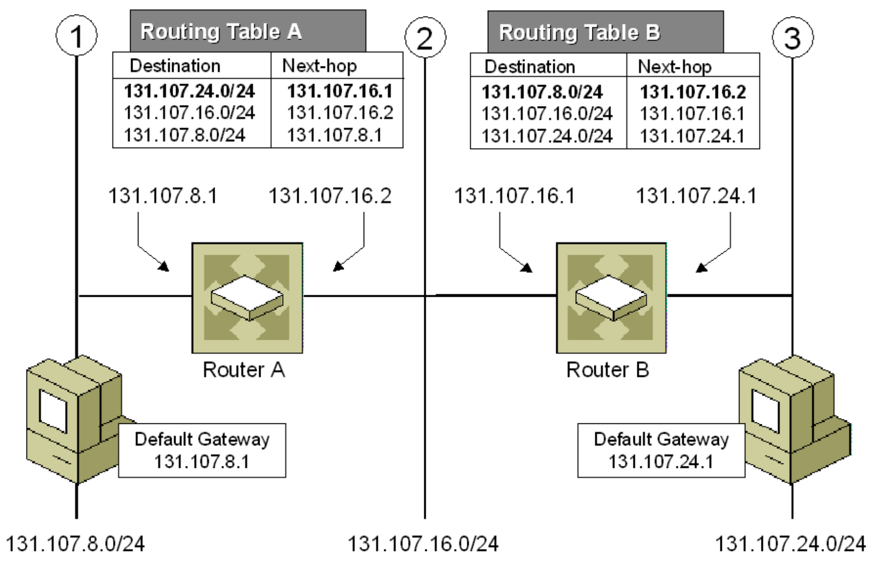
spling

This word was not found in the spelling dictionary.

selling
spelling
spilling
spalling

Automatic by Language

3. IP routing (Longest prefix matching)



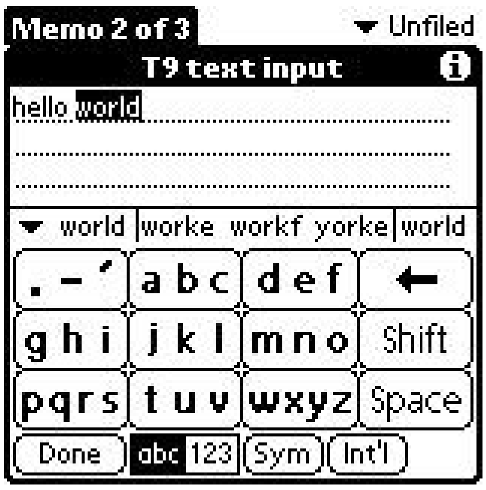
Routing Table A

Destination	Next-hop
131.107.24.0/24	**131.107.16.1**
131.107.16.0/24	131.107.16.2
131.107.8.0/24	131.107.8.1

Routing Table B

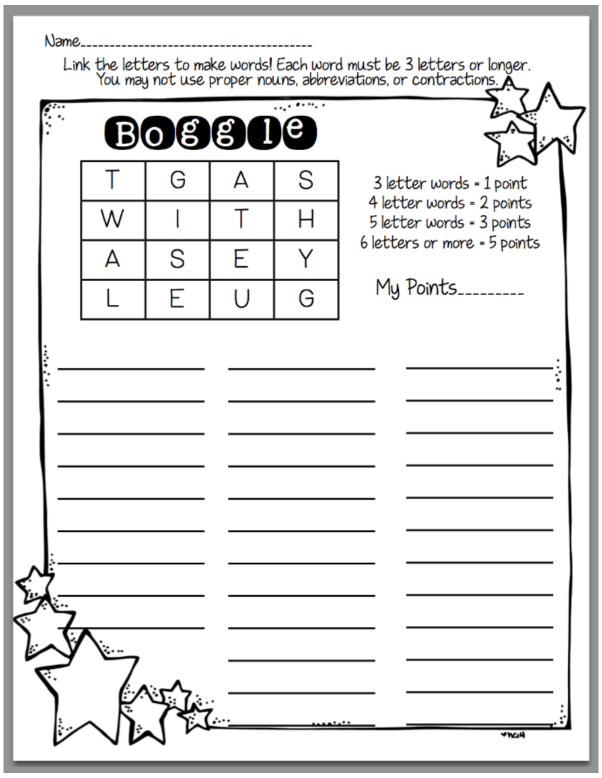
Destination	Next-hop
131.107.8.0/24	**131.107.16.2**
131.107.16.0/24	131.107.16.1
131.107.24.0/24	131.107.24.1

4. T9 predictive text



hello world

5. Solving word games


Name_______________________________

Link the letters to make words! Each word must be 3 letters or longer.
You may not use proper nouns, abbreviations, or contractions.

Boggle

T	G	A	S
W	I	T	H
A	S	E	Y
L	E	U	G

3 letter words = 1 point
4 letter words = 2 points
5 letter words = 3 points
6 letters or more = 5 points

My Points________

Requirements / limitations.

The main disadvantage of tries is that they need a lot of memory for storing the strings. For each node we have too many node pointers (equal to number of characters of the alphabet), if space is concerned, then Ternary Search Tree can be preferred for dictionary implementations. In Ternary Search Tree, the time complexity of search operation is O(h) where h is the height of the tree. Ternary Search Trees also supports other operations supported by Trie like prefix search, alphabetical order printing, and nearest neighbour search.
The conclusion is regarding *tries data structure* is that they are faster but require *huge memory* for storing the strings.

