Red Black Trees vs Skip-List

Jibran Sheikh, Daniyal Farooqui, Ghulam Mustafa, Ikhlas Ahmed
Instructor: Mohammad Mobeen Movania

April 28, 2024

Red Black Tree Definition

A red-black tree is a binary search tree which has the following red-black properties:

1. Every node is either red or black. By constraining the node colors on any simple path from root to a leaf, RBT's ensure that no such path is twice as long as any other, so that the tree is approximately balanced.
2. Root of the tree must be Black.
3. Every leaf (NULL) is black.
4. If a node is red, then both its children are black. This point implies that on any path from the root to a leaf, red nodes must not be adjacent. However, any number of black nodes may appear in a sequence.
5. Every simple path from a node to a descendant leaf contains the same number of black nodes.

A basic Red Black Tree (Without Sentinels)

A red-black tree is a binary search tree which has the following red-black properties:

1. Every Red Node's children are black. Every leaf node is black (This is on the next slide)
2. Every simple path from a node to a descendant leaf contains the same number of black nodes.

Figure 1 illustrates this property:

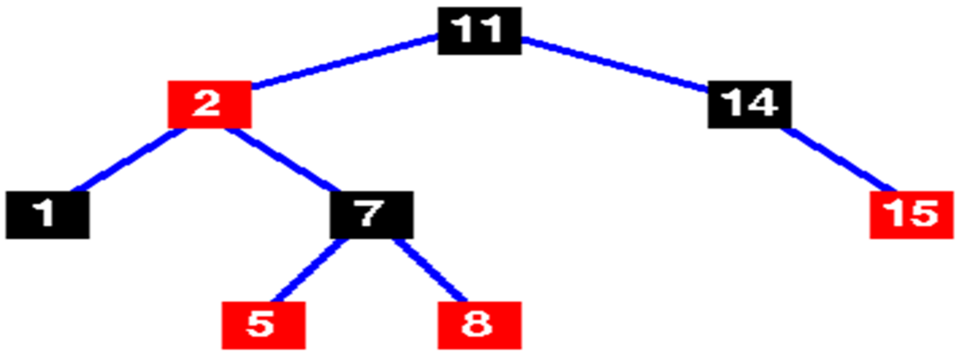


Figure 1

A basic Red Black Tree (With Sentinels)

Basic red-black tree with the **sentinel** nodes added. Implementations of the red-black tree algorithms will usually include the sentinel nodes as a convenient means of flagging that you have reached a leaf node.

Figure 2 illustrates this property:

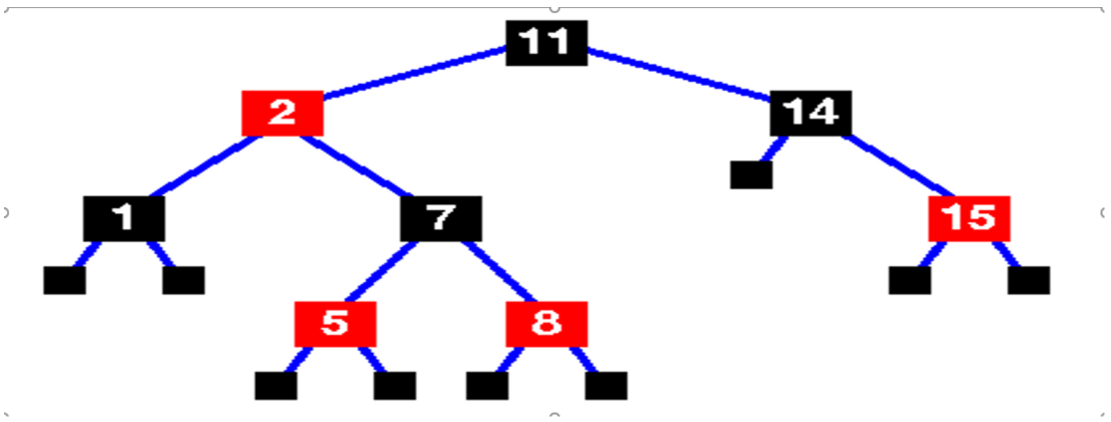


Figure 2

An alternative design would use a distinct sentinel node for each NIL in the tree, so that the parent of each NIL is well defined. That approach needlessly wastes space, however, Instead, just the one sentinel **T.nil** represents all the NILs-all leaves and the root's parent.

Figure 3 illustrates this property:

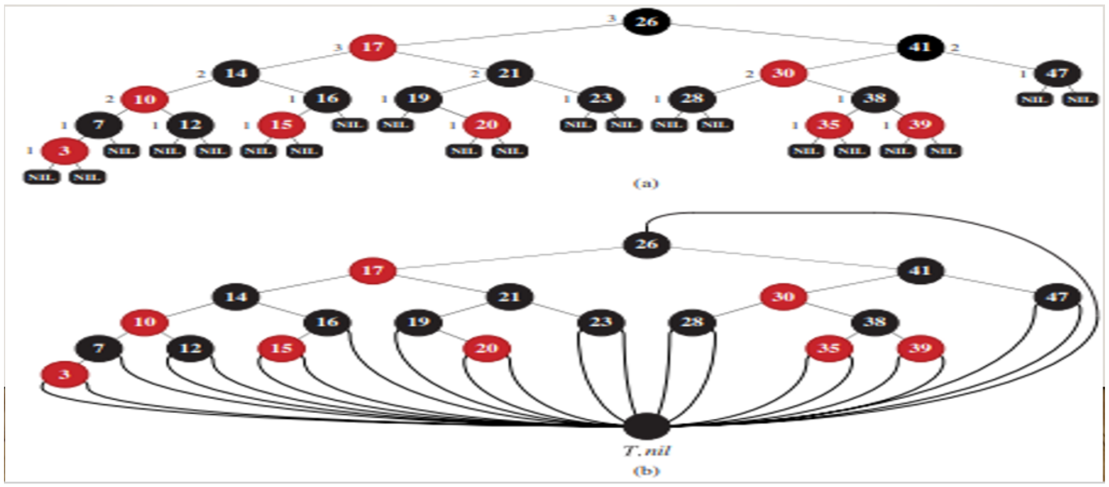


Figure 3

Black Height of the Node

We call the number of black nodes on any simple path from, but not including, a node 'x' down to a leaf, the **black-height** of the node, denoted by **bh(x)**. The black height of a Red Black Tree is the black-height of its root.

Lemma 13.1: A Red Black Tree with 'n' internal nodes has Height at most 2Log(n+1)

Proof:

- Start by showing that the sub-tree rooted at node 'x' contains at least $2^{bh(x)} - 1$ internal nodes.

- Prove by induction. If the height of 'x' is 0, then 'x' must be a leaf node (**T.nil**), and the sub-tree rooted at 'x' contains at least $2^{bh(x)}-1 = 2^0-1 = 0$ internal nodes. This is illustrated in figure 4.

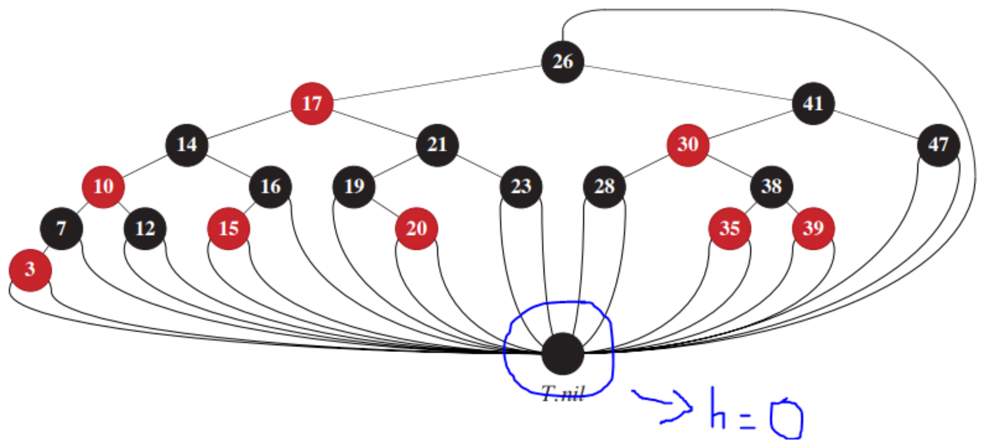


Figure 4

- Inductive Step:

  Consider a node ‘**x**’ which has a positive height and is an internal node. The node will have two children of which either both or one may be a leaf.

  If a child is black, then it contributes 1 to node x’s black height and none to its own. If a child is red, then it contributes none to it’s own and node x’s black height.

  Therefore, each child has a black-height of either **bh(x)–1** (if child is black) or **bh(x)** (if child is red).

  Since the height of the Child of x is less than node x’s height, we apply inductive hypothesis to conclude that each child has at least $2^{bh(x)-1}$–1 internal nodes.

  Thus, the sub-tree at node ‘x’ contains at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes.

- Final Step:

  Let ’**h**’ be height of the tree. At least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently. the Black Height of the root must be at least **h/2** and so,

  $$\mathbf{n} \geq 2^{h/2} - 1$$

  Rearranging the expression and taking log on both sides results in:

  $$\mathbf{n+1} \leq 2^{h/2}$$

  $$\log(n+1) \leq \log(2^{h/2})$$

$$\log(n+1) \leq h/2$$

$$\mathbf{h} \leq 2\log(n+1)$$

Search Runtime

As an immediate consequence of this lemma, the Search operation runs in O($\log(n)$) time on a Red Black Tree.

Since each run in O(h) time on a binary search tree of height 'h' and any red black tree on 'n' nodes is a binary search tree with height O($\log(n)$)

Rotations

The rotations in a Red Black Tree are similar to what we apply in Binary Search Tree. There are two types of rotations, Left Rotation and Right Rotation. Figure 1e below demonstrates how both Rotations work.

Rotations are performed in O(1) time since only pointers are rearranged to rotate nodes.

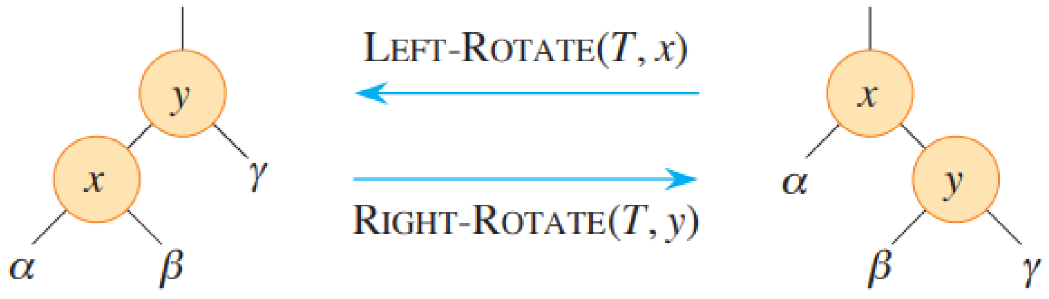
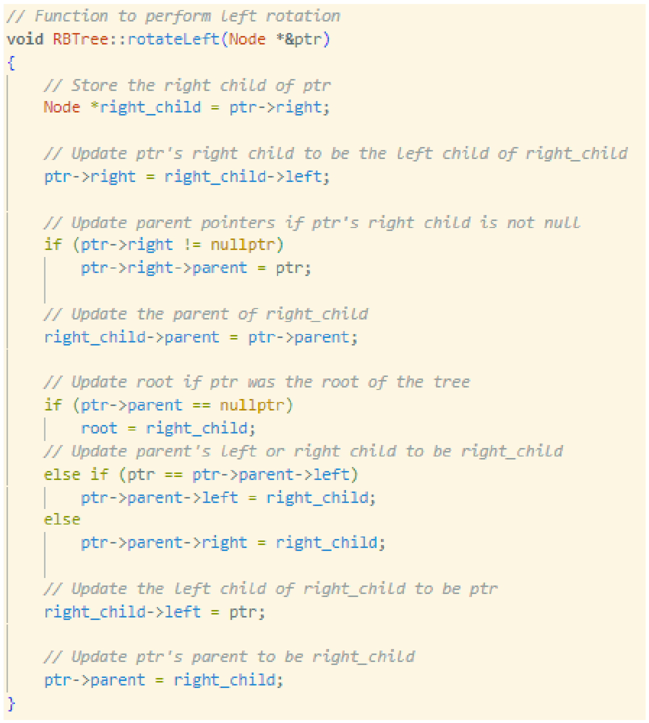


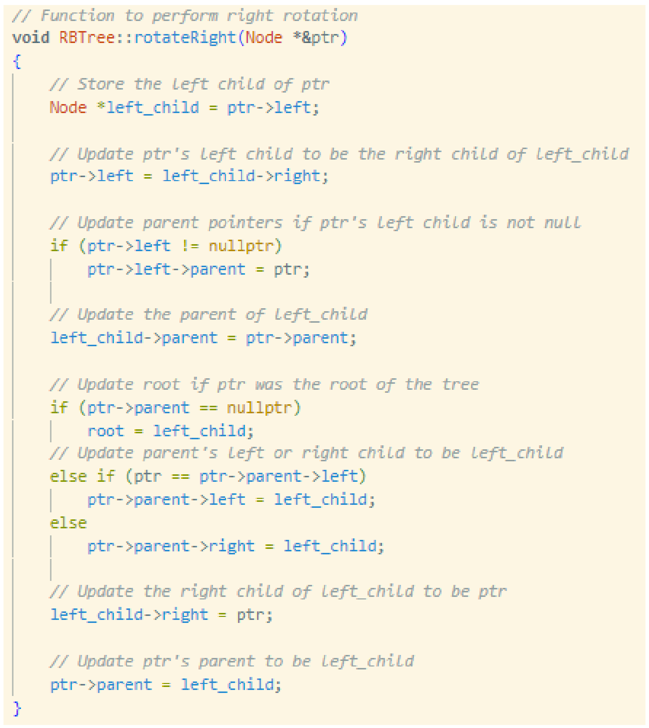
Figure 5

Figure 1f and 1g below show how left and right rotations work respectively.



```cpp
// Function to perform left rotation
void RBTree::rotateLeft(Node *&ptr)
{
    // Store the right child of ptr
    Node *right_child = ptr->right;

    // Update ptr's right child to be the left child of right_child
    ptr->right = right_child->left;

    // Update parent pointers if ptr's right child is not null
    if (ptr->right != nullptr)
        ptr->right->parent = ptr;

    // Update the parent of right_child
    right_child->parent = ptr->parent;

    // Update root if ptr was the root of the tree
    if (ptr->parent == nullptr)
        root = right_child;
    // Update parent's left or right child to be right_child
    else if (ptr == ptr->parent->left)
        ptr->parent->left = right_child;
    else
        ptr->parent->right = right_child;

    // Update the left child of right_child to be ptr
    right_child->left = ptr;

    // Update ptr's parent to be right_child
    ptr->parent = right_child;
}
```

Figure 6



```
// Function to perform right rotation
void RBTree::rotateRight(Node *&ptr)
{
    // Store the left child of ptr
    Node *left_child = ptr->left;

    // Update ptr's left child to be the right child of left_child
    ptr->left = left_child->right;

    // Update parent pointers if ptr's left child is not null
    if (ptr->left != nullptr)
        ptr->left->parent = ptr;

    // Update the parent of left_child
    left_child->parent = ptr->parent;

    // Update root if ptr was the root of the tree
    if (ptr->parent == nullptr)
        root = left_child;
    // Update parent's left or right child to be left_child
    else if (ptr == ptr->parent->left)
        ptr->parent->left = left_child;
    else
        ptr->parent->right = left_child;

    // Update the right child of left_child to be ptr
    left_child->right = ptr;

    // Update ptr's parent to be left_child
    ptr->parent = left_child;
}
```

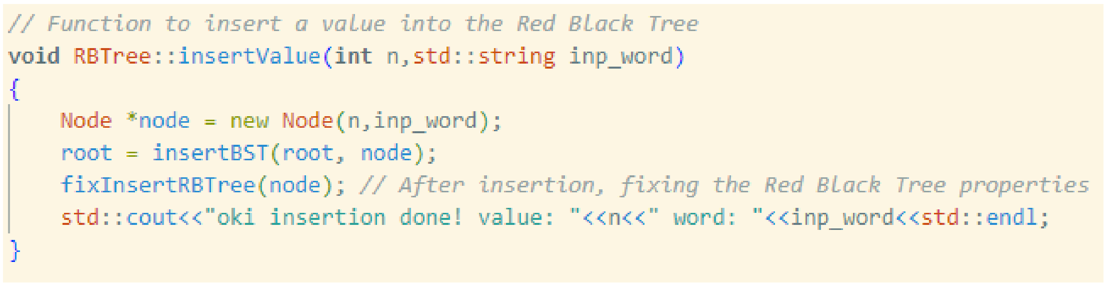
Figure 7

Insertion

In order to insert a node into a red-black tree with n internal nodes in $\mathbf{O}(\log(n))$ time and maintain the red-black properties, we call a RB-Tree-Insert-Fixup function which performs rotations and colors the nodes to maintain the Red Black Tree property.

We insert nodes in the tree as if it were a Binary Search Tree and color it red. Then, to guarantee that Red Black Tree properties are followed, we call the fixInsertRBTree function which recolors nodes and performs rotations.