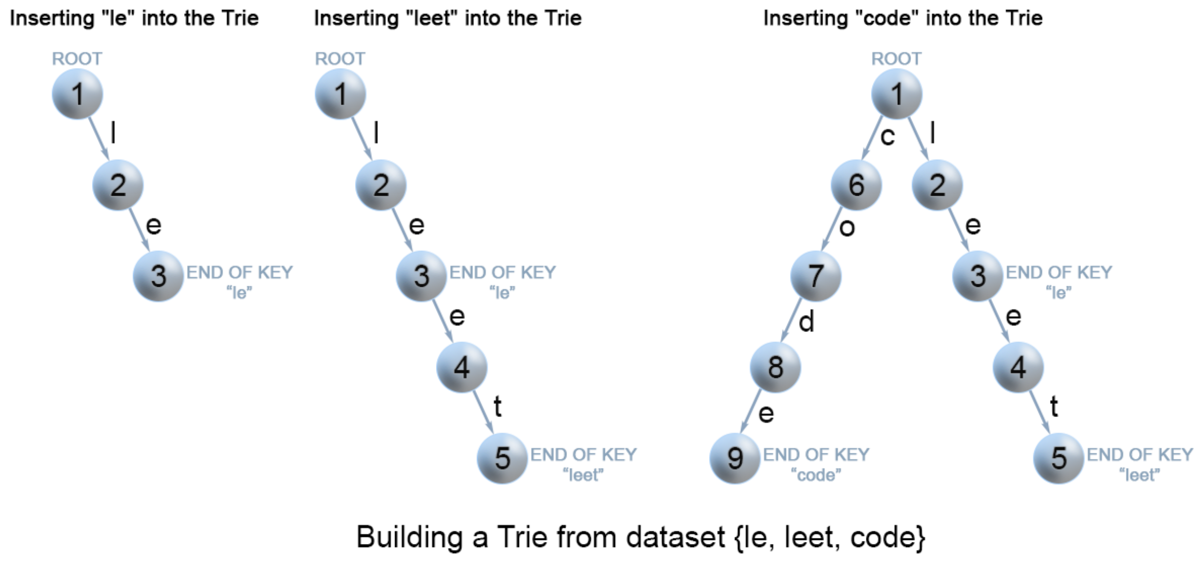
Operations

1. Insertion of a key to a trie

We insert a key by searching into the trie. We start from the root and search a link, which corresponds to the first key character. There are two cases :

- A link exists. Then we move down the tree following the link to the next child level. The algorithm continues with searching for the next key character.

- A link does not exist. Then we create a new node and link it with the parent's link matching the current key character. We repeat this step until we encounter the last character of the key, then we mark the current node as an end node and the algorithm finishes.



Inserting "le" into the Trie
Inserting "leet" into the Trie
Inserting "code" into the Trie
ROOT
1
l
2
e
3
END OF KEY "le"
4
t
5
END OF KEY "leet"
c
6
o
7
d
8
9
END OF KEY "code"
Building a Trie from dataset {le, leet, code}

Insertion of keys into a trie.

Code/Algorithm

```
void insert(string word) {
        Trie* node = this;
        for (char ch : word) {
            ch -= 'a';
            if (!node->next[ch])
                node->next[ch] = new Trie();
            node->cnt++;
            node = node->next[ch];
        }
        node->isEndOfWord = true;
    }
```

Complexity Analysis

- Time complexity : O(m) where m is the key length.

In each iteration of the algorithm, we either examine or create a node in the trie till we reach the end of the key. This takes only m operations.

- Space complexity : O(m)

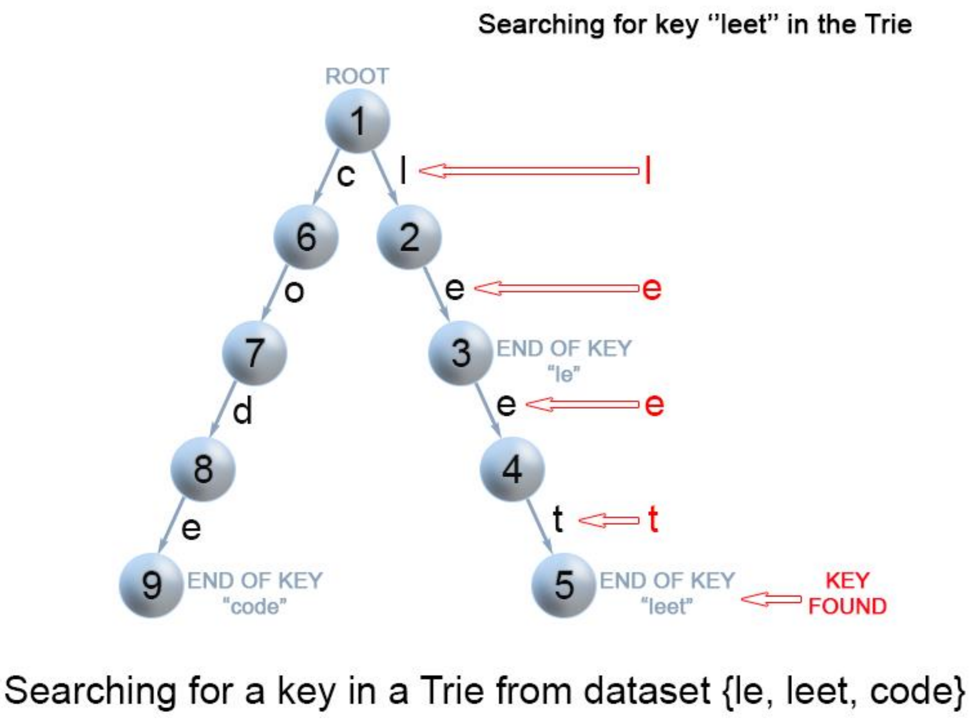
In the worst case newly, inserted key doesn't share a prefix with the keys already inserted in the trie. We have to add mm new nodes, which takes us O(m) $O(m)$ space.

2. Search for a key in a trie

Each key is represented in the trie as a path from the root to the internal node or leaf. We start from the root with the first key character. We examine the current node for a link corresponding to the key character. There are two cases :

- A link exists. We move to the next node in the path following this link and proceed searching for the next key character.

- A link does not exist. If there are no available key characters and current node is marked as isEnd we return true. Otherwise, there are possible two cases in each of them we return false :
  - There are key characters left, but it is impossible to follow the key path in the trie, and the key is missing.
  - No key characters left, but current node is not marked as isEnd. Therefore, the search key is only a prefix of another key in the trie.


Searching for key "leet" in the Trie
ROOT
1
c
l
l
6
2
o
e
e
7
3
END OF KEY "le"
d
e
e
8
4
e
t
t
9
END OF KEY "code"
5
END OF KEY "leet"
KEY FOUND
Searching for a key in a Trie from dataset {le, leet, code}

. Search for a key in a trie.

Code/Algorithm

```
bool search(string word) {
      Trie* node = this;
      for (char ch : word) {
         ch -= 'a';
         if (!node->next[ch]) { return false; }
         node = node->next[ch];
      }
      return node->isEndOfWord;
  }
```

Complexity Analysis

- Time complexity : O(m) In each step of the algorithm we search for the next key character. In the worst case the algorithm performs m*m* operations.
- Space complexity : O(1)

3. Search for a key prefix in a trie

The approach is very similar to the one we used for searching a key in a trie. We traverse the trie from the root, till there are no characters left in key prefix or it is impossible to continue the path in the trie with the current key character. The only difference with the mentioned above search for a key algorithm is that when we come to an end of the key prefix, we always return true. We don't need to consider the isEnd mark of the current trie node, because we are searching for a prefix of a key, not for a whole key.

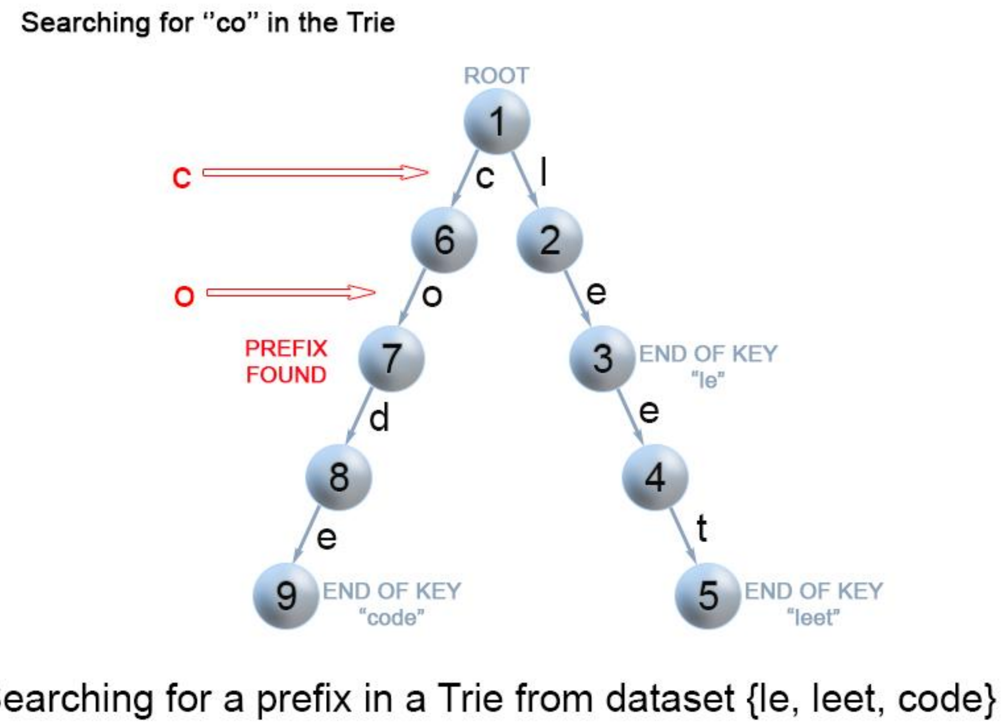

Searching for "co" in the Trie
ROOT
1
c
l
c
6
2
o
o
e
PREFIX FOUND
7
3
END OF KEY "le"
d
e
8
4
e
t
9
END OF KEY "code"
5
END OF KEY "leet"
Searching for a prefix in a Trie from dataset {le, leet, code}

Figure 9. Search for a key prefix in a trie.

Code / Algorithm

```
int startsWith(string prefix) {
        Trie* node = this;
        for (char ch : prefix) {
            ch -= 'a';
            if (!node->next[ch]) { return 0; }
            node = node->next[ch];
        }
        return node->cnt;
    }
```

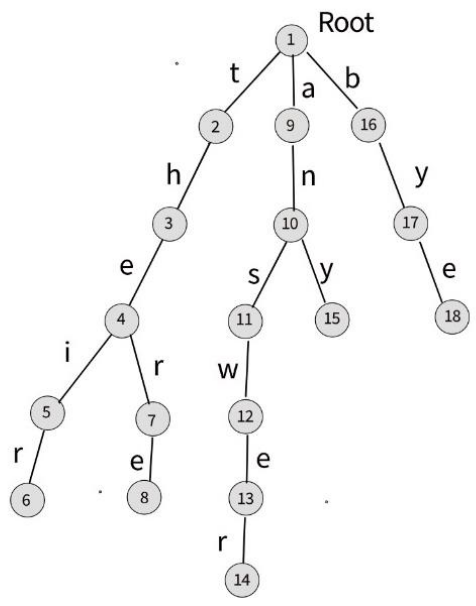
Complexity Analysis

- Time complexity : O(m)
- Space complexity : O(1)

4. Display

The idea to do this is to start traversing from the root node of trie, whenever we find a NON-NULL child node, we add parent key of child node in the "string str" at the current index(level) and then recursively call the same process for the child node and same goes on till we find the node which is a leaf node, which actually marks the end of the string.

If Trie is:



Contents of Trie:

answer

any

bye

their

there

Code/Algorithm

```
void display(Trie* root, char str[], int level = 0)
    {
        if (root->isEndOfWord)
        {
            str[level] = '\0';
            cout << str << endl;
        }
        for (int i = 0; i < ALPHABET_SIZE; i++)
        {
            if (root->next[i])
            {
                str[level] = i + 'a';
                display(root->next[i], str, level + 1);
            }
        }
    }
```

5. Delete

During delete operation we delete the key in bottom-up manner using recursion. The following are possible conditions when deleting key from trie,

1. Key may not be there in trie. Delete operation should not modify trie.

2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.
3. Key is prefix key of another long key in trie. Unmark the leaf node.
4. Key present in trie, having at least one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

Code/Algorithm

```
Trie* remove(Trie* root, string word, int depth = 0)
  {
      if (!root)
          return NULL;
      if (depth == word.size()) {
          if (root->isEndOfWord)
              root->isEndOfWord = false;
          if (isEmpty(root)) {
              delete (root);
              root = NULL;
          }

          return root;
      }

      int index = word[depth] - 'a';
      root->next[index] = remove(root->next[index], word, depth + 1);

      if (isEmpty(root) && root->isEndOfWord == false) {
          delete (root);
          root = NULL;
      }
      return root;
  }
```

Fenwick Trees

Description / Definition

Let, f be some *reversible* function and A be an array of integers of length N.

Fenwick tree is a data structure which:

- calculates the value of function f in the given range [l,r] (i.e. f(Al,Al+1,...,Ar)) in O(logn) time
- updates the value of an element of A in O(logn) time;
- requires O(N) memory, or in other words, exactly the same memory required for A;
- is easy to use and code, especially, in the case of multidimensional arrays.

Fenwick tree is also called Binary Indexed Tree, or just BIT abbreviated.

The most common application of Fenwick tree is *calculating the sum of a range* (i.e. f(A1,A2,...,Ak)=A1+A2+⋯+Ak).

Fenwick tree was first described in a paper titled "A new data structure for cumulative frequency tables" (Peter M. Fenwick, 1994).

Working

Every Index i in the BIT[] array stores the cumulative sum from the index i to i - (1<<r) + 1 (both inclusive), where r represents the last set bit in the index i.