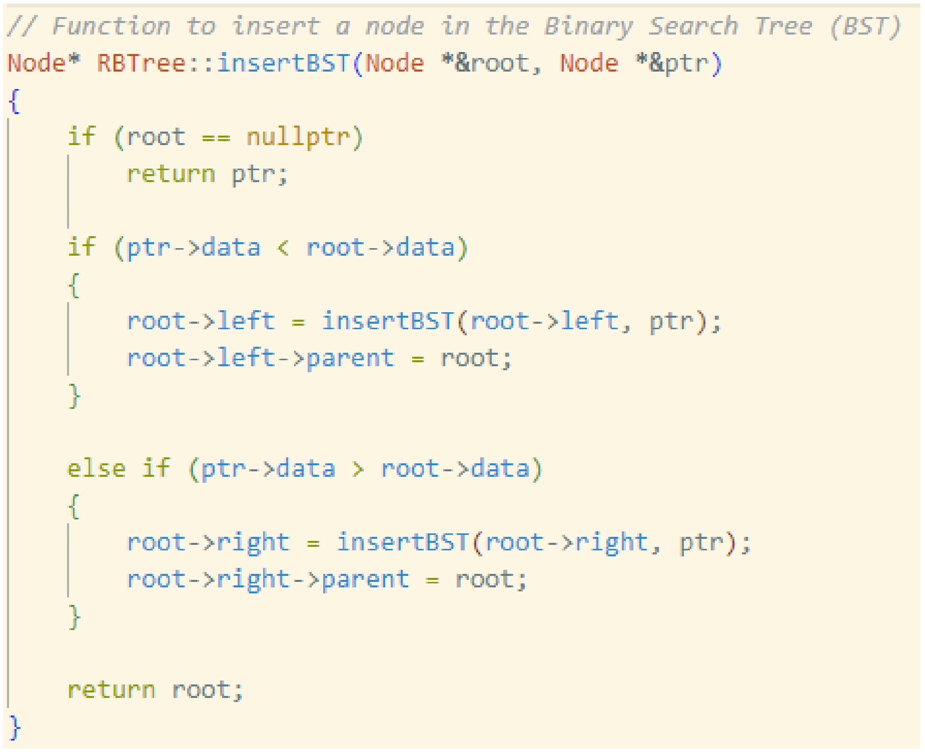
Figure 8 below shows the insert function which inserts a node in the Red Black Tree:



```
// Function to insert a value into the Red Black Tree
void RBTree::insertValue(int n,std::string inp_word)
{
    Node *node = new Node(n,inp_word);
    root = insertBST(root, node);
    fixInsertRBTree(node); // After insertion, fixing the Red Black Tree properties
    std::cout<<"oki insertion done! value: "<<n<<" word: "<<inp_word<<std::endl;
}
```

Figure 8

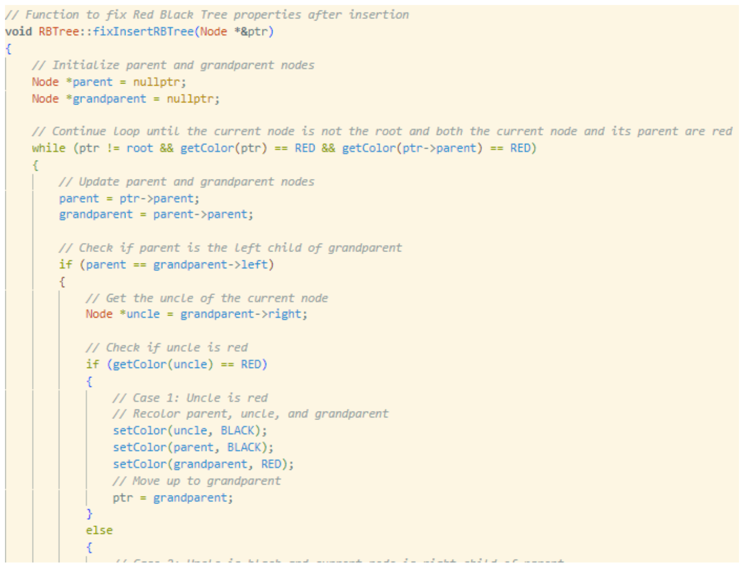
Figure 9 shows the insertBST function which inserts a node following the properties of a BST:



```
// Function to insert a node in the Binary Search Tree (BST)
Node* RBTree::insertBST(Node *&root, Node *&ptr)
{
    if (root == nullptr)
        return ptr;

    if (ptr->data < root->data)
    {
        root->left = insertBST(root->left, ptr);
        root->left->parent = root;
    }

    else if (ptr->data > root->data)
    {
        root->right = insertBST(root->right, ptr);
        root->right->parent = root;
    }

    return root;
}
```

Figure 9

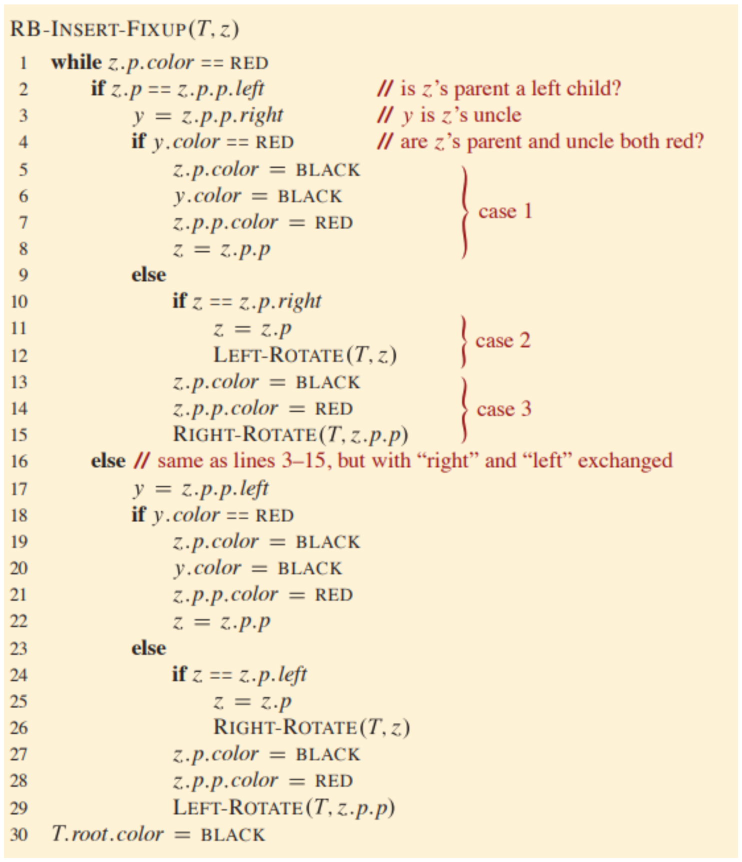
Figure 10 shows the fixInsertRBTree function which recolors the nodes and performs rotations (This is only part of the code):



```
// Function to fix Red Black Tree properties after insertion
void RBTree::fixInsertRBTree(Node *&ptr)
{
    // Initialize parent and grandparent nodes
    Node *parent = nullptr;
    Node *grandparent = nullptr;

    // Continue loop until the current node is not the root and both the current node and its parent are red
    while (ptr != root && getColor(ptr) == RED && getColor(ptr->parent) == RED)
    {
        // Update parent and grandparent nodes
        parent = ptr->parent;
        grandparent = parent->parent;

        // Check if parent is the left child of grandparent
        if (parent == grandparent->left)
        {
            // Get the uncle of the current node
            Node *uncle = grandparent->right;

            // Check if uncle is red
            if (getColor(uncle) == RED)
            {
                // Case 1: Uncle is red
                // Recolor parent, uncle, and grandparent
                setColor(uncle, BLACK);
                setColor(parent, BLACK);
                setColor(grandparent, RED);
                // Move up to grandparent
                ptr = grandparent;
            }
            else
            {
```

Figure 10

Figure 11 is an image of the function from the book we have referenced as it provides a summarized logic of this function.



```
RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left              // is z's parent a left child?
3          y = z.p.p.right               // y is z's uncle
4          if y.color == RED             // are z's parent and uncle both red?
5              z.p.color = BLACK         ⎫
6              y.color = BLACK           ⎬ case 1
7              z.p.p.color = RED         ⎪
8              z = z.p.p                 ⎭
9          else
10             if z == z.p.right
11                 z = z.p               ⎫ case 2
12                 LEFT-ROTATE(T, z)     ⎭
13             z.p.color = BLACK         ⎫
14             z.p.p.color = RED         ⎬ case 3
15             RIGHT-ROTATE(T, z.p.p)    ⎭
16     else // same as lines 3–15, but with "right" and "left" exchanged
17         y = z.p.p.left
18         if y.color == RED
19             z.p.color = BLACK
20             y.color = BLACK
21             z.p.p.color = RED
22             z = z.p.p
23         else
24             if z == z.p.left
25                 z = z.p
26                 RIGHT-ROTATE(T, z)
27             z.p.color = BLACK
28             z.p.p.color = RED
29             LEFT-ROTATE(T, z.p.p)
30 T.root.color = BLACK
```

Figure 11

Working of fixInsertRBTree

The properties violated after the insertion of a node in a Red Black Tree are Property 2 and Property 4.

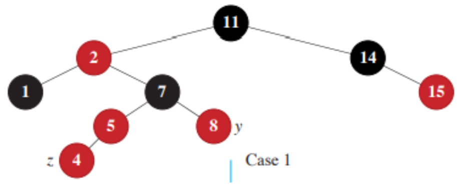
Property 2: Requires the root to be black.

Property 4: Red Node cannot have a Red child.

Property 2 is violated if the inserted node is the root of the tree and Property 4 is violated if inserted node's parent is red.

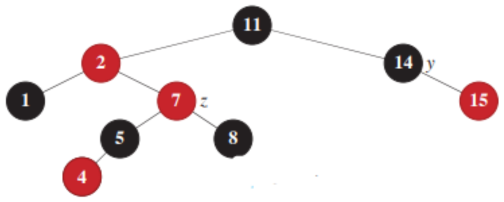
To maintain the properties of a Red Black Tree, there are two cases which occur.

- Case 1: Z's parent is a left child of Z's grandparent.

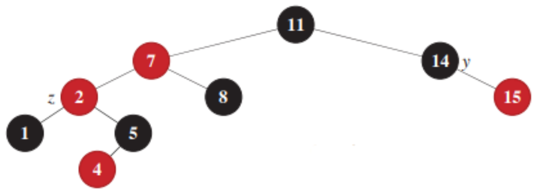


When such a case occurs, there are further two sub-cases which occur,

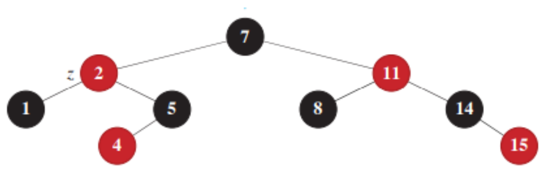
  – When Z's parent and Uncle are both Red. We perform the following operations;
    * Z's Parent color change from RED to BLACK
    * Z's Uncle color change from RED to BLACK
    * Z's Grand-Parent color change from BLACK to RED
    * Make Z it's Grand-Parent.



  – When Z's parent is red but Uncle is Black. We perform the following operations;
    * If Z is right child of Parent then,
      · Make Z its Parent and Left-Rotate over Z.



    * If Z is left child of Parent then,
      · Change Z's Parent color from RED to BLACK.
      · Change Z's Grand-Parent color from Black to RED.
      · Right-Rotate over Z's Grand-Parent



- Case 2: Z's Parent is a right child of Z's Grand-Parent.

  Similar cases occur in this as mentioned above. The only difference is that the Left and Right Rotations are now interchanged.

This way, the Properties of a Red Black Tree will be maintained after a node has been inserted.

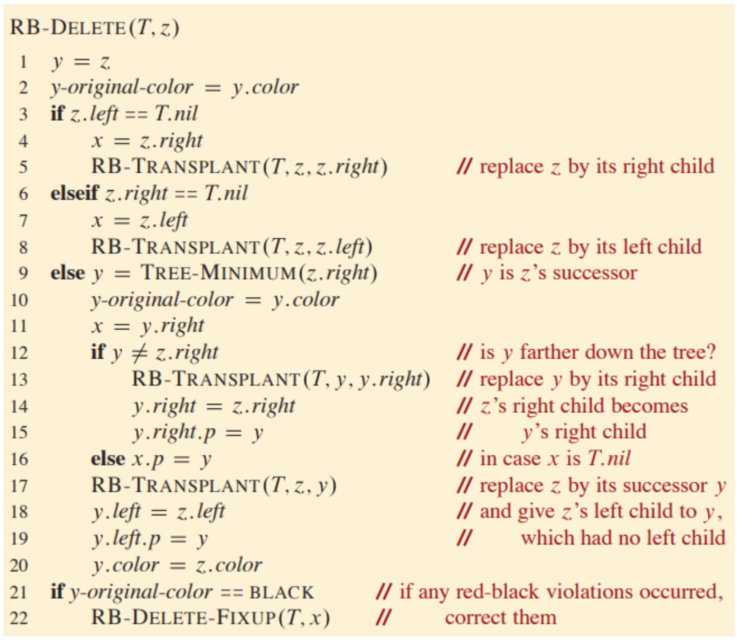
Analysis of fixInsertRBTree

Since the height of a Red Black Tree on 'n' nodes is O(log(n)), Case 1 of the function takes O(log(n)) time.

The while loop only repeats if sub-case 1 occurs after which the pointer Z moves two levels up the tree. Hence, the While loop can be executed O(log(n)) times and so the function takes O(log(n)) time to execute.

Deletion

The RB-DELETE-FIXUP function is used in Red-Black Tree deletion to maintain the Red-Black properties after a node is deleted. It works by fixing any violations that may occur after deletion, such as violating the red-black color rules or unbalancing the tree. Figure 11 below shows the Delete function logic,



```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)        // replace z by its right child
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)         // replace z by its left child
else y = TREE-MINIMUM(z.right)          // y is z's successor
    y-original-color = y.color
    x = y.right
    if y ≠ z.right                      // is y farther down the tree?
        RB-TRANSPLANT(T, y, y.right)    // replace y by its right child
        y.right = z.right               // z's right child becomes
        y.right.p = y                   //     y's right child
    else x.p = y                        // in case x is T.nil
    RB-TRANSPLANT(T, z, y)              // replace z by its successor y
    y.left = z.left                     // and give z's left child to y,
    y.left.p = y                        //     which had no left child
    y.color = z.color
if y-original-color == BLACK        // if any red-black violations occurred,
    RB-DELETE-FIXUP(T, x)           //     correct them
```

Figure 11

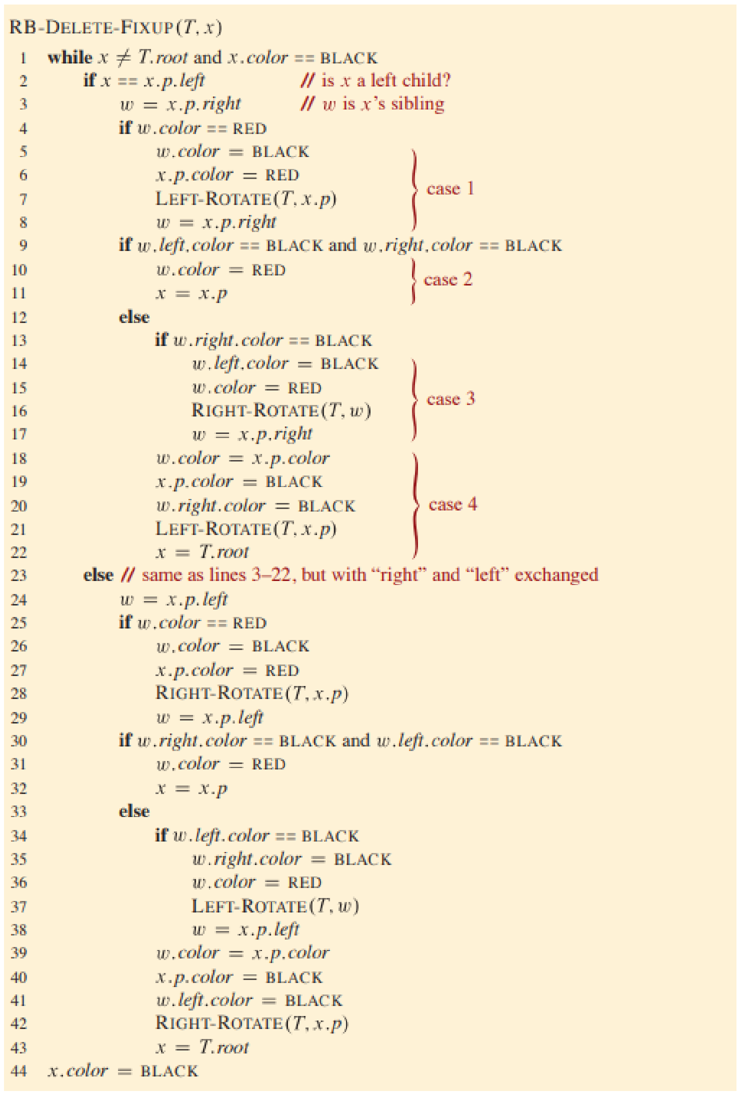
RB-DELETE-FIXUP

The RB-DELETE-FIXUP function has a few cases and we will be explaining them below.

- Case 1: x's sibling w is red.
  If w is red, it must have black children. Colors of w and x:p are switched, and a left rotation is performed on x:p. This case ensures that the new sibling of x is black. Case 1 then converts into one of cases 2, 3, or 4.
- Case 2: x's sibling w is black, and both of w's children are black.
  Both of w's children are black, and w itself is black. Removes one black from both x and w, leaving x with only one black and w red. x:p takes on an extra black to compensate, and x moves up one level. Terminates when x is singly black.
- Case 3: x's sibling w is black, w's left child is red, and w's right child is black.
  w is black, its left child is red, and its right child is black. Colors of w and its left child are switched, and a right rotation is performed on w. Ensures that the new sibling of x is a black node with a red right child. Case 3 falls through into case 4.
- Case 4: x's sibling w is black, and w's right child is red.
  w is black, and its right child is red. Color changes and a left rotation

on x:p allow the extra black on x to vanish. x becomes the root, and the while loop terminates.

The working of the function is shown in the below figure 12.



```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left                    // is x a left child?
 3          w = x.p.right                   // w is x's sibling
 4          if w.color == RED
 5              w.color = BLACK             ⎫
 6              x.p.color = RED             ⎬ case 1
 7              LEFT-ROTATE(T, x.p)         ⎪
 8              w = x.p.right               ⎭
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED               ⎫ case 2
11              x = x.p                     ⎭
12          else
13              if w.right.color == BLACK
14                  w.left.color = BLACK    ⎫
15                  w.color = RED           ⎬ case 3
16                  RIGHT-ROTATE(T, w)      ⎪
17                  w = x.p.right           ⎭
18              w.color = x.p.color         ⎫
19              x.p.color = BLACK           ⎪
20              w.right.color = BLACK       ⎬ case 4
21              LEFT-ROTATE(T, x.p)         ⎪
22              x = T.root                  ⎭
23      else // same as lines 3–22, but with "right" and "left" exchanged
24          w = x.p.left
25          if w.color == RED
26              w.color = BLACK
27              x.p.color = RED
28              RIGHT-ROTATE(T, x.p)
29              w = x.p.left
30          if w.right.color == BLACK and w.left.color == BLACK
31              w.color = RED
32              x = x.p
33          else
34              if w.left.color == BLACK
35                  w.right.color = BLACK
36                  w.color = RED
37                  LEFT-ROTATE(T, w)
38                  w = x.p.left
39              w.color = x.p.color
40              x.p.color = BLACK
41              w.left.color = BLACK
42              RIGHT-ROTATE(T, x.p)
43              x = T.root
44  x.color = BLACK
```

Figure 12

Deletion Analysis

Since the height of the Red Black Tree of 'n' nodes is O(log(n)), the total cost of the procedure without the call to RB-Delete-Fixup takes O(log(n)) time. Case 2 is the only case in the function where the while loop can be repeated, and then the pointer 'x' moves up the tree at most O(log(n)) times, performing no rotations. Thus, the function takes O(log(n)).

Skiplist Definition

Skiplist is a type of linked list with the following properties:

1. Skip list consists of multiple levels. The lowest level contains all the elements in sorted order. Each higher level is a subset of the level below it, containing fewer elements but possibly skipping over some.

2. Each node in the skip list contains a key-value pair. Additionally, each node has multiple pointers, one for each level it appears in.

3. The top level contains only the head and tail nodes. The head node serves as the starting point for searches and contains the smallest key in the skip list. The tail node contains the largest key and serves as the end of the skip list.

4. Skip lists use randomization to determine the number of levels for each node. Nodes at higher levels are less frequent and are reached by "skipping" over multiple nodes at lower levels.

5. Searching and insertion in a skip list are similar to searching and insertion in a linked list. Starting from the top level, we move forward until we find the desired element or reach a node with a larger key.

A basic Skip-List

A skip list is a probabilistic data structure that provides logarithmic time complexity for search, insertion, and deletion operations while maintaining approximate balance through randomization. It achieves this balance by using multiple levels of nodes with carefully chosen connections between them.

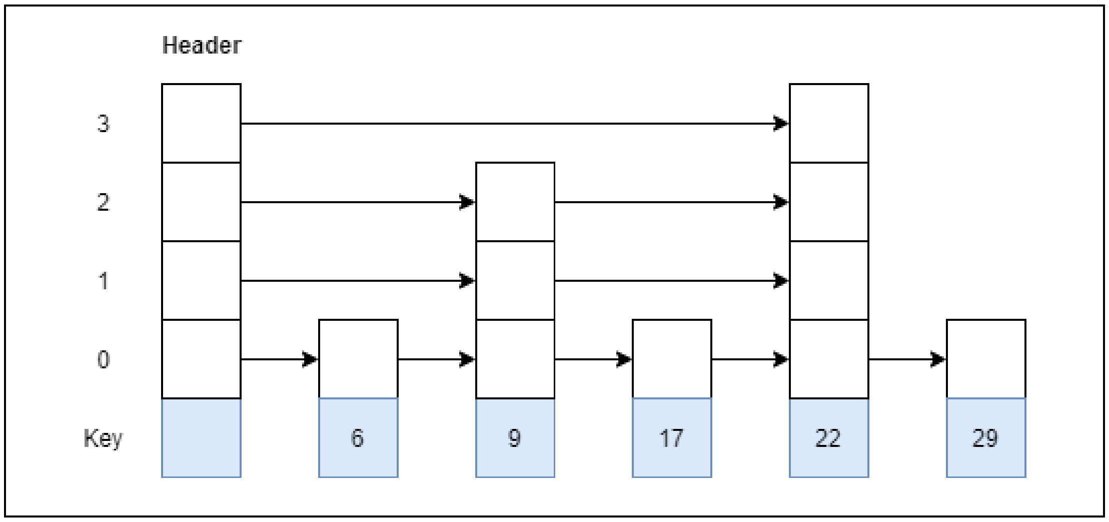


Figure 13

The figure 13, above demonstrates a basic skip list structures with levels and nodes.

0.1 Insertion

To insert a new node, we would first find the location where this new key should be inserted in the list at the very bottom. This can be simply done be re-using the search logic, we start traversing to the right from the list on top and we go one step down if the next item is bigger than the key that we want to insert, else go right. Once we reach at a position in bottom list where we can't go more to the right, we insert the new value on the right side of that position. The figures below demonstrate the process.

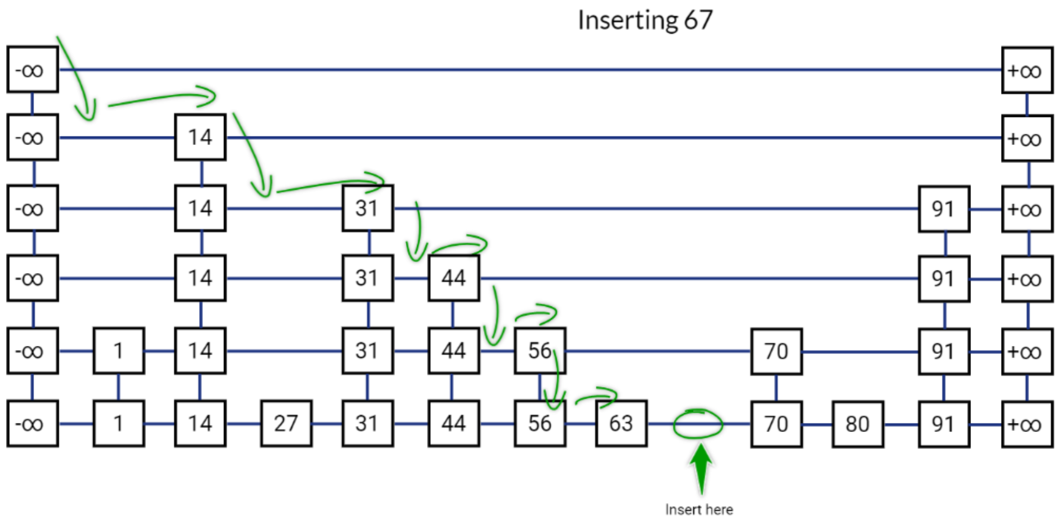


Figure 14

Skip List after inserting 67:

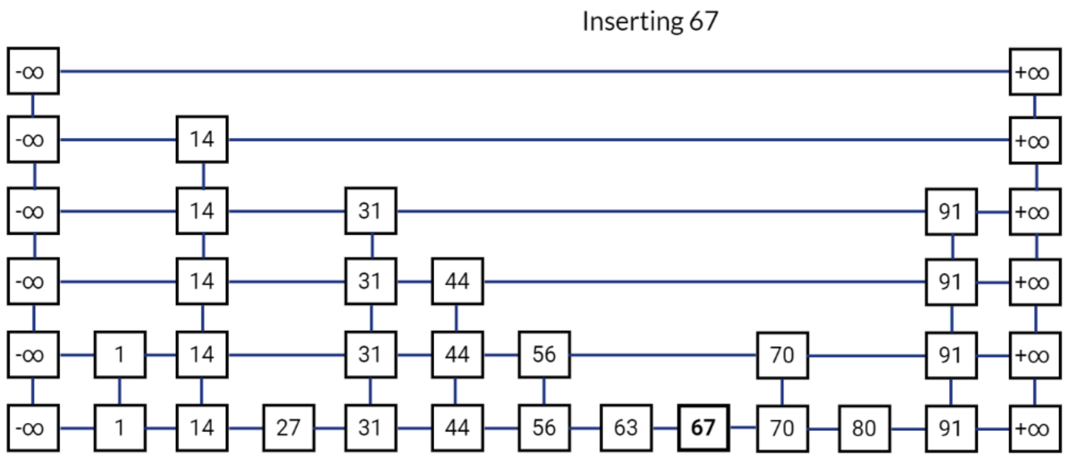


Figure 15

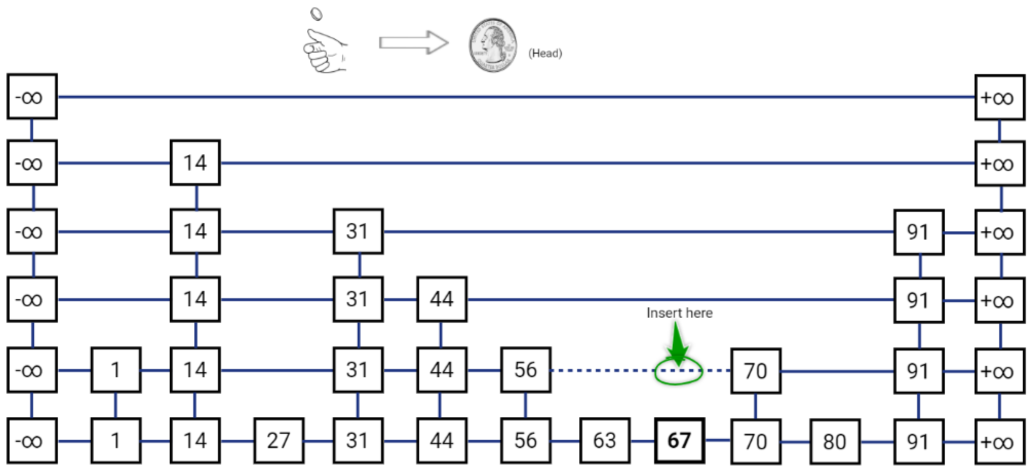


Figure 16

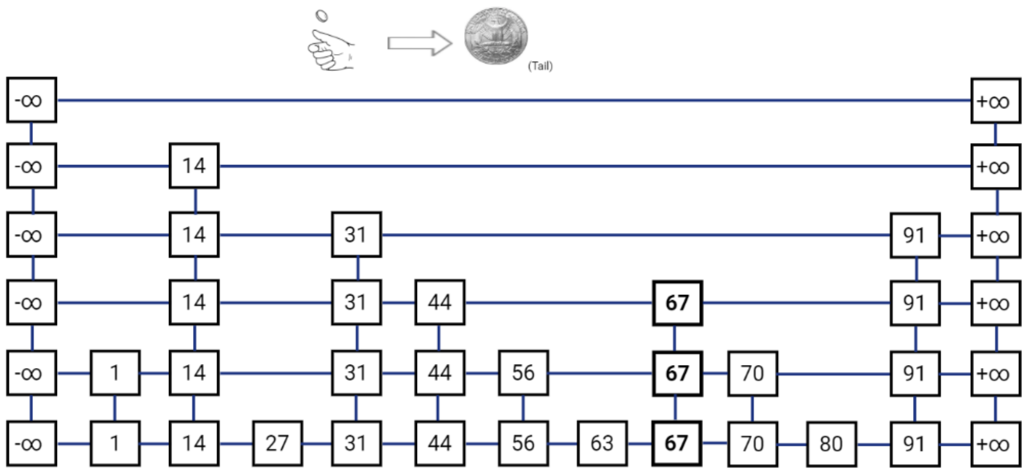


Figure 17

0.2 Searching

Searching in a Skip List in straightforward. We have already seen an example above with an example. We can extend the same approach to a Skip List with h lists. We will simply start with the leftmost node in the list on top. Go down if the next value in the same list is greater than the one we are looking for, go right otherwise.
The main idea is again to skip comparisons with as many keys as possible,

while compromising a little on the extra storage required in the additional lists containing subset of all keys. On average, “expected” Search complexity for Skip List is O(log(n))

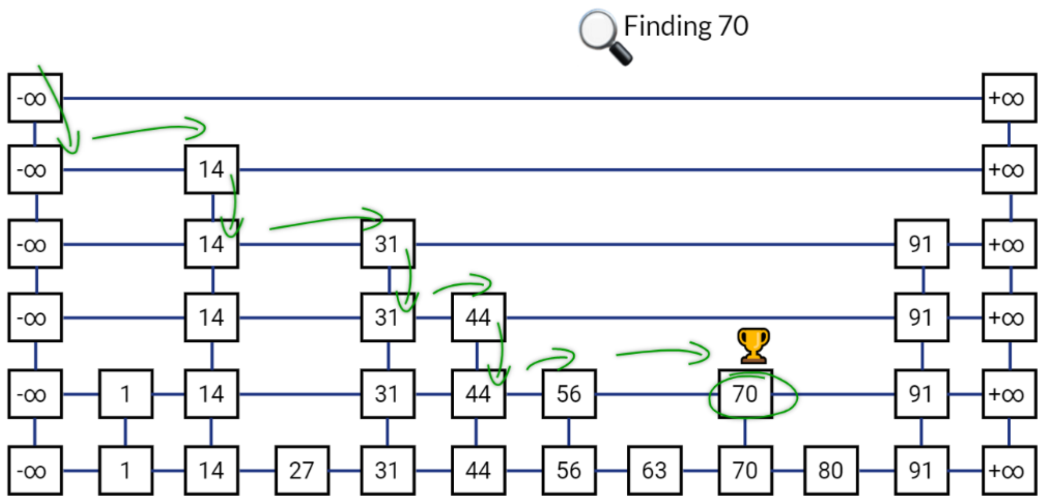


Figure 18

0.3 Deletion

Deletion operation in Skip List is pretty straightforward. We first perform search operation to find the location of the node. If we find the node, we simply delete it at all levels. Before deleting a node, we simply ensure that no other node is pointing to it. The figures below demonstrate the process.

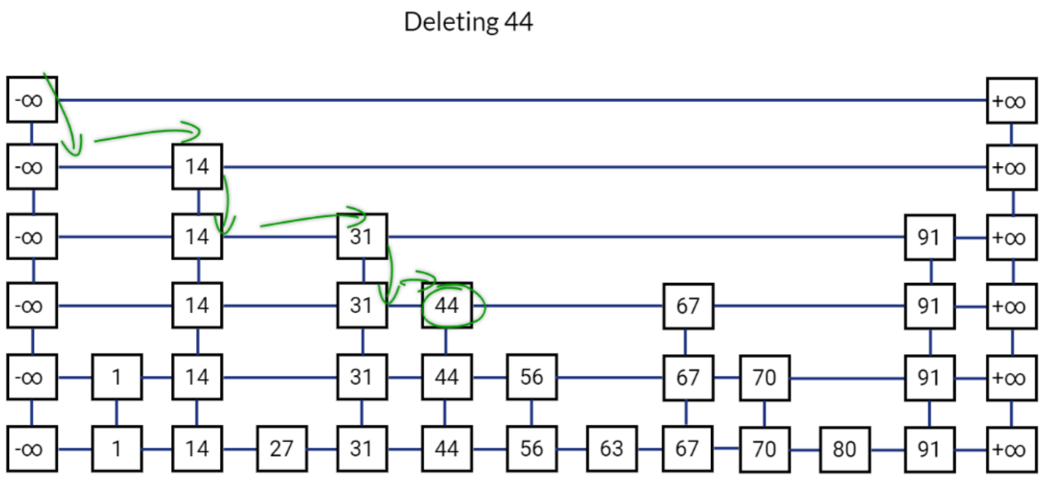


Figure 19

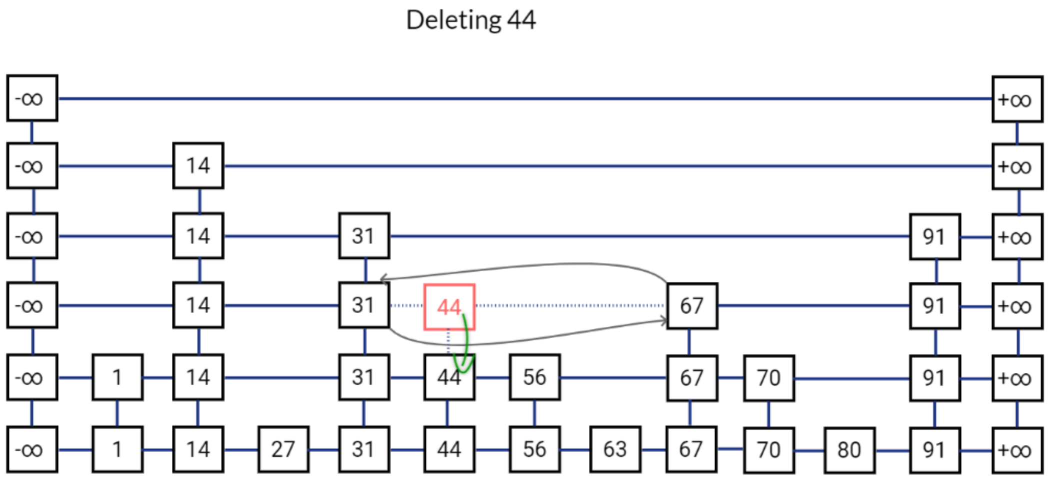


Figure 20

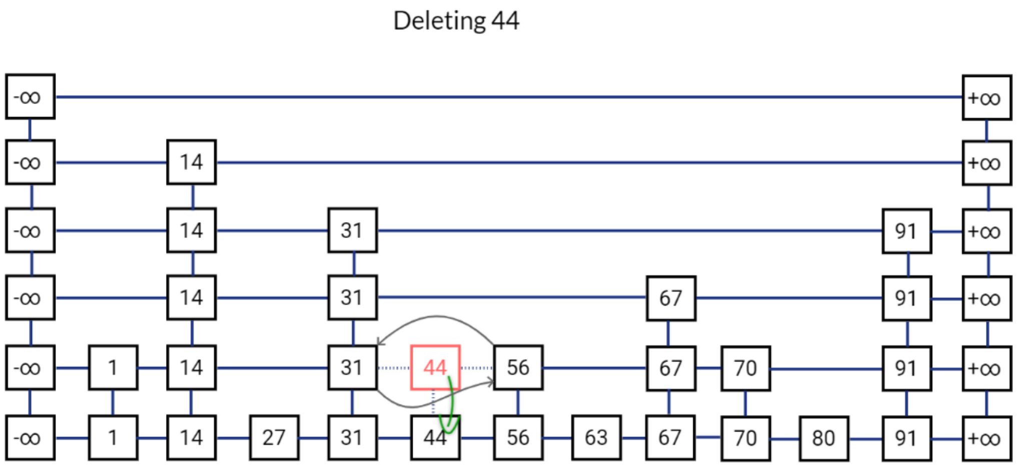


Figure 21

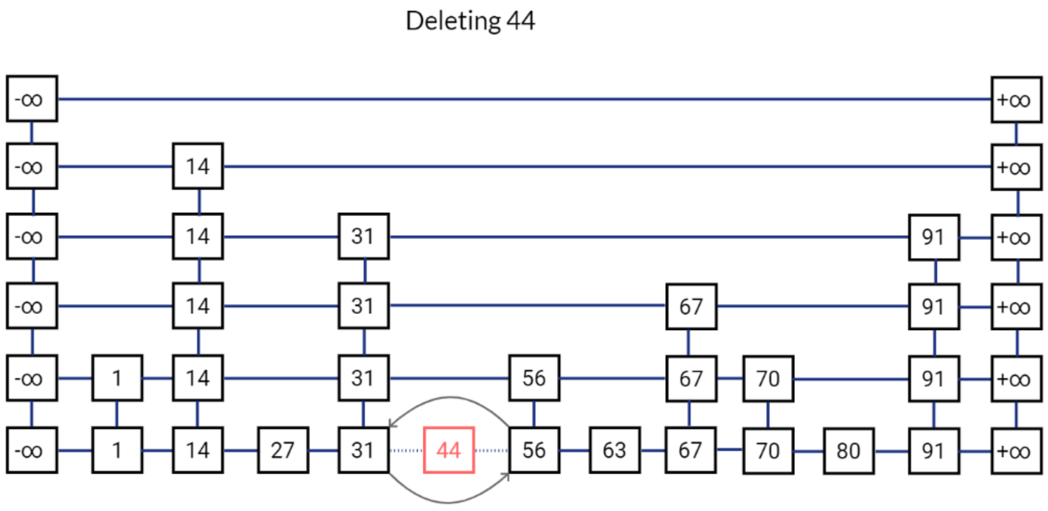


Figure 22

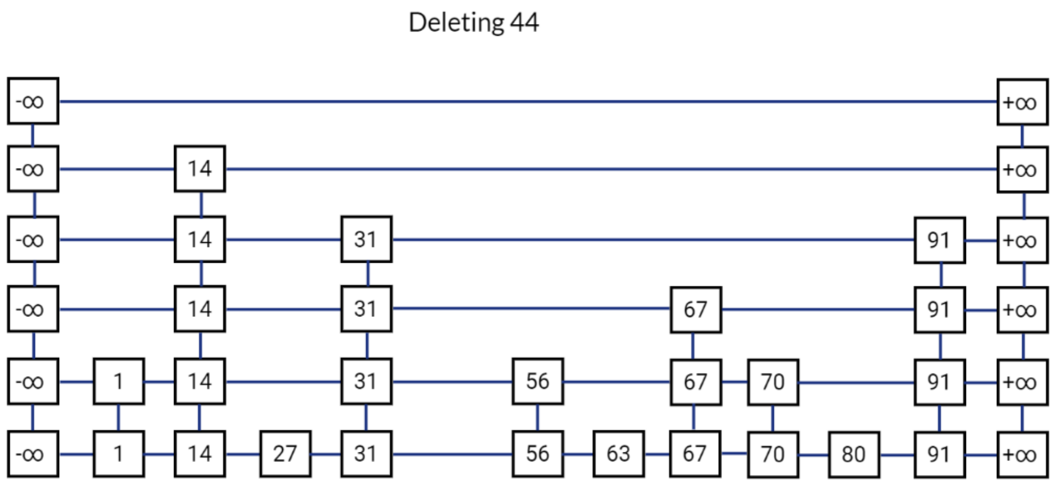


Figure 23

Skip-list vs Red Black Trees Analysis

Now we will be moving to the part where we compare both the data structures, Skiplist and Red-Black Trees. We will be testing both data structures on same dataset. The three basic functions of both Data Structures, Search, Insert, and Delete will be tested and compared with each other.

The Datasets have been divided into two. One containing words which may result in duplicates whereas the second dataset results in unique nodes. There are three datasets, all of which contain different number of words. This variation in dataset size allows us to thoroughly evaluate the efficiency of both data structures.