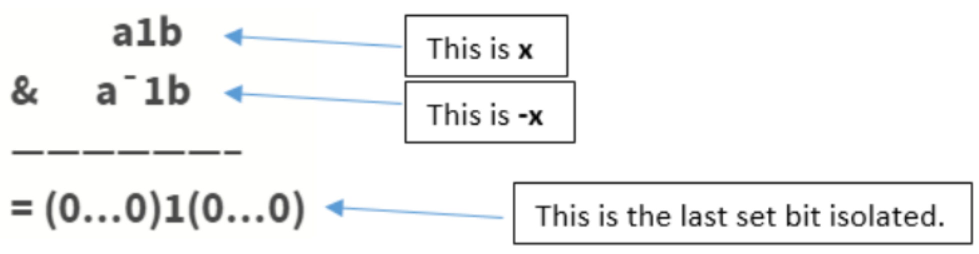
Also, We have a clean way of calculating (1<<r) which is required above.

If r is the last set bit of i, then:

(1<<r)=i&-i

How? :

-x = 2’s complement of x = (a1b)’ + 1 = a’0b’ + 1 = a’0(0….0)’ + 1 = a’0(1…1) + 1 = a’1(0…0) = a’1b



Example: x = 10(in decimal) = 1010(in binary)

The last set bit is given by x&(-x) = (10)1(0) & (01)1(0) = 0010 = 2(in decimal)

Applications of Fenwick tree

- Fenwick tree are used to implement the arithmetic coding compression algorithm.
- Fenwick Tree can be used to count inversions in an array in O(nlogn) time.
- Sum/XOR/Reversible Operations Sum of range can be computed with Updates using Fenwick tree efficiently.
- In Real Life, Software involving above functions require using Fenwick Trees.

Requirements / Limitations

1) Only Point/Range Queries can be answered using the Data structure

2) There’s no Pre-requisite for the Sequence of Numbers to be provided.

3) It can only operate for Operations which are associative in Nature (because Fenwick Tree cannot handle Inverse Operations which means Updates will not be proper).

4) It is stored as an integer array not like some tree structure.

NOTE:

For all examples to come, assume the Original Array to be as follows for simplicity:

int a[] = {0,1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};

And the operation targeted is addition (works for all ass).

Operation 1 (Point Update)

Explanation:

Here, we want to Increment an index by Some Value. In the fashion that we store the Fenwick Tree, for a single index in our original Array, we have to update all the indices that are responsible for that index, which are the indices we encounter while consecutively adding index's last set bit's value till N starting from index to be updated in original array.

Note that, maximum number of indices required to be visited are log(N) because the last set bit of index keeps shifting towards left in the procedure until getting bigger than N.

Hence, Time Complexity : O(log(N))

Algorithm:

```
void update(int idx, int delta, vector<int> bit) {
        // All Appropriate Indices until n are updated
      // Last Set bit's Value is added in current idx to obtain next idx
      for (; idx <= n; idx += idx & -idx)
          bit[idx] += delta;
  }
```

Example :

Let’s update index 13 and increment it by 2.

Note that for index 13, Following indices in bit are responsible(have its value):

13 => bit[13] = a[13]

14 => bit[14] = a[13]+a[14]

16 => bit[16] = a[1]+a[2]+……+a[15]+a[16]

So, we want to increment these indices in Fenwick Tree(bit array)

We obtain these indices by starting with 13.

Then we reach index (13+last set bit’s value of 13)

- ⇨ 13+1=14

Then from 14 we reach

- ⇨ 14+2=16

From 16, we reach 32 which is beyond the end of array.

We increment all these indices by 2 (desired increment).

Operation 2 (Range Query)

Explanation:

To answer a Range Sum Query from index l to r, we can write it as difference of prefix sum until r and prefix sum until l-1(included). In the fashion that we store the Fenwick Tree, To obtain to the prefix sum till some index, we only need summation of those indices in Fenwick Tree which combined together cover the whole range from 0 to that index (which are the indices we encounter while consecutively subtracting index’s last set bit’s value till 0 starting from the index for which Prefix Sum is required.)

Note that, the number of indices to be visited for a prefix sum query are equal to the number of set bits in Binary Representation of index (for which Query was made). And In Range Query, 2 Prefix Query are performed.

Hence, Time Complexity : O(log(N))

Algorithm

```
int PrefixQuery(int idx, vector<int> bit) {
    int ret = 0;
    for (; idx > 0; idx -= idx & -idx)
        ret += bit[idx];
    return ret;
}

int query(int l, int r) {
    return PrefixQuery(r) - PrefixQuery(l - 1);
}
```

Example:

Let's find Range Sum in range: [2,14].

Range Sum [2,14] = Prefix Sum(14) - Prefix Sum(1)

Following indices in bit complete the Prefix Sum up to index 14:

14 => bit[14] = a[13]+a[14]

Then we reach index (14-last set bit's value of 14) :

- ⇨ 14-2=12

12 => bit[12] = a[9]+a[10]+a[11]+a[12]

Then from 12 we reach :

- ⇨ 12-4=8

8 => bit[8] = a[1]+a[2]+……+a[7]+a[8]

From 8, we reach 0 which is not above 0.

Prefix Sum(14) = bit[14]+bit[12]+bit[8] = 105

Similarly, To Calculate Prefix Sum (1)

1 => bit[1] = a[1]

From 1, we again reach 0.

Hence, Prefix Sum(1) = bit[1] =1

Range Sum[2,14] = Prefix Sum(14)- Prefix Sum(1)

= 105 -1 = 104

Operation 3 (Range Update but for Point Queries)

Explanation:

Let the Fenwick tree be initialized with zeros. Suppose that we want to increment the interval [l,r] by x. We make 2 Point Update operations on Fenwick tree which are add(l, x) and add(r+1, -x).

If we want to get the value of a[i], we just need to take the prefix sum using the ordinary range sum method (and also add original value later). To see why this is true, we can just focus on the previous increment operation again.

- ⇨ If i<l, then the two update operations have no effect on the query and we get the sum 0.
- ⇨ If i∈[l,r], then we get the answer x because of the first update operation.
- ⇨ And if i>r, then the second update operation will cancel the effect of first one.

Notice that, as we perform only 2 Point Updates which have O(log(N)) complexity. This has the same Time Complexity!

Algorithm:

```
void RangeUpdatePQ(int l, int r, int delta) {
    update(l, delta);
    update(r + 1, -delta);
}
```

Example:

Suppose we want to increment range [2,5] in our Array by 7.

Then, we just increment index 2 by 7 and,

Decrement index 5+1=6 by 7.

Now, Suppose we ask Point Queries:

a) Index 1

    answer[1] = a[1] + Prefix Query(1)
              = 1 + 0 = 1

b) Index 4

    answer[4] = a[4] + Prefix Query(4)
              = 4 + 7
              = 11

c) Index 8

    answer[8] = a[8] + Prefix Query(8)
              = 8 + (7-7)
              = 8

As we can see, this method does the trick for All Cases!

Operation 4 (Range Update while handling Range Queries)

Explanation:

To support both range updates and range queries we will use two BITs namely B1[] and B2[], initialized with zeros.

Suppose that we want to increment the interval [l,r] by the value x. Similarly as in the previous method, we perform two point updates on B1: add(B1, l, x) and add(B1, r+1, -x). And we also update B2 cleverly, such that after updates in both places:

$$sum[0,i]=sum(B1,i)\cdot i-sum(B2,i)$$

Time Complexity : O(log(N))

(Because, Just Constant Number of Logarithmic Operations are required)

Algorithm:

```
void RangeUpdateRQRU(int l, int r, int val) {
    update(l, val,bit);
    update(r + 1, -val,bit);
    update(l, (l - 1)*val, bit2);
    update(r + 1, -r * val, bit2);
}
```

Example:

If we want to increment [3,5] by 6,

We have to do 4 point updates, similar to the previous trick used:

In First Bitset:

- ⇨ Increase index 3 by 6
- ⇨ Decrease index 5+1 by 6

In Second Bitset:

- ⇨ Increase index 3 by 2*6
- ⇨ Increase index 5+1 by 5*6

Operation 5 (Range Query while handling Range Updates)

Explanation:

After the range update (l,r,x) the range sum query should return the following values:

sum[0,i] =	0	when $i<l$
sum[0,i] =	$x\cdot(i-(l-1))$	when $l<=i<=r$
sum[0,i] =	$x\cdot(r-l+1)$	when $i>r$

We can write the range sum as difference of two terms, where we use B1 for first term and B2 for second term. The difference of the queries will give us prefix sum over [0,i].

sum[0,i] =	$sum(B1,i)\cdot i-sum(B2,i)$	(handles all the cases)
sum[0,i] =	$0\cdot i-0$	when $i<l$
sum[0,i] =	$x\cdot i-x\cdot(l-1)$	when $l<=i<=r$
sum[0,i] =	$0\cdot i-(x\cdot(l-1)-x\cdot r)$	when $i>r$

The last expression is exactly equal to the required terms. Thus we can use B2 for shaving off extra terms when we multiply $B1[i]\times i$.

We can find arbitrary range sums by computing the prefix sums for $l-1$ and r and taking the difference of them again.

Time Complexity : O(log(N))

(Because, Just Constant Number of Logarithmic Operations are required)

Algorithm:

```
int PrefixQueryRQRU(int idx, vector<int> b) {
    return PrefixQuery(idx, bit) * idx -  PrefixQuery(idx, bit2);
}

int RangeQueryRQRU(int l, int r) {
    return PrefixQueryRQRU(r) - PrefixQueryRQRU(l - 1);
}
```

In the code above, RQRU stands for Range Query and Range Update denoting that these are specifically for those purposes.

Example:

Suppose after the previous range update (increment [3,5] by 6), We want to query the range sum between [2,4], We do it as follows:

Range QueryRQRU[1,4] = Prefix QueryRQRU(4) - Prefix QueryRQRU(1)

Let's calculate Prefix QueryRQRU(4):

Prefix QueryRQRU(4) = Prefix Query(4, bit) * 4 - Prefix Query(4, bit2)

= 6*4 – 12

= 12

Notice we got 12 as increase, which is the same as effective increase in range [3,4] due to previous Range Update performed.

Now, we just need to subtract Prefix QueryRQRU(1) from 12:

Prefix QueryRQRU(1) = Prefix Query(1, bit) * 1 - Prefix Query(1, bit2)

= 0*1 – 0

= 0

Hence, net sum of changes due to updates was 12-0=12 in the Queried change.

The Final Answer, However, will be the initial sum of the range in Original Array combined with this net sum of changes:

Initial Sum[2,4] = 2+3+4 = 9

Range QueryRQRU[2,4] = 9+12-0 = 21

If we visualize the Original Array with Updates, We get:

```
int a[] = {0,1, 2, 9, 10, 11, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
                   ^   ^   ^
```

As we can see, We Queried at the Marked Points whose sum=21 which is what we got!

Summary Table

	Application	Insert	Delete	Search /Query	Update	Special Operation
Splay Tree	Network router, Intrusion detection systems (IDS), Cache Memory.	BST insertion following by Splaying. **Time** : O(log n)	**Top-Down** : Splaying of key followed by removal of root and merging Sub-trees. **Time :** O(log n)	BST Searching followed by Splaying **Time :** O(log n)	Search for Key then Delete the Key and Insertion. **Time :** O(log n)	**Splaying Operation:** Last Operated Node become the Root. **Time :** O(log n)
Trie	Auto Complete, Spell Checker , IP routing (Longest prefix matching), T9 predictive text.	Searching into the trie character wise. **Time :** O(key-length)	In bottom-up manner using recursion. **Time :** O(key-length)	Go Char wise Return true if node after last char of key is marked with **isEnd.** **Time :** O(key-length)	Preserve common Prefix and replace the remaining key. **Time :** O(key-length)	**Prefix Search** Searching Trie for Words with Common prefix. **Time :** O(key-length)
Fenwick Tree	Arithmetic Coding Compression Algorithm, Software requiring efficient range operations.	**Update** by that value which was initially zero. **Time** : O(log n)	**Update** by Adding Inverse (Operation Specific) of current value. **Time** : O(log n)	**Point and Range Query**: Sum (Operation Specific) for index/segment of Array **Time** : O(log n)	Update by adding Difference of new and previous value. **Time** : O(log n)	Efficient Cumulative Sum for Operations with Inverse.(**+** , **XOR**) **Time** : O(log n)