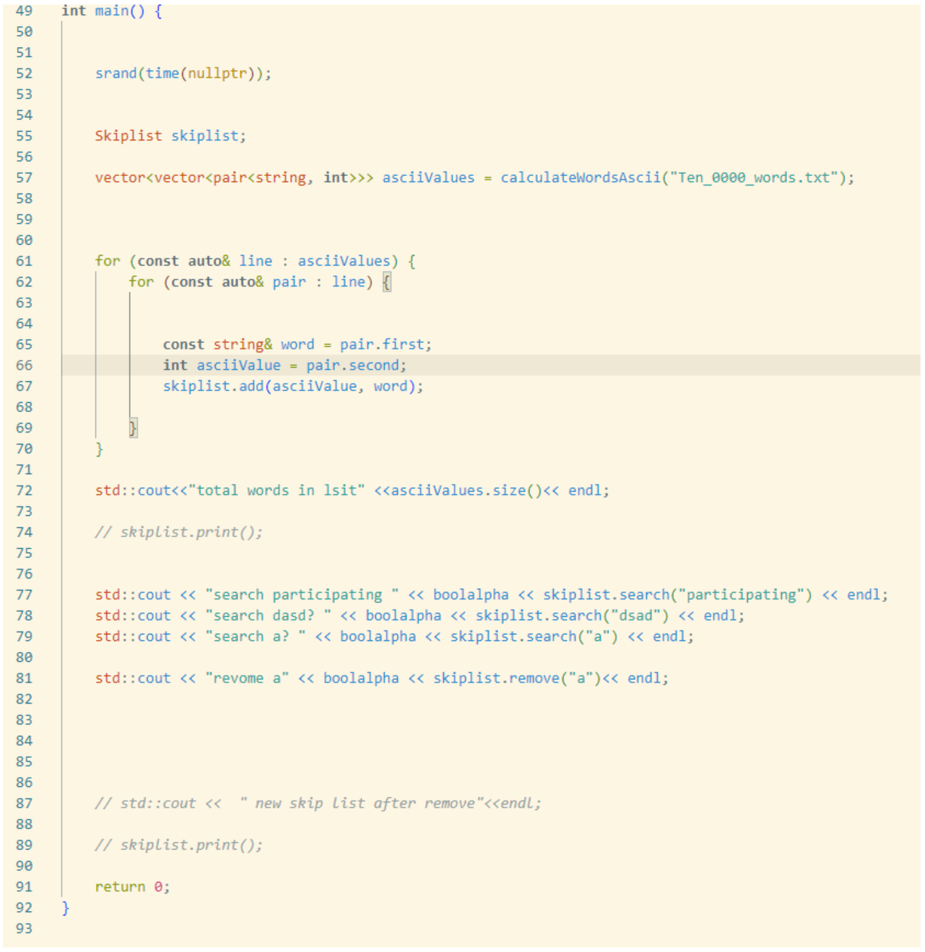
We have divided the analysis into 'Analysis for Unqiue Data-Set' and 'Analysis for Duplicate Data-Set' since the analysis for both would slightly differ.

It is **important to note** that the time calculated for each function may differ from device to device. We noticed that the time for the same function called again with no changes also results in a different value hence to achieve better accuracy, we have tested all the functions 10 times without any changes and have taken their average to get better accuracy.

Analysis of Red-Black Trees vs Skiplist (Unique Entries Data-Set)

Skiplist Formation

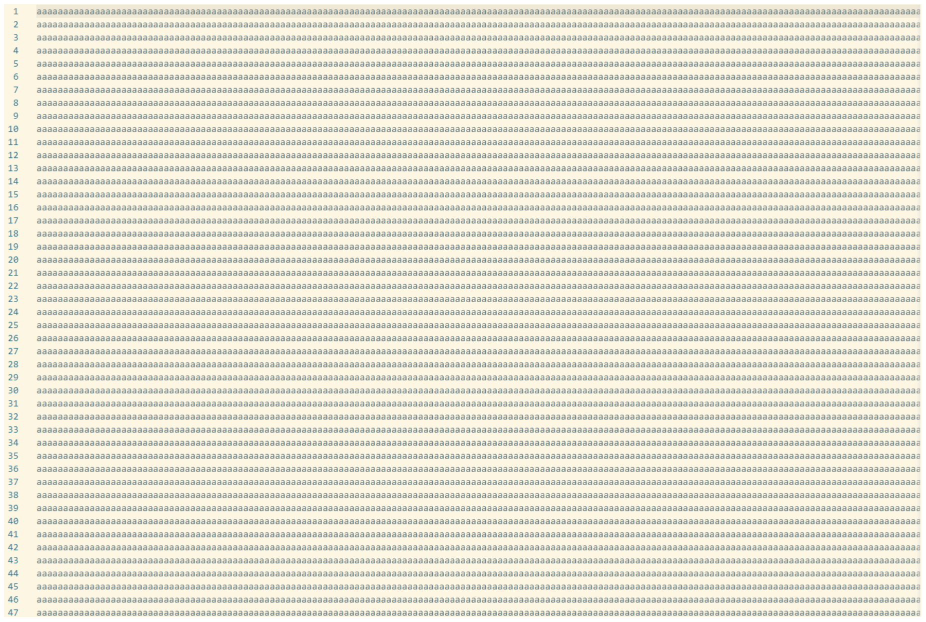
The Skip list formation and its functions have been explained in detail earlier. Now, we will read the words in the two datasets and form a Skip List.
The following figure 24 shows the code reading the unique database and forming the Skip List



```
int main() {


    srand(time(nullptr));


    Skiplist skiplist;

    vector<vector<pair<string, int>>> asciiValues = calculateWordsAscii("Ten_0000_words.txt");



    for (const auto& line : asciiValues) {
        for (const auto& pair : line) {


            const string& word = pair.first;
            int asciiValue = pair.second;
            skiplist.add(asciiValue, word);

        }
    }

    std::cout<<"total words in lsit" <<asciiValues.size()<< endl;

    // skiplist.print();


    std::cout << "search participating " << boolalpha << skiplist.search("participating") << endl;
    std::cout << "search dasd? " << boolalpha << skiplist.search("dsad") << endl;
    std::cout << "search a? " << boolalpha << skiplist.search("a") << endl;

    std::cout << "revome a" << boolalpha << skiplist.remove("a")<< endl;





    // std::cout <<  " new skip list after remove"<<endl;

    // skiplist.print();

    return 0;
}

```

Figure 24

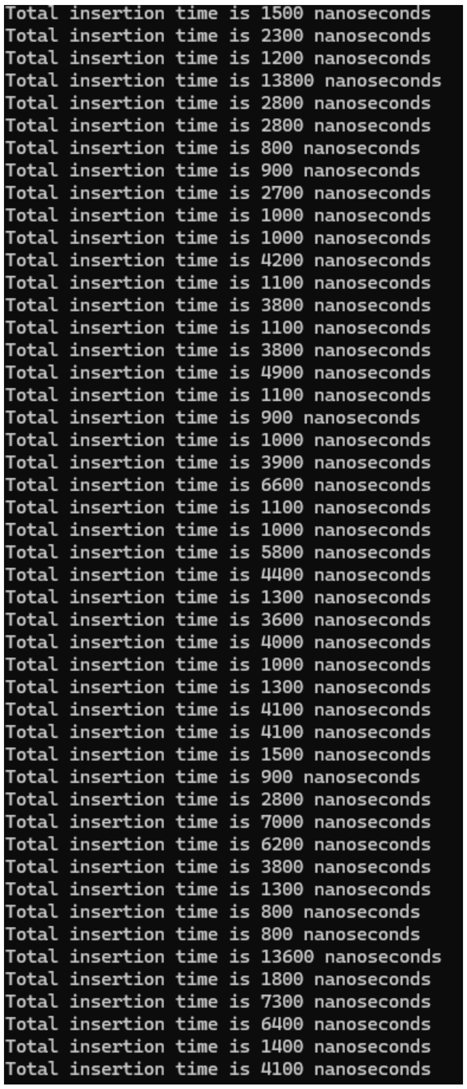
We are creating nodes for each entry (word) from the data-set by calculating the ASCII value of the word. Since the length and each word is kept different, it will result in a Unique ASCII value hence unique nodes will be formed.
The following figure 25 shows what the Data-Set comprises of,



```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 25

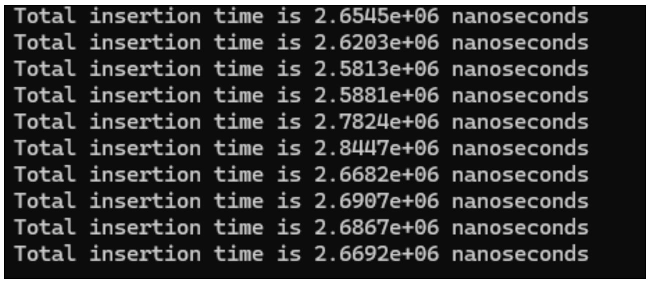
This is the sample dataset which contains only 100 entries. The first 47 entries are shown in the screenshot above. We will test on 100 words, 2000 words, 5000 words and 10000 words.
The below figure 26 shows us how the skip list is formed.



```
Total insertion time is 1500 nanoseconds
Total insertion time is 2300 nanoseconds
Total insertion time is 1200 nanoseconds
Total insertion time is 13800 nanoseconds
Total insertion time is 2800 nanoseconds
Total insertion time is 2800 nanoseconds
Total insertion time is 800 nanoseconds
Total insertion time is 900 nanoseconds
Total insertion time is 2700 nanoseconds
Total insertion time is 1000 nanoseconds
Total insertion time is 1000 nanoseconds
Total insertion time is 4200 nanoseconds
Total insertion time is 1100 nanoseconds
Total insertion time is 3800 nanoseconds
Total insertion time is 1100 nanoseconds
Total insertion time is 3800 nanoseconds
Total insertion time is 4900 nanoseconds
Total insertion time is 1100 nanoseconds
Total insertion time is 900 nanoseconds
Total insertion time is 1000 nanoseconds
Total insertion time is 3900 nanoseconds
Total insertion time is 6600 nanoseconds
Total insertion time is 1100 nanoseconds
Total insertion time is 1000 nanoseconds
Total insertion time is 5800 nanoseconds
Total insertion time is 4400 nanoseconds
Total insertion time is 1300 nanoseconds
Total insertion time is 3600 nanoseconds
Total insertion time is 4000 nanoseconds
Total insertion time is 1000 nanoseconds
Total insertion time is 1300 nanoseconds
Total insertion time is 4100 nanoseconds
Total insertion time is 4100 nanoseconds
Total insertion time is 1500 nanoseconds
Total insertion time is 900 nanoseconds
Total insertion time is 2800 nanoseconds
Total insertion time is 7000 nanoseconds
Total insertion time is 6200 nanoseconds
Total insertion time is 3800 nanoseconds
Total insertion time is 1300 nanoseconds
Total insertion time is 800 nanoseconds
Total insertion time is 800 nanoseconds
Total insertion time is 13600 nanoseconds
Total insertion time is 1800 nanoseconds
Total insertion time is 7300 nanoseconds
Total insertion time is 6400 nanoseconds
Total insertion time is 1400 nanoseconds
Total insertion time is 4100 nanoseconds
```

Figure 26

As we saw earlier, the complexity of the Insert function is O(log(n)). According to the above figure showing the time taken for insertion and number of nodes in the tree, we can analyze if the time taken is in fact correct. For the formation of the Tree, we have taken the sum of time taken by the insert function each time a node is inserted and displayed it in the terminal.
To get better accuracy, we have formed the skip list with this dataset 10 times and have taken the average of time taken for insertion.



```
Total insertion time is 2.6545e+06 nanoseconds
Total insertion time is 2.6203e+06 nanoseconds
Total insertion time is 2.5813e+06 nanoseconds
Total insertion time is 2.5881e+06 nanoseconds
Total insertion time is 2.7824e+06 nanoseconds
Total insertion time is 2.8447e+06 nanoseconds
Total insertion time is 2.6682e+06 nanoseconds
Total insertion time is 2.6907e+06 nanoseconds
Total insertion time is 2.6867e+06 nanoseconds
Total insertion time is 2.6692e+06 nanoseconds
```

Figure 27

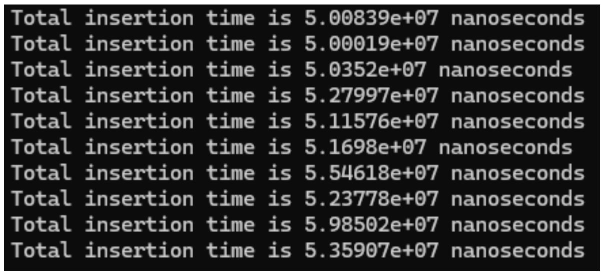
Time taken = $((2.6545 \times 10^6) + (2.6203 \times 10^6) + \ldots + (2.5813 \times 10^6))/10$
Time taken = $(26.786 \times 10^6)/10 = 2.67861 \times 10^6$ ns = **0.00267861 s**

According to the complexity of the function, the time should be less than or equal to this,
$O(\log(n)) = O(\log(100))$ = **6.64 s**

The time taken for forming a skiplist with 100 nodes is less than the time according to $O(\log(n))$ complexity of the insert function since **0.002678618s is less than 6.64s**.

Below is the analysis of the formation of Skiplist for Small, Medium, and Larger datasets.

- Small Dataset:
  The formation of the tree can be tested out in the code. Below is the analysis of the time complexity.

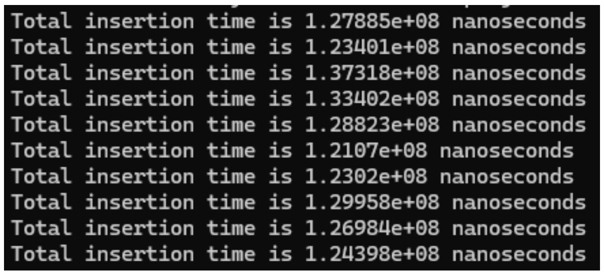


```
Total insertion time is 5.00839e+07 nanoseconds
Total insertion time is 5.00019e+07 nanoseconds
Total insertion time is 5.0352e+07 nanoseconds
Total insertion time is 5.27997e+07 nanoseconds
Total insertion time is 5.11576e+07 nanoseconds
Total insertion time is 5.1698e+07 nanoseconds
Total insertion time is 5.54618e+07 nanoseconds
Total insertion time is 5.23778e+07 nanoseconds
Total insertion time is 5.98502e+07 nanoseconds
Total insertion time is 5.35907e+07 nanoseconds
```

Figure 28

  Time Taken = $(5.0083 \times 10^7 + \ldots + 5.3509 \times 10^7)/10 = 5.2737 \times 10^7$ ns = **0.05273796 s**
  $O(\log(n)) = O(\log(2500))$ = **11.28 s**

- Medium Dataset:
  The formation of the tree can be tested out in the code. Below is the analysis of the time complexity.

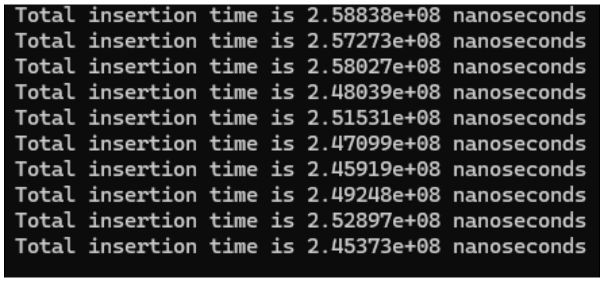


```
Total insertion time is 1.27885e+08 nanoseconds
Total insertion time is 1.23401e+08 nanoseconds
Total insertion time is 1.37318e+08 nanoseconds
Total insertion time is 1.33402e+08 nanoseconds
Total insertion time is 1.28823e+08 nanoseconds
Total insertion time is 1.2107e+08 nanoseconds
Total insertion time is 1.2302e+08 nanoseconds
Total insertion time is 1.29958e+08 nanoseconds
Total insertion time is 1.26984e+08 nanoseconds
Total insertion time is 1.24398e+08 nanoseconds
```

Figure 29

Time Taken = $(1.27 \times 10^8 + \ldots + 1.24 \times 10^8)/10 = 1.271 \times 10^8$ ns = **0.1271 s**
O(log(n)) = O(log(5000)) = **12.28 s**

- Large Dataset:
  The formation of the tree can be tested out in the code. Below is the analysis of the time complexity.



```
Total insertion time is 2.58838e+08 nanoseconds
Total insertion time is 2.57273e+08 nanoseconds
Total insertion time is 2.58027e+08 nanoseconds
Total insertion time is 2.48039e+08 nanoseconds
Total insertion time is 2.51531e+08 nanoseconds
Total insertion time is 2.47099e+08 nanoseconds
Total insertion time is 2.45919e+08 nanoseconds
Total insertion time is 2.49248e+08 nanoseconds
Total insertion time is 2.52897e+08 nanoseconds
Total insertion time is 2.45373e+08 nanoseconds
```

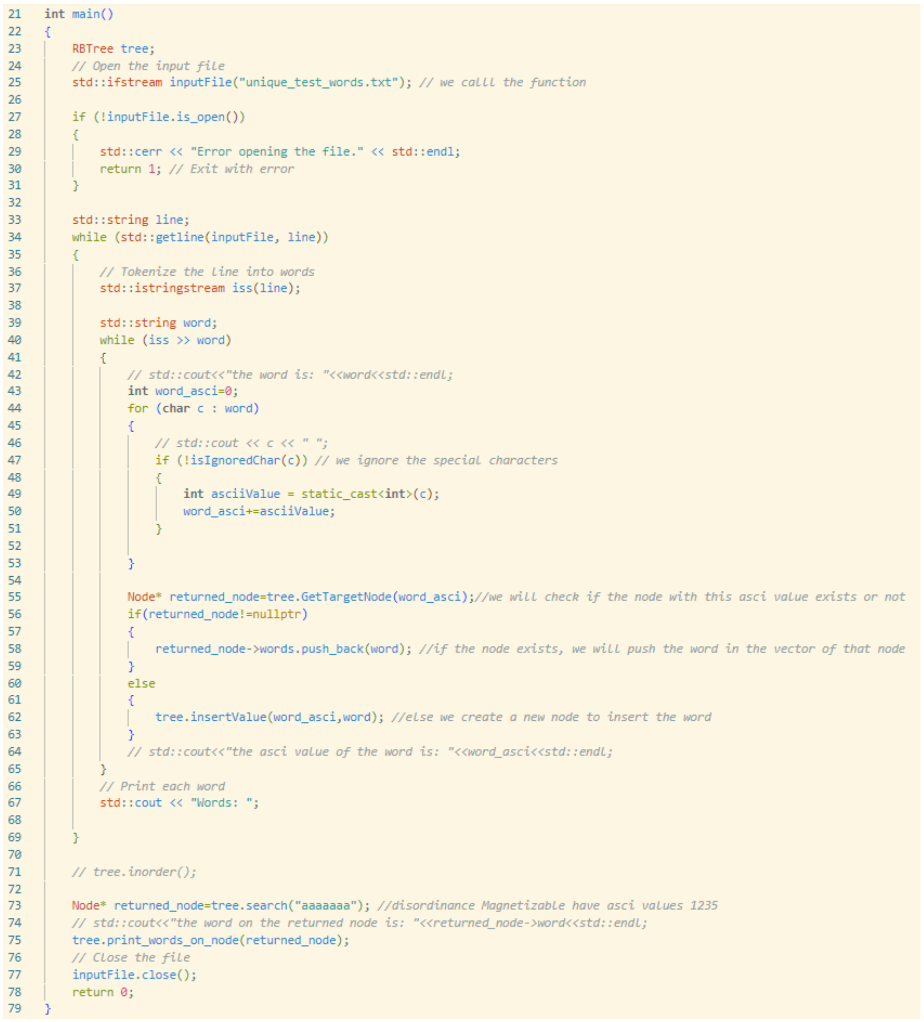
Figure 30

Time Taken = $(2.5883 \times 10^8 + \ldots + 2.4537 \times 10^7)/10 = 2.51 \times 10^8$ ns = **0.251 s**
O(log(n)) = O(log(10000)) = **13.28 s**

Red-Black Trees Formation

The Red-Black Trees formation and its functions have been explained in detail earlier. Now, we will read the words from the unqiue dataset and form a Red-Black tree.

The following figure 31 shows the code reading the unique database and forming the Red-Black Tree.