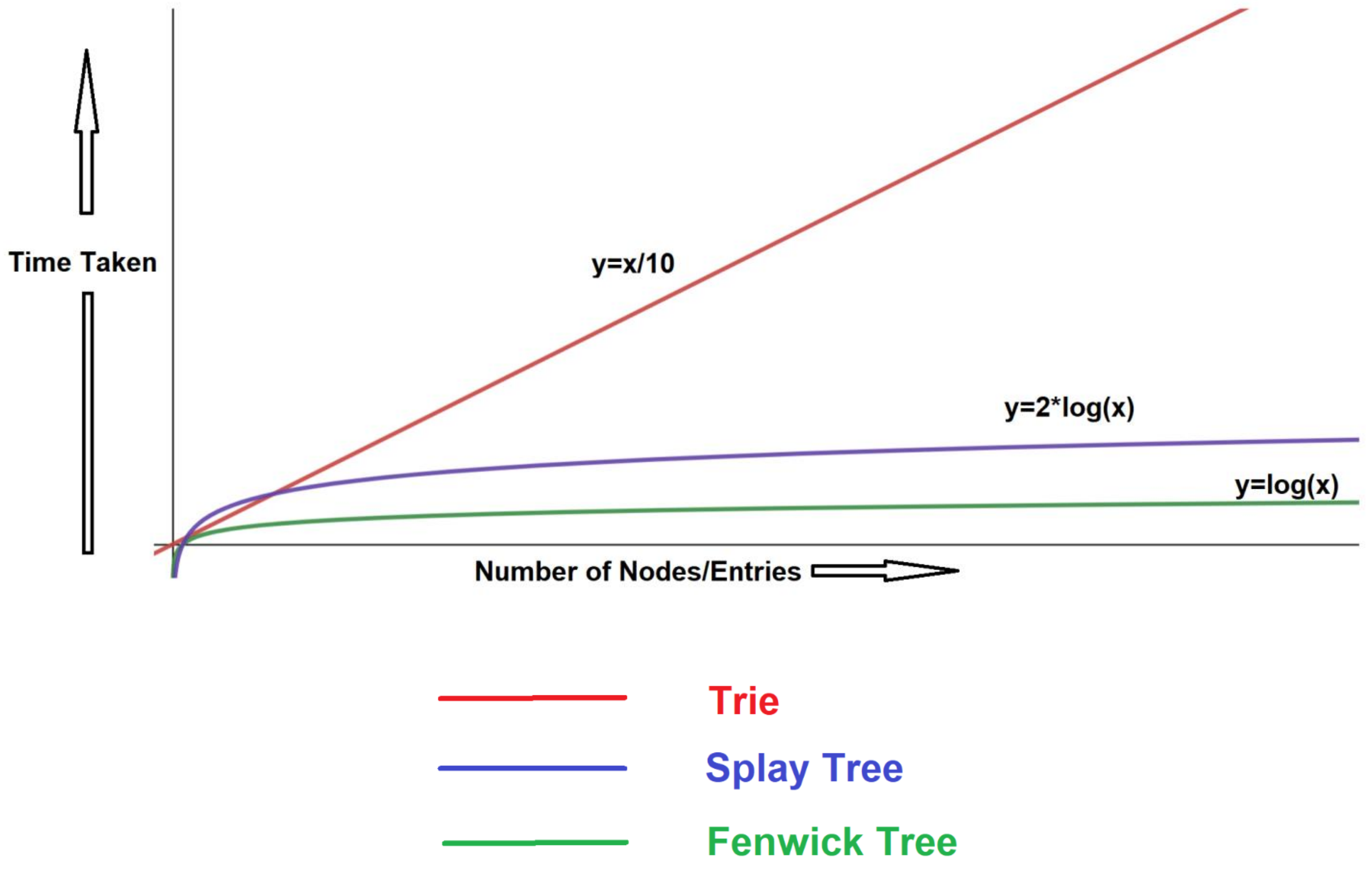
NOTE:

Assuming Average Word to be of length 10, We equated 10 length word in Trie (approx.) with 1 node in Fenwick/Splay Tree for Comparison in all the following Operations.

For Tries, X-axis denotes the Length of Word being operated.

Performance comparison

1) INSERTION



Performance:

Time Complexity:

Trie : O(w) where w is the length of word.

Splay Tree : 2*log(n) (because of Positioning of New Node and Splaying) which makes the Overall Time Complexity O(log(n)) where n are the number of Nodes

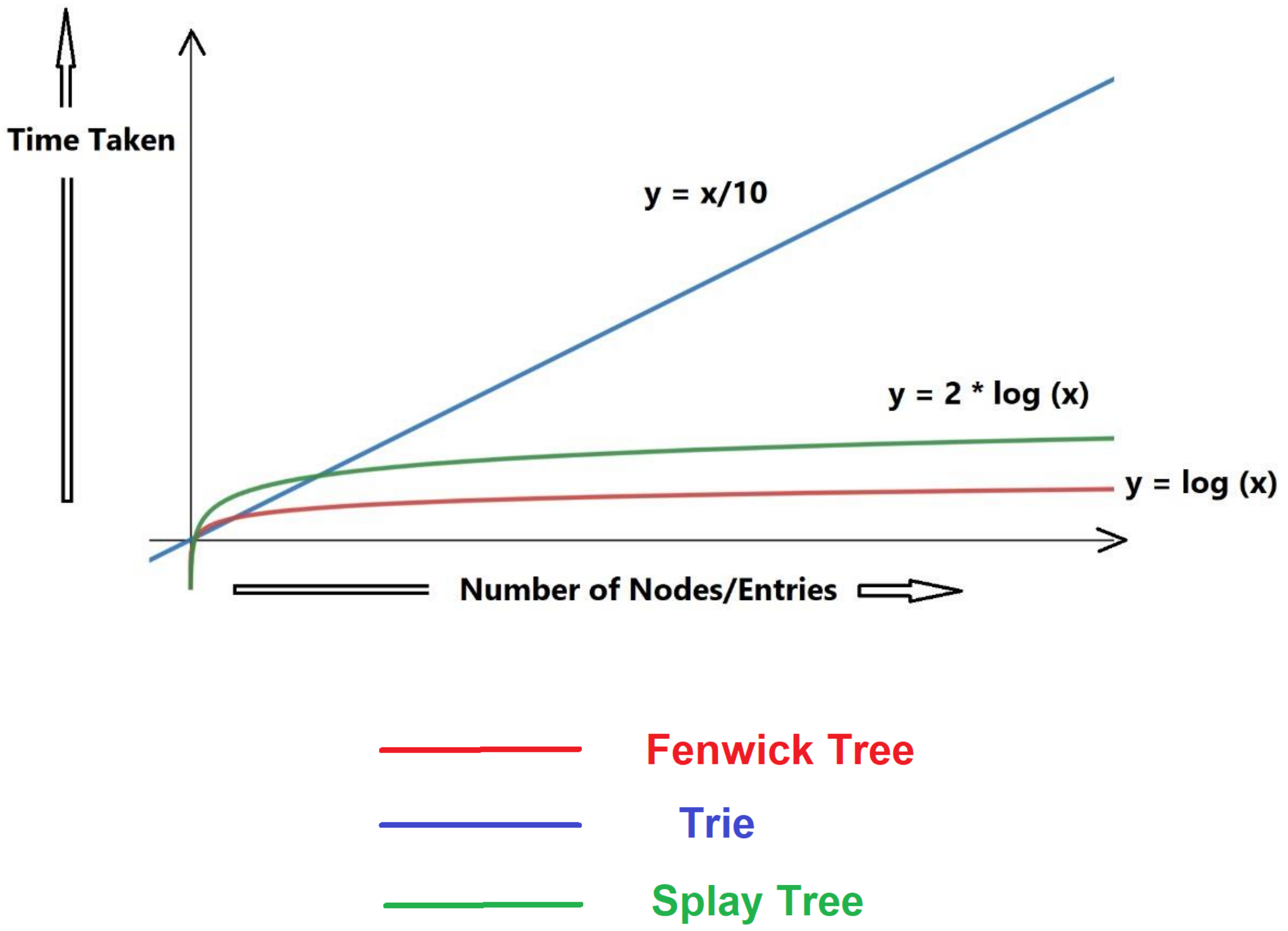
Fenwick Tree : Just 1 Point Update is called, leading to O(log(n)) Time Complexity where n are the number of Nodes