```cpp
int main()
{
    RBTree tree;
    // Open the input file
    std::ifstream inputFile("unique_test_words.txt"); // we calll the function

    if (!inputFile.is_open())
    {
        std::cerr << "Error opening the file." << std::endl;
        return 1; // Exit with error
    }

    std::string line;
    while (std::getline(inputFile, line))
    {
        // Tokenize the line into words
        std::istringstream iss(line);

        std::string word;
        while (iss >> word)
        {
            // std::cout<<"the word is: "<<word<<std::endl;
            int word_asci=0;
            for (char c : word)
            {
                // std::cout << c << " ";
                if (!isIgnoredChar(c)) // we ignore the special characters
                {
                    int asciiValue = static_cast<int>(c);
                    word_asci+=asciiValue;
                }

            }

            Node* returned_node=tree.GetTargetNode(word_asci);//we will check if the node with this asci value exists or not
            if(returned_node!=nullptr)
            {
                returned_node->words.push_back(word); //if the node exists, we will push the word in the vector of that node
            }
            else
            {
                tree.insertValue(word_asci,word); //else we create a new node to insert the word
            }
            // std::cout<<"the asci value of the word is: "<<word_asci<<std::endl;
        }
        // Print each word
        std::cout << "Words: ";

    }

    // tree.inorder();

    Node* returned_node=tree.search("aaaaaaa"); //disordinance Magnetizable have asci values 1235
    // std::cout<<"the word on the returned node is: "<<returned_node->word<<std::endl;
    tree.print_words_on_node(returned_node);
    // Close the file
    inputFile.close();
    return 0;
}
```

Figure 31

We are creating nodes for each entry (word) from the data-set by calculating the ASCII value of the word. Since the length and each word is kept different, it will result in a Unique ASCII value hence unique nodes will be formed.
The following figure 32 shows what the Data-Set comprises of,

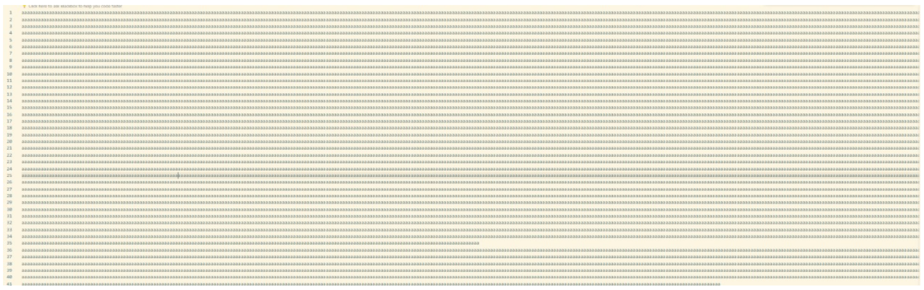


Figure 32

This is the sample dataset which contains only 100 entries. The first 48 entries are shown in which each word increments by 1 hence making it unique.
The below figure 33 shows us how the RBT is formed.

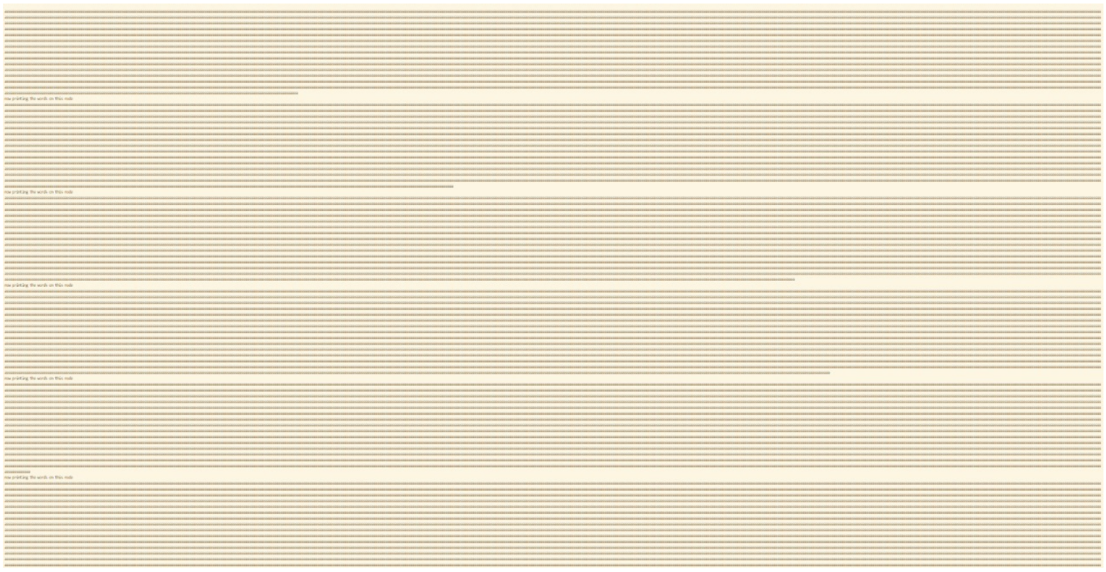
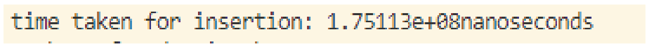
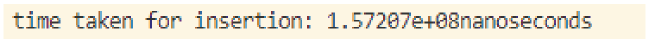


Figure 33

As we saw earlier, the complexity of the Insert function is O(log(n)). According to the above figure showing the time taken for insertion and number of nodes in the tree, we can analyze if the time taken is in fact correct. For the formation of the Tree, we have taken the sum of time taken by the insert function each time a node is inserted and displayed it in the terminal.
To get better accuracy, we have formed the tree with this dataset 10 times and have taken the average of time taken for insertion.

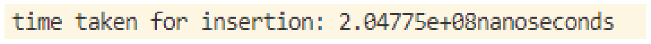


```
time taken for insertion: 1.75113e+08nanoseconds
```



```
time taken for insertion: 1.57207e+08nanoseconds
```

⋮



```
time taken for insertion: 2.04775e+08nanoseconds
```

Figure 34

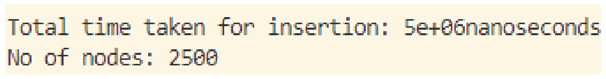
Time taken = $((1.44445 \times 10^8) + (1.75113 \times 10^8) + \ldots + (2.04775 \times 10^8))/10$
Time taken = $(15.815 \times 10^8)/10 = 15.815 \times 10^7$ ns = **0.158 s**

According to the complexity of the function, the time should be less than or equal to this,
$O(\log(n)) = O(\log(100)) =$ **6.64 s**

The time taken for forming a Red-Black Tree with 100 nodes is less than the time according to $O(\log(n))$ complexity of the insert function since **0.158s is less than 6.64s**.
(The time taken for the tree also includes the time for print statements. Calculations ahead won't include print statements time and so time shown may be lower.)

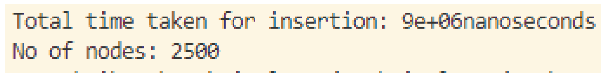
Below is the analysis of the formation of Red-Black Tree for Small, Medium, and Larger datasets.

- Small Dataset:
  The formation of the tree can be tested out in the code. Below is the analysis of the time complexity.



```
Total time taken for insertion: 5e+06nanoseconds
No of nodes: 2500
```

⋮

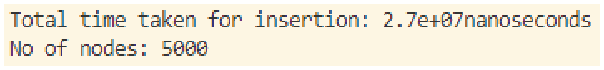


```
Total time taken for insertion: 9e+06nanoseconds
No of nodes: 2500
```

Figure 35

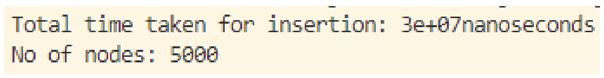
  Time Taken = $(5 \times 10^6 + \ldots + 9 \times 10^6)/10 = 7.833 \times 10^6$ ns = **0.0007833 s**
  $O(\log(n)) = O(\log(2500)) =$ **11.28 s**

- Medium Dataset:
  The formation of the tree can be tested out in the code. Below is the analysis of the time complexity.



```
Total time taken for insertion: 2.7e+07nanoseconds
No of nodes: 5000
```

$\vdots$

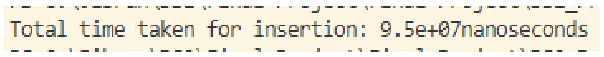


```
Total time taken for insertion: 3e+07nanoseconds
No of nodes: 5000
```

Figure 36

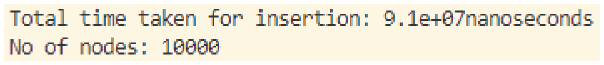
Time Taken $= (2.7 \times 10^7 + \ldots + 3.0 \times 10^7)/10 = 2 - .714 \times 10^7$ ns $=$ **0.02714 s**
$\mathrm{O}(\log(n)) = \mathrm{O}(\log(5000)) =$ **12.28 s**

- Large Dataset:
  The formation of the tree can be tested out in the code. Below is the analysis of the time complexity.



```
Total time taken for insertion: 9.5e+07nanoseconds
```

$\vdots$



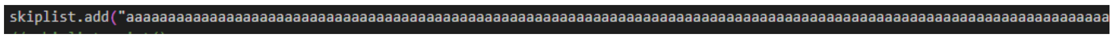
```
Total time taken for insertion: 9.1e+07nanoseconds
No of nodes: 10000
```

Figure 37

Time Taken $= (9.5 \times 10^7 + \ldots + 9.1 \times 10^7)/10 = 9.75 \times 10^7$ ns $=$ **0.0975 s**
$\mathrm{O}(\log(n)) = \mathrm{O}(\log(10000)) =$ **13.28 s**

Insert Function

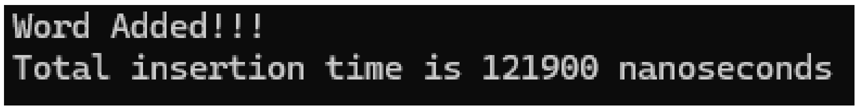
- Small Dataset Skiplist:
  We insert the following word in the tree,



```
skiplist.add("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 38

The time taken for the word to be inserted in the skiplist is,

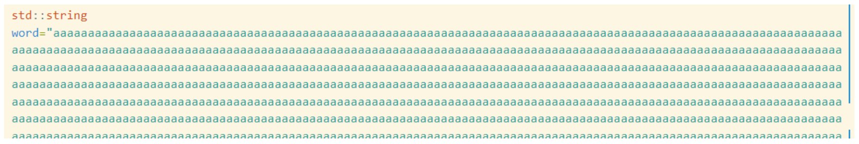


```
Word Added!!!
Total insertion time is 121900 nanoseconds
```

Figure 39

As we can see in the figure, the time taken for insertion of the word is **0.0001219s**

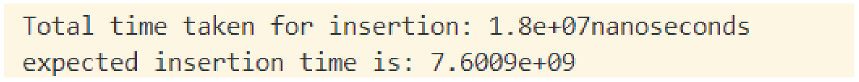
- Small Dataset RBT:
  We insert the following word in the tree,



```
std::string
word="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 40

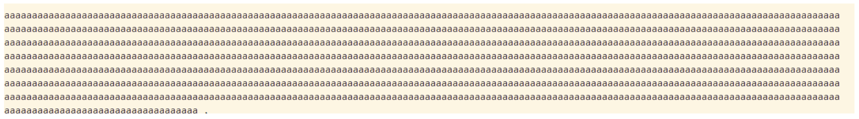
The time taken for the word to be inserted in the tree is,



```
Total time taken for insertion: 1.8e+07nanoseconds
expected insertion time is: 7.6009e+09
```

Figure 41

The word can be viewed in the tree now,

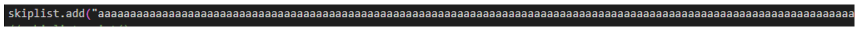


```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa .
```

Figure 42

As we can see in the figure, the time taken for insertion of the word is **0.018s** which is less than the expected time **7.6s**

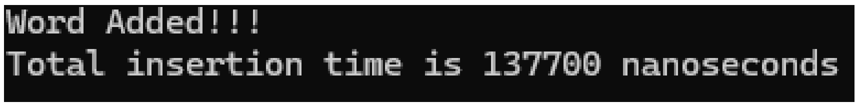
- Medium Dataset Skiplist:
  We insert the following word in the tree,



```
skiplist.add("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 43

The time taken for the word to be inserted in the tree is,



```
Word Added!!!
Total insertion time is 137700 nanoseconds
```

Figure 44

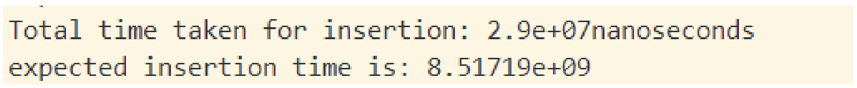
As we can see in the figure, the time taken for insertion of the word is **0.0001317s**

- Medium Dataset RBT:
  We insert the following word in the tree,

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa ,
```

Figure 45

The time taken for the word to be inserted in the tree is,



```
Total time taken for insertion: 2.9e+07nanoseconds
expected insertion time is: 8.51719e+09
```

Figure 46

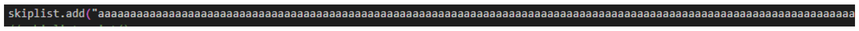
The word can be viewed in the tree now,

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa ,
```

Figure 47

As we can see in the figure, the time taken for insertion of the word is **0.029s** which is less than the expected time **8.51s**

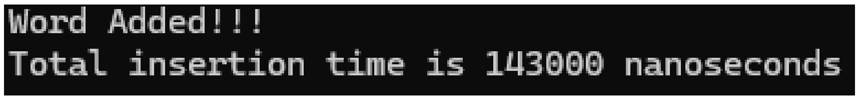
- Large Dataset Skiplist:
  We insert the following word in the tree,



```
skiplist.add("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 48

The time taken for the word to be inserted in the tree is,

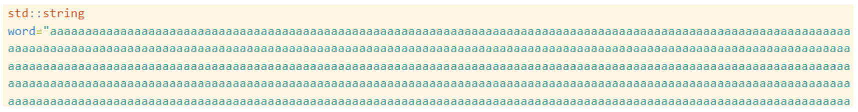


```
Word Added!!!
Total insertion time is 143000 nanoseconds
```

Figure 49

As we can see in the figure, the time taken for insertion of the word is **0.000143s**

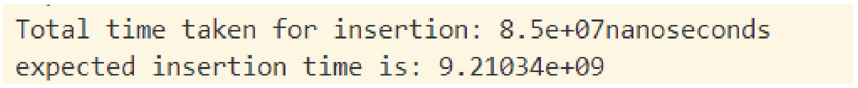
- Large Dataset RBT:
  We insert the following word in the tree,



```
std::string
word="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 50

The time taken for the word to be inserted in the tree is,



```
Total time taken for insertion: 8.5e+07nanoseconds
expected insertion time is: 9.21034e+09
```

Figure 51

The word can be viewed in the tree now,

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa ,
```

Figure 52

As we can see in the figure, the time taken for insertion of the word is **0.085** which is less than the expected time **9.21s**

Below we have shown the analysis of the Insertion function through a graph for both data structures;

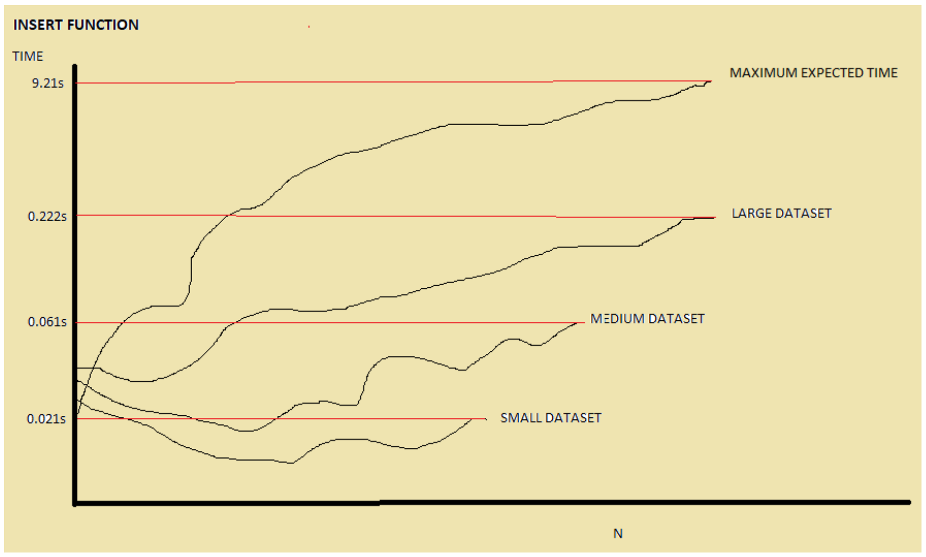


Figure 53

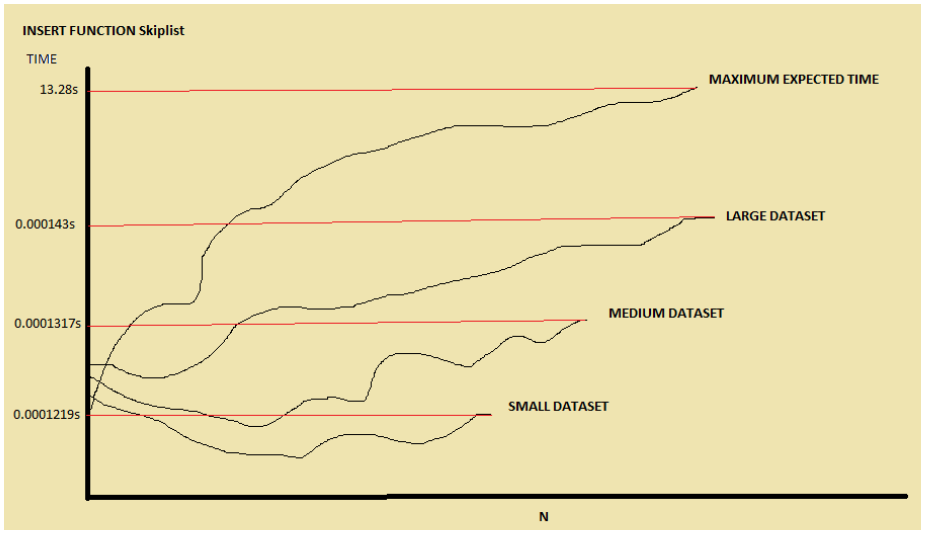
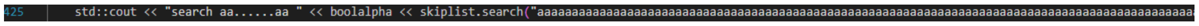


Figure 54

Search Function

In this sub-section, we will be comparing the search function of both data structures by using on the datasets.

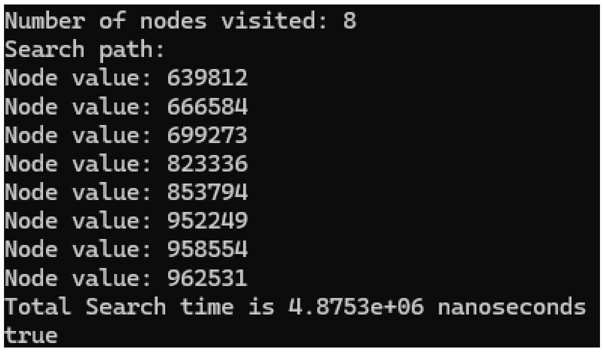
- Small Dataset Skiplist:
  We will be searching the max value node since it will take the longest time. We search for the this node,



```
std::cout << "search aa......aa " << boolalpha << skiplist.search("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

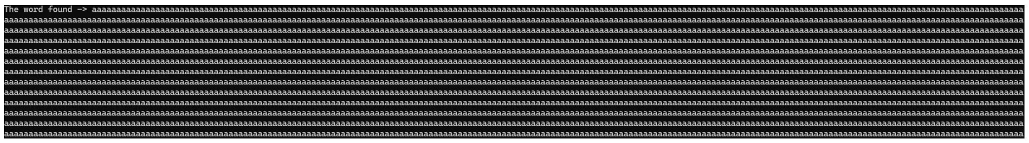
Figure 55

The time taken and the expected time is shown in the figure below.



```
Number of nodes visited: 8
Search path:
Node value: 639812
Node value: 666584
Node value: 699273
Node value: 823336
Node value: 853794
Node value: 952249
Node value: 958554
Node value: 962531
Total Search time is 4.8753e+06 nanoseconds
true
```

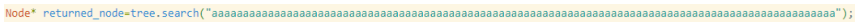
Figure 56



```
The word found -> aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 57

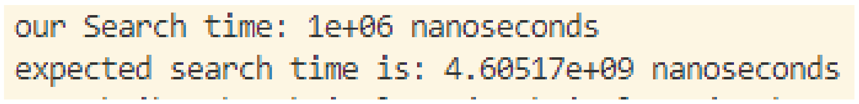
- Small Dataset RBT:
  When we search for the leaf node in the



```
Node* returned_node=tree.search("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa");
```

Figure 58

The time taken and the expected time is shown in the figure below.

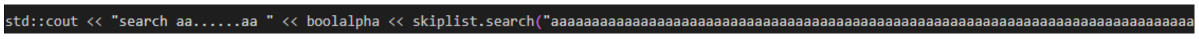


```
our Search time: 1e+06 nanoseconds
expected search time is: 4.60517e+09 nanoseconds
```

Figure 59

As we can see in the figure, the time taken for insertion of the word is **0.0007** which is less than the expected time **7.601s**

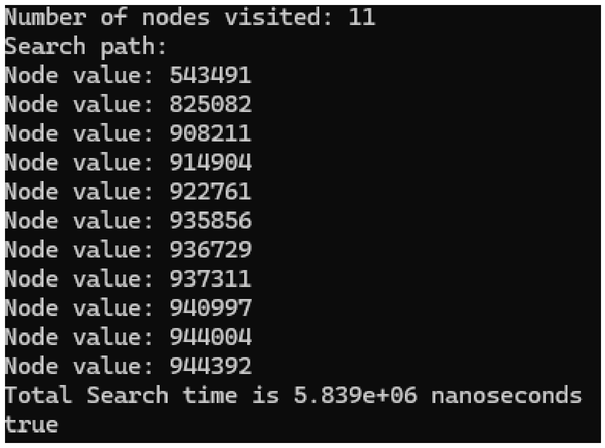
- Medium Dataset Skiplist:
  When we search for the max value node in the



```
std::cout << "search aa......aa " << boolalpha << skiplist.search("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 60

The time taken and the expected time is shown in the figure below.



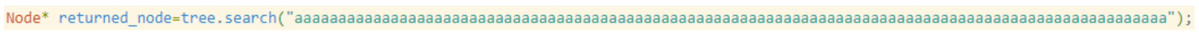
```
Number of nodes visited: 11
Search path:
Node value: 543491
Node value: 825082
Node value: 908211
Node value: 914904
Node value: 922761
Node value: 935856
Node value: 936729
Node value: 937311
Node value: 940997
Node value: 944004
Node value: 944392
Total Search time is 5.839e+06 nanoseconds
true
```

Figure 61

```
Value found -> 944392
The word found -> aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 62

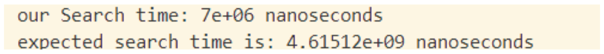
- Medium Dataset RBT:
  When we search for the leaf node in the



```
Node* returned_node=tree.search("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa");
```

Figure 63

The time taken and the expected time is shown in the figure below.

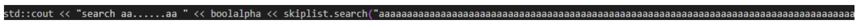


```
our Search time: 7e+06 nanoseconds
expected search time is: 4.61512e+09 nanoseconds
```

Figure 64

As we can see in the figure, the time taken for insertion of the word is **0.0007** which is less than the expected time **4.61s**

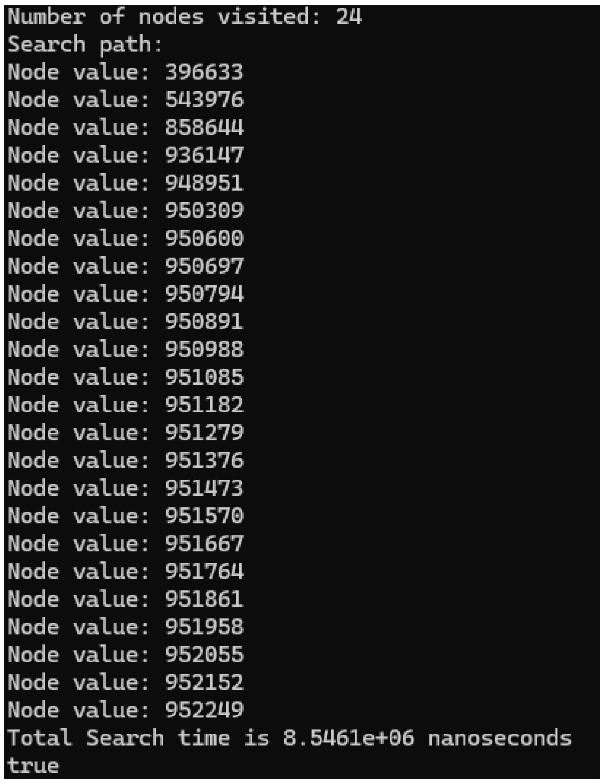
- Large Dataset Skiplist:
  We will be searching the max value node since it will take the longest time. We search for that node,



```
std::cout << "search aa......aa " << boolalpha << skiplist.search("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

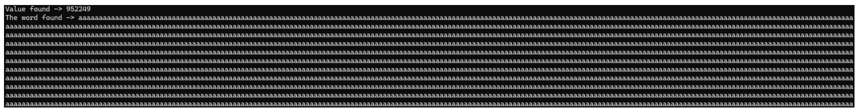
Figure 65

The time taken and the expected time is shown in the figure below.



```
Number of nodes visited: 24
Search path:
Node value: 396633
Node value: 543976
Node value: 858644
Node value: 936147
Node value: 948951
Node value: 950309
Node value: 950600
Node value: 950697
Node value: 950794
Node value: 950891
Node value: 950988
Node value: 951085
Node value: 951182
Node value: 951279
Node value: 951376
Node value: 951473
Node value: 951570
Node value: 951667
Node value: 951764
Node value: 951861
Node value: 951958
Node value: 952055
Node value: 952152
Node value: 952249
Total Search time is 8.5461e+06 nanoseconds
true
```

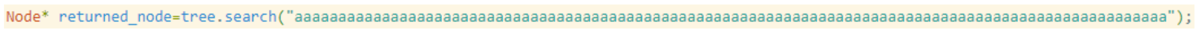
Figure 66



```
Value found -> 952249
The word found -> aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 67

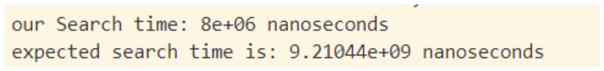
- Large Dataset RBT:
  When we search for the leaf node in the



```
Node* returned_node=tree.search("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa");
```

Figure 68

The time taken and the expected time is shown in the figure below.



```
our Search time: 8e+06 nanoseconds
expected search time is: 9.21044e+09 nanoseconds
```

Figure 69

As we can see in the figure, the time taken for insertion of the word is **0.0008** which is less than the expected time **9.21s**

Below we have shown the analysis of the Search function through a graph for both data structures;

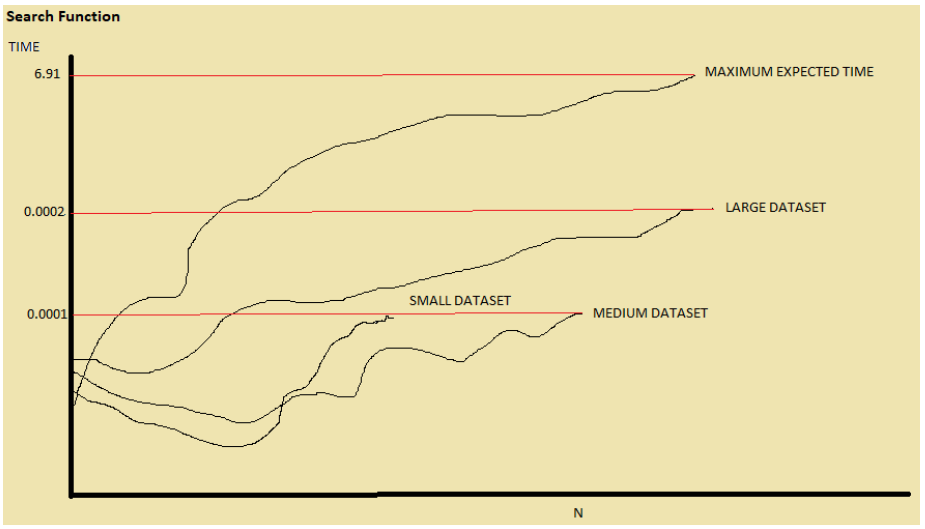


Figure 70

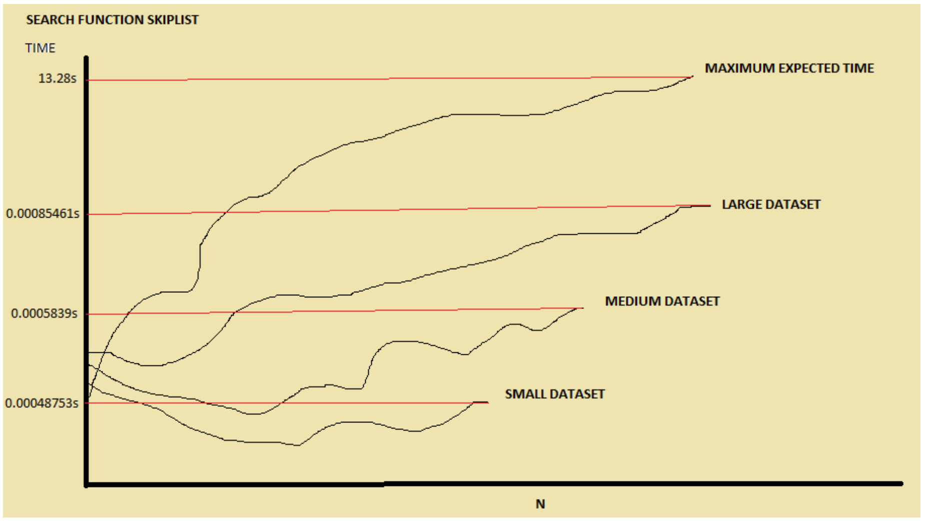


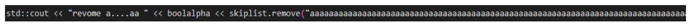
Figure 71

Delete Function

For the analysis of the delete function, we will be deleting the node/value which we inserted previously for all datasets.

- Small Dataset Skiplist:

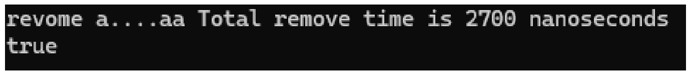
  We inserted the following word in the Skiplist previously, and now we are deleting it



```
std::cout << "revome a....aa " << boolalpha << skiplist.remove("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 72

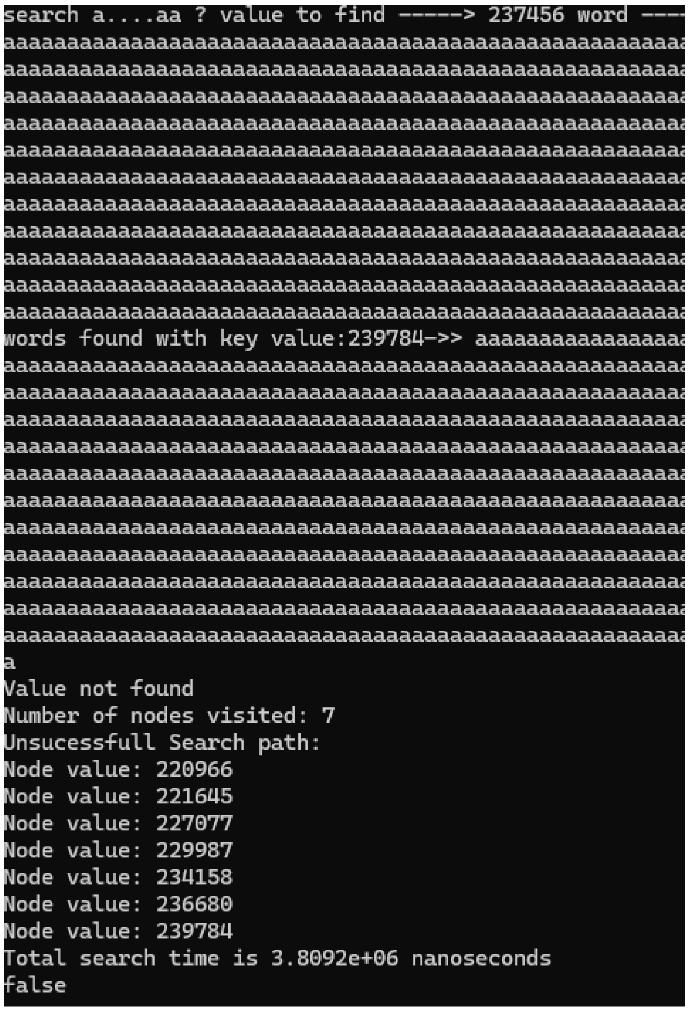
Now we call the delete function to delete this node, the result are following;



```
revome a....aa Total remove time is 2700 nanoseconds
true
```

Figure 73

To check whether the value has been deleted or not, we search for that value in the skiplist,



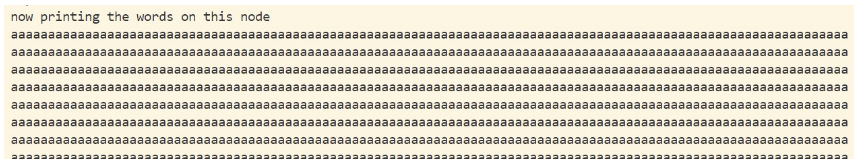
```
search a....aa ? value to find -----> 237456 word ----
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
words found with key value:239784->> aaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
a
Value not found
Number of nodes visited: 7
Unsucessfull Search path:
Node value: 220966
Node value: 221645
Node value: 227077
Node value: 229987
Node value: 234158
Node value: 236680
Node value: 239784
Total search time is 3.8092e+06 nanoseconds
false
```

Figure 74

The output of the search function is false for the node we deleted.

As we can see in the figure, the time taken for deletion of the word is **0.000380s**

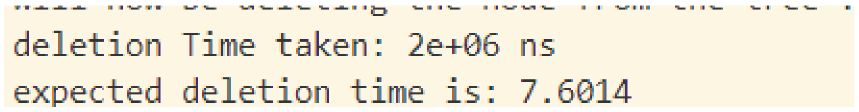
- Small Dataset RBT:
  We inserted the following word in the tree previously,



```
now printing the words on this node
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 75

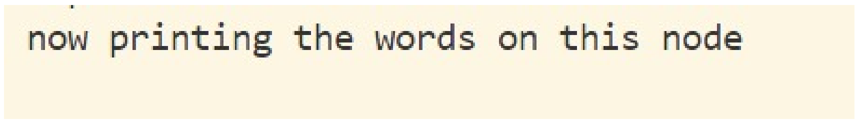
Now we call the delete function to delete this node,



```
will now be deleting the node from the tree :
deletion Time taken: 2e+06 ns
expected deletion time is: 7.6014
```

Figure 76

To check whether the value has been deleted or not, we search for that value in the tree, Now we call the delete function to delete this node,



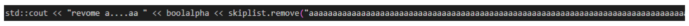
```
now printing the words on this node
```

Figure 77

There is nothing on the output which proves that the node does not exist in the tree hence it has been deleted.
As we can see in the figure, the time taken for deletion of the word is **0.0002s** which is less than the expected time **7.60s**

- Medium Dataset Skiplist:

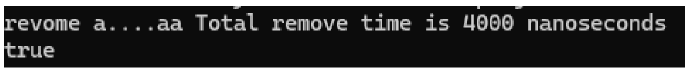
  We inserted the following word in the skip list previously, now we are deleting it



```
std::cout << "revome a....aa " << boolalpha << skiplist.remove("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 78

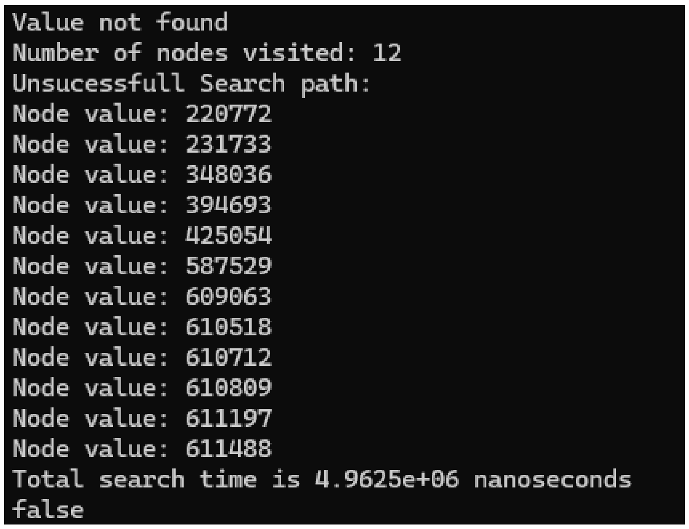
Now we call the delete function to delete this node, the results are following



```
revome a....aa Total remove time is 4000 nanoseconds
true
```

Figure 79

To check whether the value has been deleted or not, we search for that value in the skiplist,



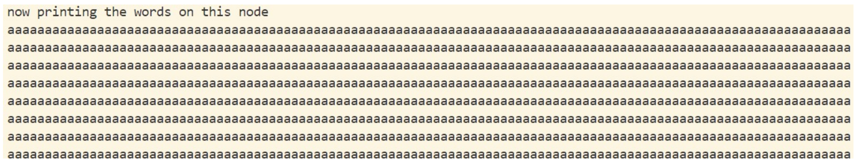
```
Value not found
Number of nodes visited: 12
Unsucessfull Search path:
Node value: 220772
Node value: 231733
Node value: 348036
Node value: 394693
Node value: 425054
Node value: 587529
Node value: 609063
Node value: 610518
Node value: 610712
Node value: 610809
Node value: 611197
Node value: 611488
Total search time is 4.9625e+06 nanoseconds
false
```

Figure 80

The output of the search function is false for the node we deleted.