Inference

Time Complexity Wise:

Splay Tree Insertion ≈ Fenwick Tree Insertion < Trie Insertion

2) SEARCH / QUERY



Performance:

Time Complexity:

Trie : O(w) where w is the length of word.

Splay Tree : 2*log(n) (because of searching the new node and Splaying) which makes the Overall Time Complexity O(log(n)) where n are the number of Nodes.

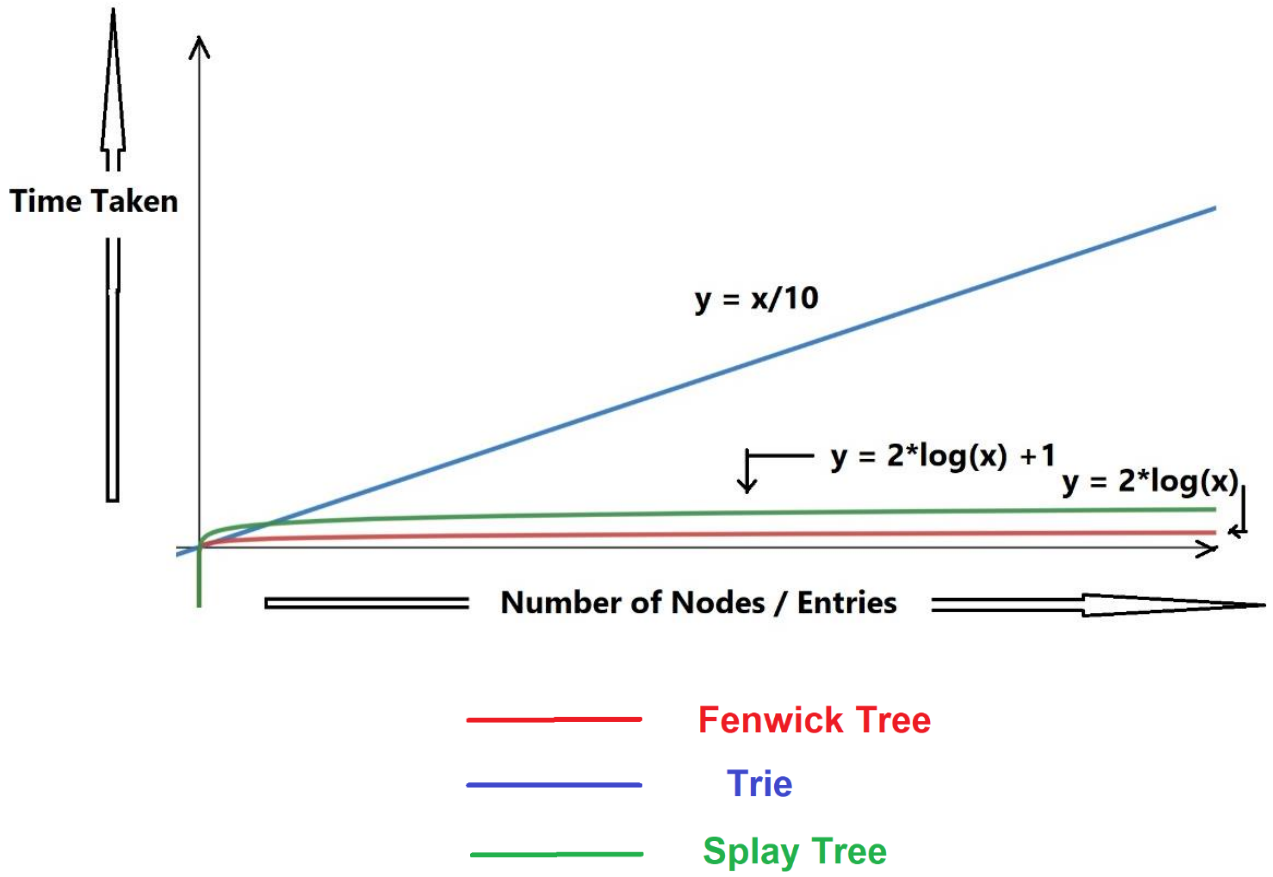
Fenwick Tree : For a Range Query, 2 Prefix Queries are called, again leading to O(log(n)) Time Complexity where n are the number of Nodes.

Inference

Time Complexity Wise:

Splay Tree Searching ≈ Fenwick Tree Searching < Trie Searching

3) Deletion



Performance:

Time Complexity:

Trie : Traverse the Trie recursively and remove the concerned edges(links) leading to Overall Time Complexity as O(w) where w is the length of word.

Splay Tree : 2*log(n)+1 (because of searching the Node to be deleted and Splaying and Merging Subtrees) which makes the Overall Time Complexity O(log(n)) where n are the number of Nodes.