As we can see in the figure, the time taken for deletion of the word is **0.00049s**

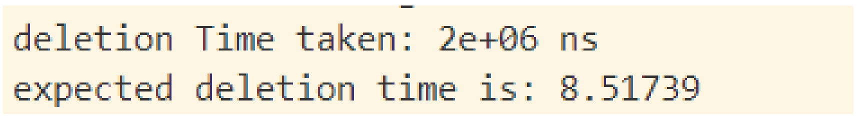
- Medium Dataset RBT:
  We inserted the following word in the tree previously,



```
now printing the words on this node
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 81

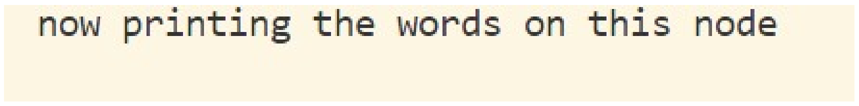
Now we call the delete function to delete this node,



```
deletion Time taken: 2e+06 ns
expected deletion time is: 8.51739
```

Figure 82

To check whether the value has been deleted or not, we search for that value in the tree, Now we call the delete function to delete this node,



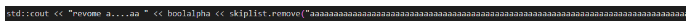
```
now printing the words on this node
```

Figure 83

There is nothing on the output which proves that the node does not exist in the tree hence it has been deleted.
As we can see in the figure, the time taken for deletion of the word is **0.0002s** which is less than the expected time **8.51s**

- Large Dataset Skiplist:

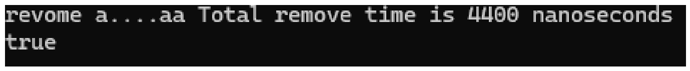
  We inserted the following word in the tree previously, now we are deleting it



```
std::cout << "revome a....aa " << boolalpha << skiplist.remove("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 84

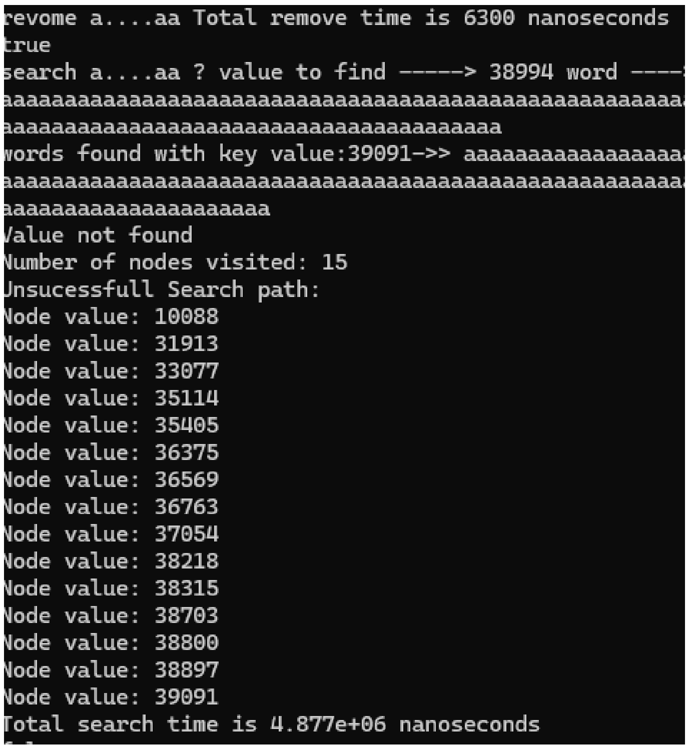
Now we call the delete function to delete this node, the results are following



```
revome a....aa Total remove time is 4400 nanoseconds
true
```

Figure 85

To check whether the value has been deleted or not, we search for that value in the skiplist,



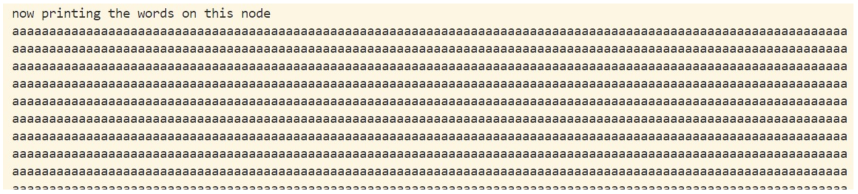
```
revome a....aa Total remove time is 6300 nanoseconds
true
search a....aa ? value to find -----> 38994 word ----
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
words found with key value:39091->> aaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaa
Value not found
Number of nodes visited: 15
Unsucessfull Search path:
Node value: 10088
Node value: 31913
Node value: 33077
Node value: 35114
Node value: 35405
Node value: 36375
Node value: 36569
Node value: 36763
Node value: 37054
Node value: 38218
Node value: 38315
Node value: 38703
Node value: 38800
Node value: 38897
Node value: 39091
Total search time is 4.877e+06 nanoseconds
```

Figure 86

The output of the search function is false for the node we deleted.

As we can see in the figure, the time taken for deletion of the word is **0.00049s**

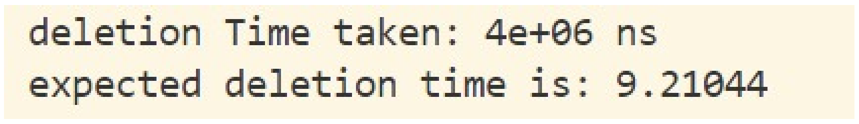
- Large Dataset RBT:
  We inserted the following word in the tree previously,



```
now printing the words on this node
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Figure 87

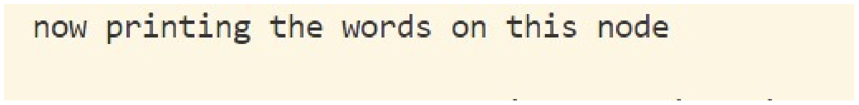
Now we call the delete function to delete this node,



```
deletion Time taken: 4e+06 ns
expected deletion time is: 9.21044
```

Figure 88

To check whether the value has been deleted or not, we search for that value in the tree, Now we call the delete function to delete this node,



```
now printing the words on this node
```

Figure 89

There is nothing on the output which proves that the node does not exist in the tree hence it has been deleted.
As we can see in the figure, the time taken for deletion of the word is **0.0004s** which is less than the expected time **9.21s**

The table below shows the time-complexity for each functions for this dataset,

Function-Dataset	**Red Black Tree**	**Skip-List**
Insert-Small	**0.0180s**	0.00012s
Insert-Medium	0.029s	**0.0001s**
Insert-Large	0.085s	**0.0001s**
Search-Small	0.0007s	0.0004s
Search-Medium	0.0007s	**0.0005s**
Search-Large	**0.00080s**	0.00084s
Delete-Small	**0.00020s**	0.00030s
Delete-Medium	**0.00020s**	0.00040s
Delete-Large	**0.00040s**	0.00049s

Table 1: Time complexity for Unique Dataset

Below we have shown the analysis of the Deletion function through a graph for both data structures;

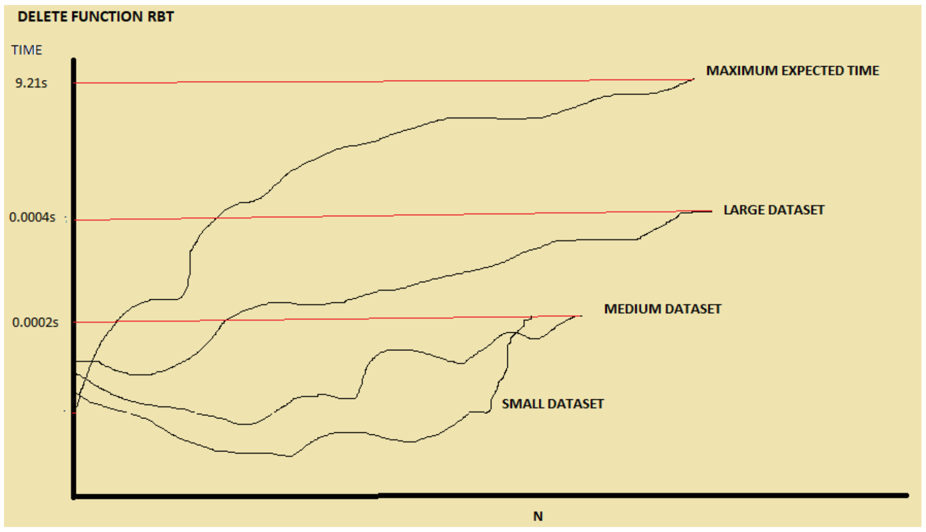


Figure 90

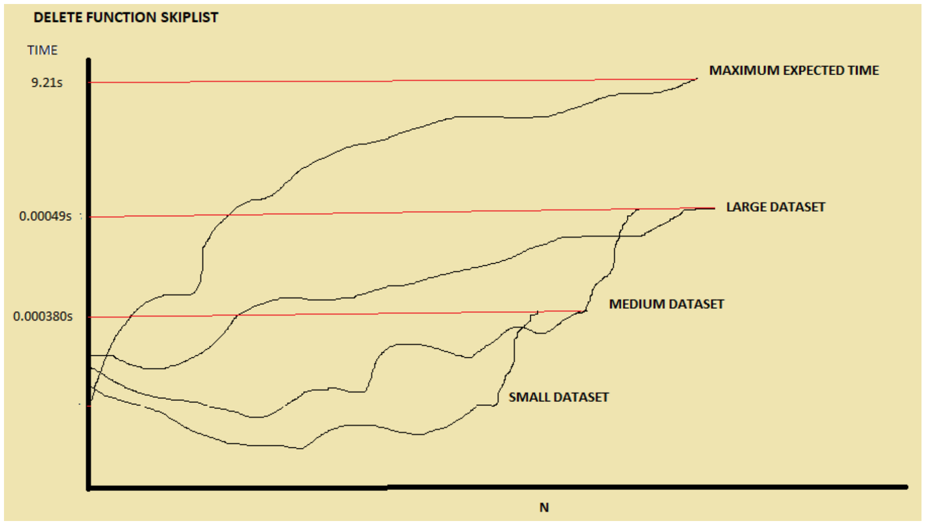
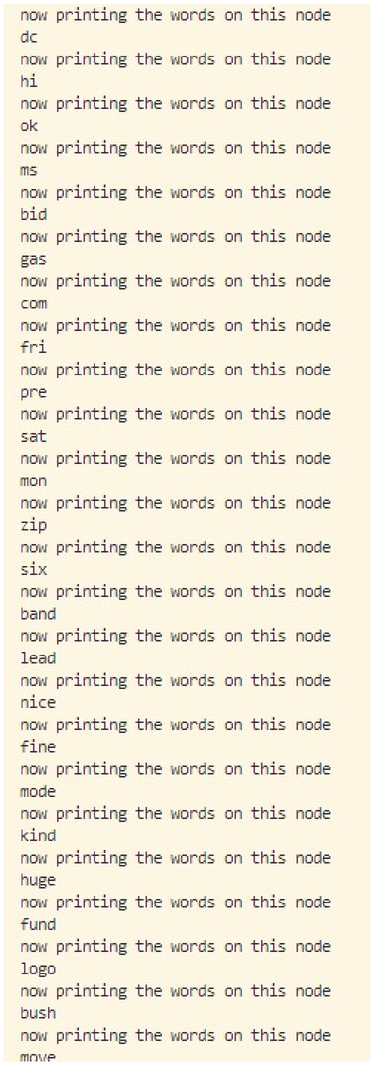


Figure 91

Analysis of Red-Black Trees vs Skiplist (Duplicate Entries Data-Set)

Skiplist Formation

The code for the formation of Skip List is shown in the previous section where we did the analysis for Unique dataset. Now the data structure will be tested out on the Unique words which may result in duplicate values if they have similar ASCII values. We are creating nodes on the basis of ASCII values so if more then one node have similar ASCII values, the words will be inserted in a vector. The figure below shows us how the Skiplist is formed.

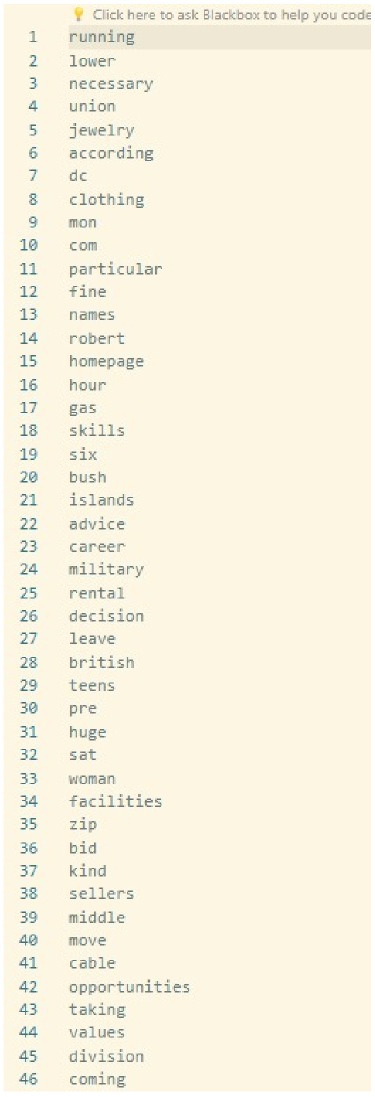


```
now printing the words on this node
dc
now printing the words on this node
hi
now printing the words on this node
ok
now printing the words on this node
ms
now printing the words on this node
bid
now printing the words on this node
gas
now printing the words on this node
com
now printing the words on this node
fri
now printing the words on this node
pre
now printing the words on this node
sat
now printing the words on this node
mon
now printing the words on this node
zip
now printing the words on this node
six
now printing the words on this node
band
now printing the words on this node
lead
now printing the words on this node
nice
now printing the words on this node
fine
now printing the words on this node
mode
now printing the words on this node
kind
now printing the words on this node
huge
now printing the words on this node
fund
now printing the words on this node
logo
now printing the words on this node
bush
now printing the words on this node
move
```

Figure 91

Red-Black Tree Formation

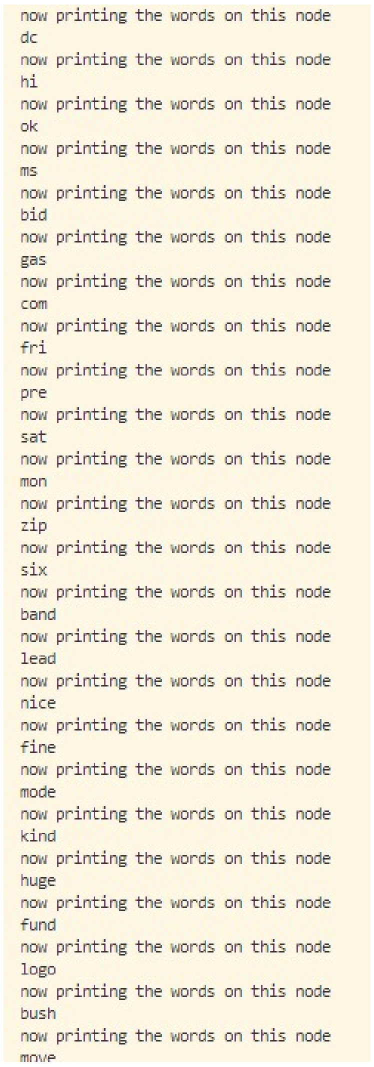
The code for the formation of the Red-Black Tree is shown in the previous section where we did the analysis for Unique dataset. Now the data structure will be tested out on the Unique words which may result in duplicate values if they have similar ASCII values. We are creating nodes on the basis of ASCII values so if more then one node have similar ASCII values, the words will be inserted in a vector.
The following figure shows what the Data-Set comprises of,



```
running
lower
necessary
union
jewelry
according
dc
clothing
mon
com
particular
fine
names
robert
homepage
hour
gas
skills
six
bush
islands
advice
career
military
rental
decision
leave
british
teens
pre
huge
sat
woman
facilities
zip
bid
kind
sellers
middle
move
cable
opportunities
taking
values
division
coming
```

Figure 92

This is the sample dataset which contains only 100 entries. The first 46 entries. The figure below shows us how the RBT is formed.

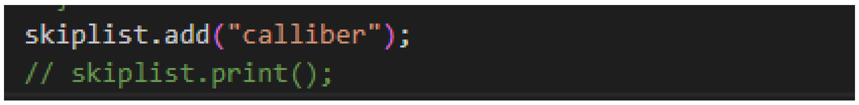


```
now printing the words on this node
dc
now printing the words on this node
hi
now printing the words on this node
ok
now printing the words on this node
ms
now printing the words on this node
bid
now printing the words on this node
gas
now printing the words on this node
com
now printing the words on this node
fri
now printing the words on this node
pre
now printing the words on this node
sat
now printing the words on this node
mon
now printing the words on this node
zip
now printing the words on this node
six
now printing the words on this node
band
now printing the words on this node
lead
now printing the words on this node
nice
now printing the words on this node
fine
now printing the words on this node
mode
now printing the words on this node
kind
now printing the words on this node
huge
now printing the words on this node
fund
now printing the words on this node
logo
now printing the words on this node
bush
now printing the words on this node
move
```

Figure 93

Insert Function

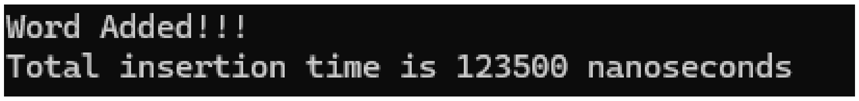
- Small Dataset Skiplist: We will insert the word 'calliber'. The figure below shows the process,



```
skiplist.add("calliber");
// skiplist.print();
```

Figure 94

So the time taken for the insertion is,

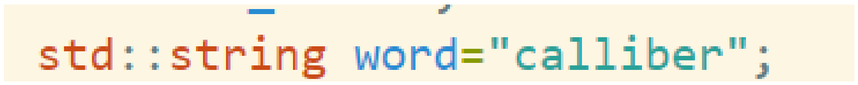


```
Word Added!!!
Total insertion time is 123500 nanoseconds
```

Figure 95

As we can see in the figure, the time taken for insertion of the word is **0.0001235s**

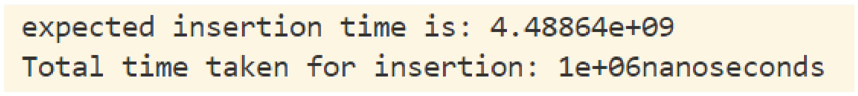
- Small Dataset RBT: We will insert the word 'calliber'. The figure below shows the process,



```
std::string word="calliber";
```

Figure 96

So the time taken for the insertion is,

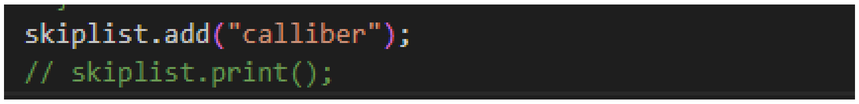


```
expected insertion time is: 4.48864e+09
Total time taken for insertion: 1e+06nanoseconds
```

Figure 97

As we can see in the figure, the time taken for deletion of the word is **0.0001s** which is less than the expected time **4.4886s**

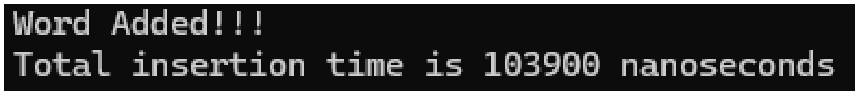
- Medium Dataset Skiplist: We will insert the word 'calliber'. The figure below shows the process,