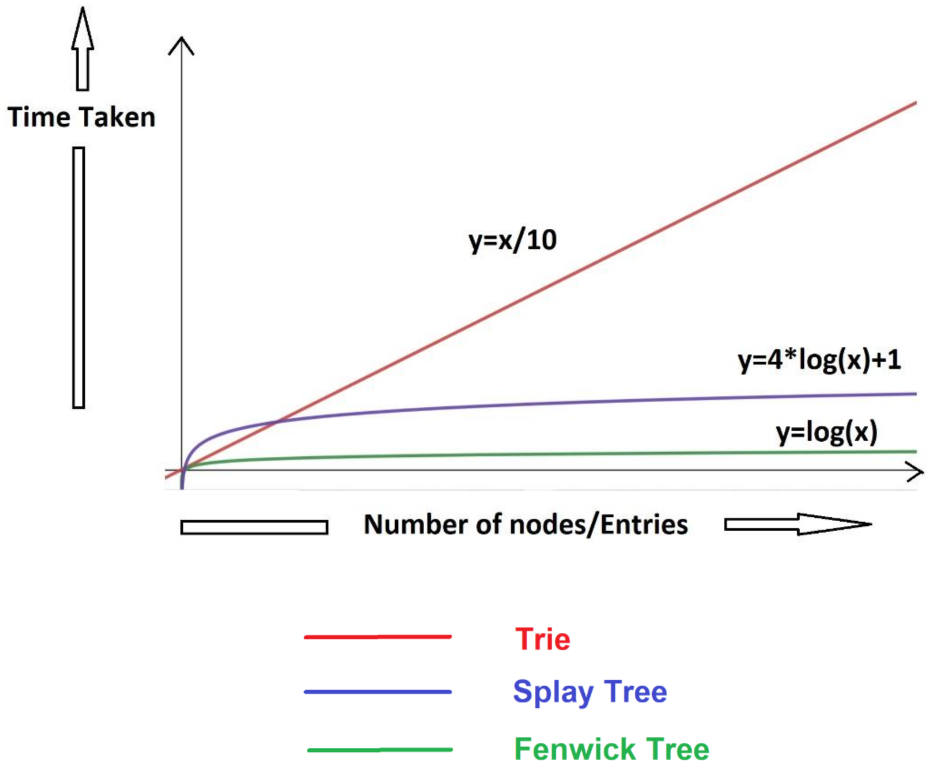
Fenwick Tree : For Deletion, The value concerned is updated by an increment of inverse of that Value itself, i.e. just 1 Point Update, again leading to O(log(n)) Time Complexity where n are the number of Nodes.

Inference

Time Complexity Wise:

Splay Tree Deletion ≈ Fenwick Tree Deletion < Trie Deletion

4) Update



Performance:

Time Complexity:

Trie : Except the Common Prefix in old and new word, replace the remaining of old word to new links again leading to O(w) Time Complexity in Worst Case where w is the length of word.

Splay Tree : 4*log(n) (because of Insertion of New Node and Deletion of Old Value) which makes the Overall Time Complexity O(log(n)) where n are the number of Nodes.

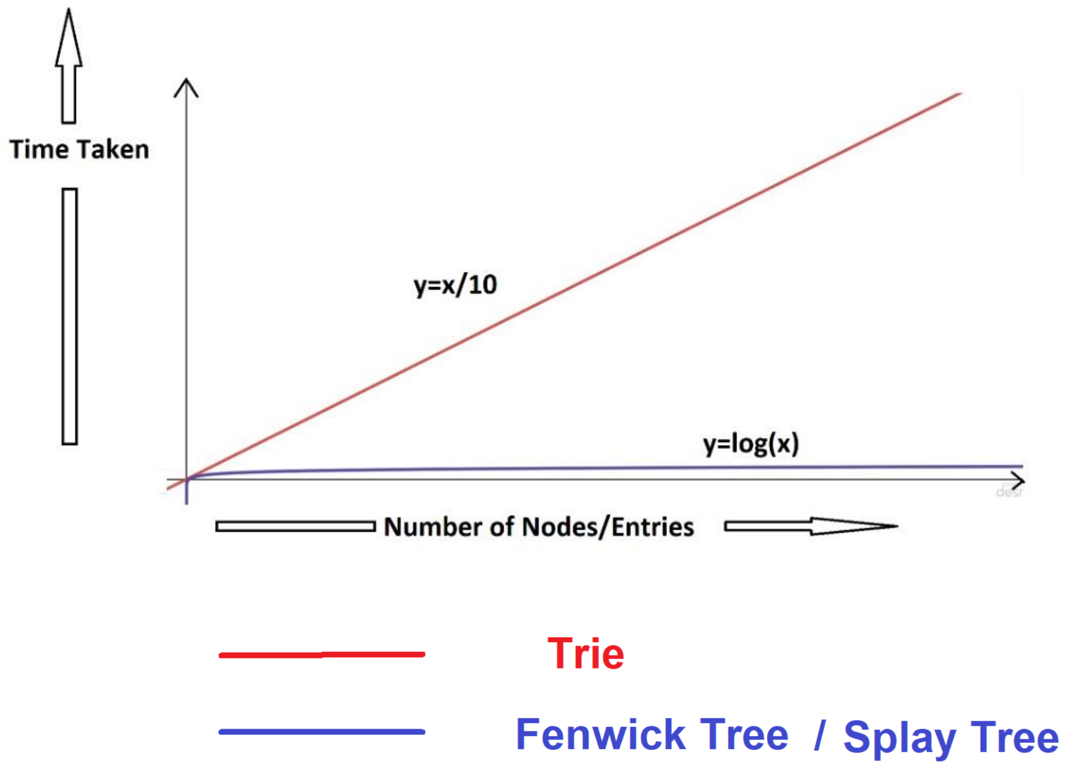
Fenwick Tree : For Updating, The value concerned is Updated by Difference of New and Old Value, i.e. just 1 Point Update, again leading to O(log(n)) Time Complexity where n are the number of Nodes.

Inference:

Time Complexity Wise:

Splay Tree Update ≈ Fenwick Tree Update < Trie Update

5) Special Operation



Performance:

As these are Special Operations, these are not meant to be compared with each other, but are just analyzed for their performance.

1) Splay Tree => SPLAYING:

Splaying involves rearranging the Splay tree to bring a particular element to the root of the tree.

Time Complexity: O(log(N)) where N are the Number of nodes in Splay Tree

2) Trie => Prefix Search:

This involves Searching the Trie for strings/words with common prefix. We can count/display/suggest these words using Trie!

Time Complexity: O(w) where w is the length of prefix

3) Fenwick Tree => Efficient Cumulative Sums for Operations with Inverse:

Cumulative Sums from beginning of array to an index can be done using this Data Structure by its implementation (binary indexing).

Time Complexity: O(log(N)) where N are the Number of Elements in Fenwick Tree

Inference:

These Data Structures are highly Optimized with respect to the Operations they are specialized for, compared to most of the other Data Structures available for the same.

Conclusion:

All throughout the report we were dealing majorly with the primarily focused three data structures and went on a deep analysis of the working of all three data structures in all five different operations. Then we also made comparison on performance of all of them. After doing this analysis we find that the each of the data structure have different application according to the advantage they have over one particular operation and thus we cannot say the supremacy of one data structure over the other (though they have structural similarity), but they have certain advantage over the other and all those are confined to a few particular operations. On the whole all the three data structures have a wide variety of applications in different areas and are equally important.