```
skiplist.add("calliber");
// skiplist.print();
```

Figure 98

So the time taken for the insertion is,

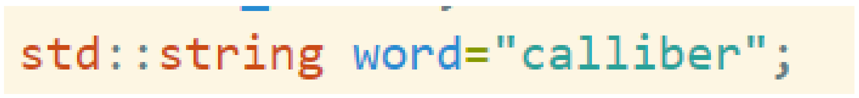


```
Word Added!!!
Total insertion time is 103900 nanoseconds
```

Figure 99

As we can see in the figure, the time taken for insertion of the word is **0.0001039s**

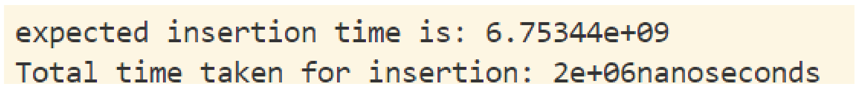
- Medium Dataset RBT: We will insert the word 'calliber'. The figure below shows the process,



```
std::string word="calliber";
```

Figure 100

So the time taken for the insertion is,

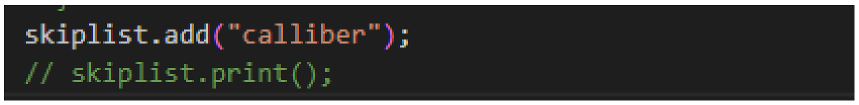


```
expected insertion time is: 6.75344e+09
Total time taken for insertion: 2e+06nanoseconds
```

Figure 101

As we can see in the figure, the time taken for deletion of the word is **0.0002s** which is less than the expected time **6.7534s**

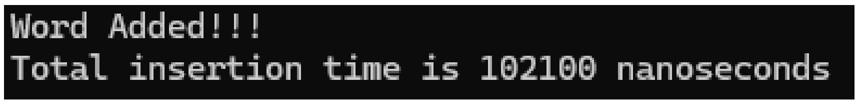
- Large Dataset Skiplist: We will insert the word 'calliber'. The figure below shows the process,



```
skiplist.add("calliber");
// skiplist.print();
```

Figure 102

So the time taken for the insertion is,

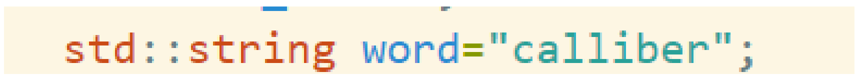


```
Word Added!!!
Total insertion time is 102100 nanoseconds
```

Figure 103

As we can see in the figure, the time taken for insertion of the word is **0.0001021s**

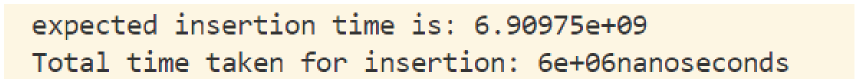
- Large Dataset RBT: We will insert the word 'calliber'. The figure below shows the process,



```
std::string word="calliber";
```

Figure 104

So the time taken for the insertion is,



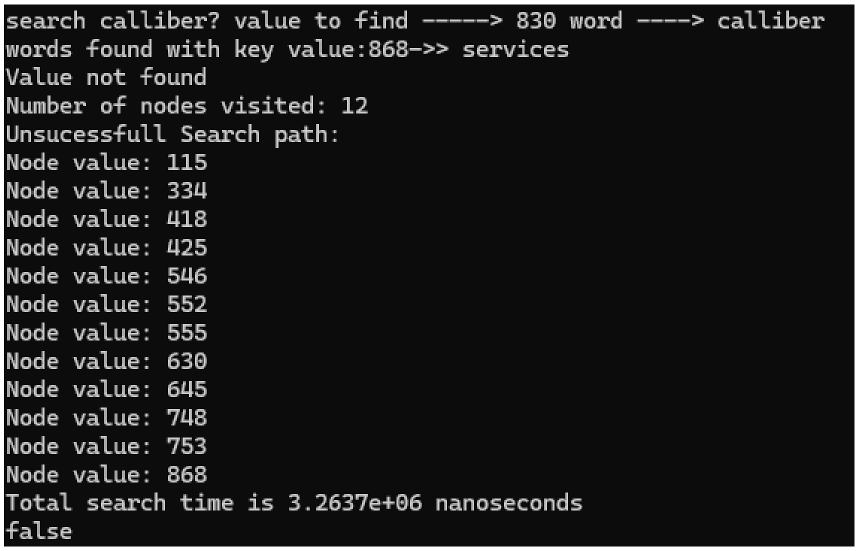
```
expected insertion time is: 6.90975e+09
Total time taken for insertion: 6e+06nanoseconds
```

Figure 105

As we can see in the figure, the time taken for deletion of the word is **0.0006s** which is less than the expected time **6.909s**

Search Function

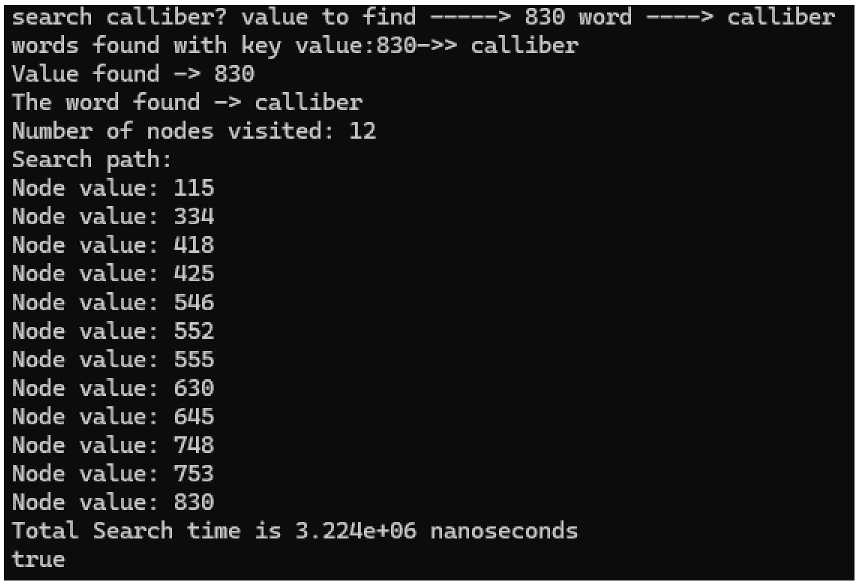
- Small Dataset Skiplist:
  In the previous subsection, we inserted the word 'calliber' into the small dataset based Skiplist, now we will search for it. function" section above. Before insertion,



```
search calliber? value to find -----> 830 word ----> calliber
words found with key value:868->> services
Value not found
Number of nodes visited: 12
Unsucessfull Search path:
Node value: 115
Node value: 334
Node value: 418
Node value: 425
Node value: 546
Node value: 552
Node value: 555
Node value: 630
Node value: 645
Node value: 748
Node value: 753
Node value: 868
Total search time is 3.2637e+06 nanoseconds
false
```

Figure 106

After insertion of the word,



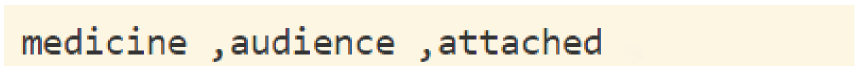
```
search calliber? value to find -----> 830 word ----> calliber
words found with key value:830->> calliber
Value found -> 830
The word found -> calliber
Number of nodes visited: 12
Search path:
Node value: 115
Node value: 334
Node value: 418
Node value: 425
Node value: 546
Node value: 552
Node value: 555
Node value: 630
Node value: 645
Node value: 748
Node value: 753
Node value: 830
Total Search time is 3.224e+06 nanoseconds
true
```

Figure 107

The time taken to search for the word is:

As we can see in the figure, the time taken for searching of the word before insertion is **0.00326s** and after insertion is **0.00324s**

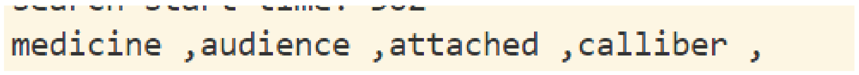
- Small Dataset RBT:
  In the previous subsection, we inserted the word 'calliber' into the small dataset based RBT, now we will search for it. function" section above. Before insertion,



```
medicine ,audience ,attached
```

Figure 108

After insertion of the word,

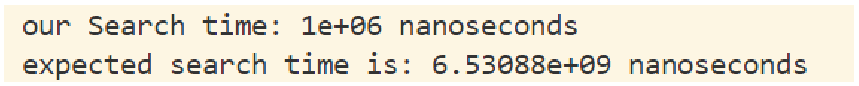


```
medicine ,audience ,attached ,calliber ,
```

Figure 109

The word 'calliber' was added to the node with the same ascii value and since that node already existed, the word was pushed into the vector.

The time taken for the word to be searched is,

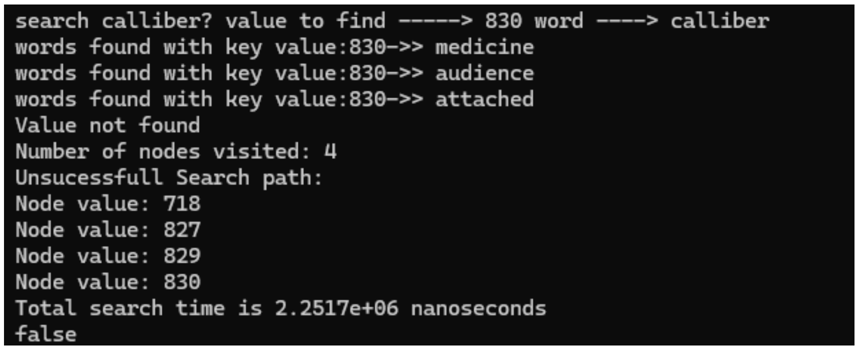


```
our Search time: 1e+06 nanoseconds
expected search time is: 6.53088e+09 nanoseconds
```

Figure 110

As we can see in the figure, the time taken for deletion of the word is **0.0001s** which is less than the expected time **6.53s**

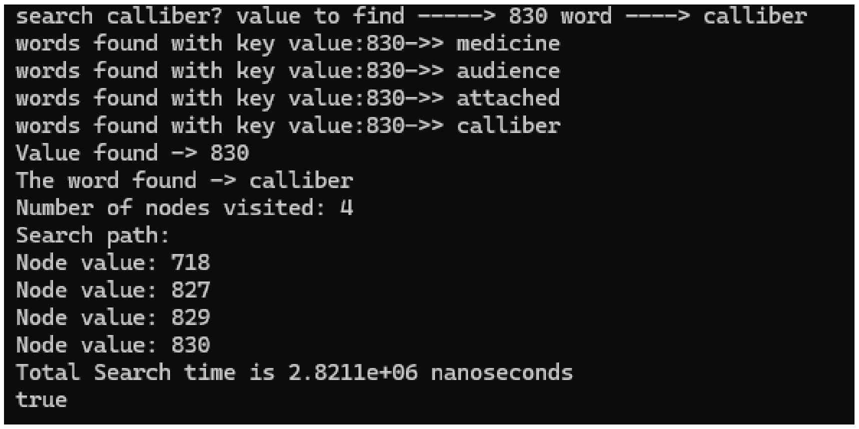
- Medium Dataset Skiplist
  In the previous subsection, we inserted the word 'calliber' into the medium dataset based Skiplist, now we will search for it. function" section above. Before insertion,



```
search calliber? value to find -----> 830 word ----> calliber
words found with key value:830->> medicine
words found with key value:830->> audience
words found with key value:830->> attached
Value not found
Number of nodes visited: 4
Unsucessfull Search path:
Node value: 718
Node value: 827
Node value: 829
Node value: 830
Total search time is 2.2517e+06 nanoseconds
false
```

Figure 111

After insertion of the word,

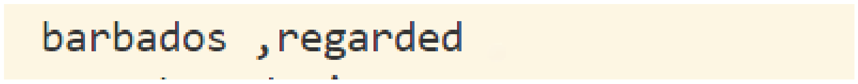


```
search calliber? value to find -----> 830 word ----> calliber
words found with key value:830->> medicine
words found with key value:830->> audience
words found with key value:830->> attached
words found with key value:830->> calliber
Value found -> 830
The word found -> calliber
Number of nodes visited: 4
Search path:
Node value: 718
Node value: 827
Node value: 829
Node value: 830
Total Search time is 2.8211e+06 nanoseconds
true
```

Figure 112

The time taken to search for the word is, As we can see in the figure, the time taken for searching of the word before insertion is **0.0022517s** and after insertion is **0.0028211s**

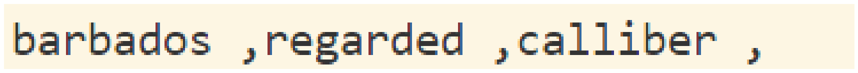
- Medium Dataset RBT:
  In the previous subsection, we inserted the word 'calliber' into the medium dataset based RBT, now we will search for it. function" section above. Before insertion,



```
barbados ,regarded
```

Figure 113

After insertion of the word,

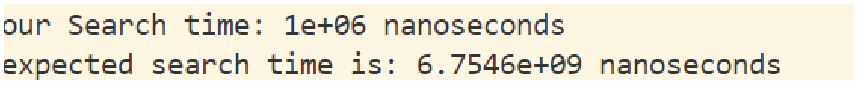


```
barbados ,regarded ,calliber ,
```

Figure 114

The word 'calliber' was added to the node with the same ascii value and since that node already existed, the word was pushed into the vector.

The time taken for the word to be searched is,

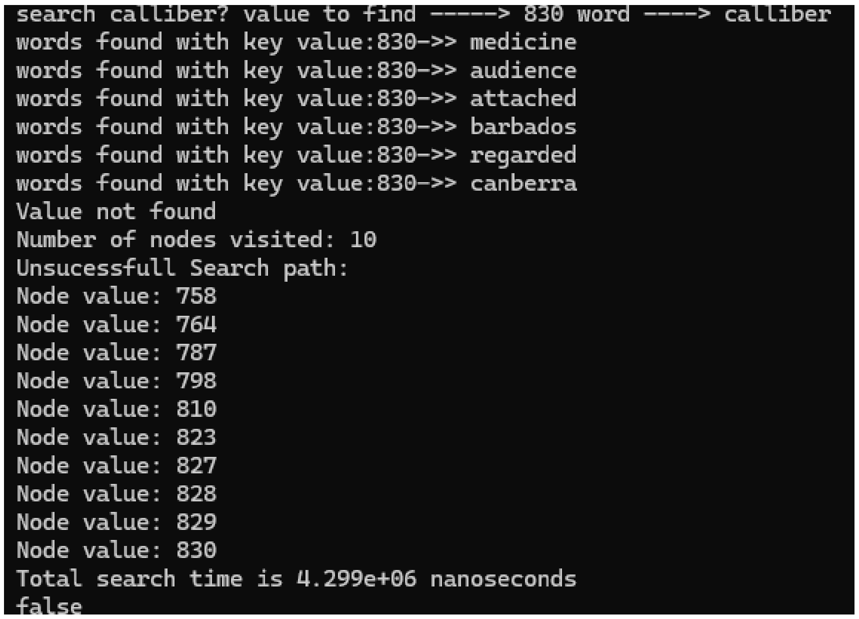


```
our Search time: 1e+06 nanoseconds
expected search time is: 6.7546e+09 nanoseconds
```

Figure 115

As we can see in the figure, the time taken for deletion of the word is **0.0001s** which is less than the expected time **6.754s**

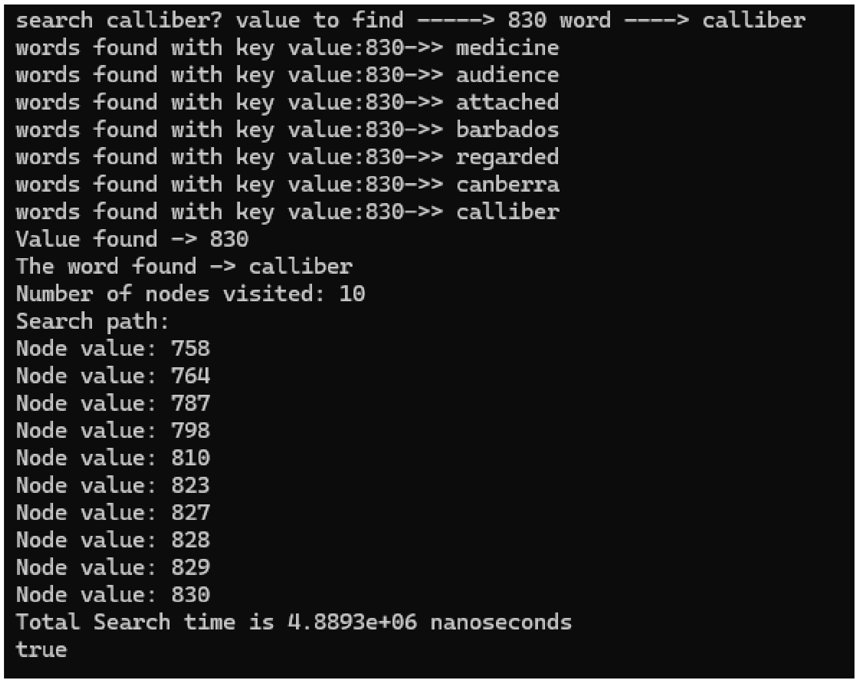
- Large Dataset Skiplist
  In the previous subsection, we inserted the word 'calliber' into the large dataset based Skiplist, now we will search for it. function" section above. Before insertion,



```
search calliber? value to find -----> 830 word ----> calliber
words found with key value:830->> medicine
words found with key value:830->> audience
words found with key value:830->> attached
words found with key value:830->> barbados
words found with key value:830->> regarded
words found with key value:830->> canberra
Value not found
Number of nodes visited: 10
Unsucessfull Search path:
Node value: 758
Node value: 764
Node value: 787
Node value: 798
Node value: 810
Node value: 823
Node value: 827
Node value: 828
Node value: 829
Node value: 830
Total search time is 4.299e+06 nanoseconds
false
```

Figure 116

After insertion of the word,



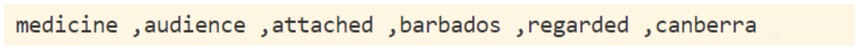
```
search calliber? value to find -----> 830 word ----> calliber
words found with key value:830->> medicine
words found with key value:830->> audience
words found with key value:830->> attached
words found with key value:830->> barbados
words found with key value:830->> regarded
words found with key value:830->> canberra
words found with key value:830->> calliber
Value found -> 830
The word found -> calliber
Number of nodes visited: 10
Search path:
Node value: 758
Node value: 764
Node value: 787
Node value: 798
Node value: 810
Node value: 823
Node value: 827
Node value: 828
Node value: 829
Node value: 830
Total Search time is 4.8893e+06 nanoseconds
true
```

Figure 117

The time taken to search for the word is,

As we can see in the figure, the time taken for searching of the word before insertion is **0.004299s** and after insertion is **0.004889s**

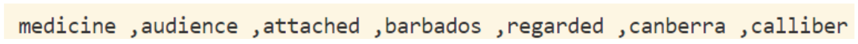
- Large Dataset RBT:
  In the previous subsection, we inserted the word 'calliber' into the large dataset based RBT, now we will search for it. function" section above. Before insertion,



```
medicine ,audience ,attached ,barbados ,regarded ,canberra
```

Figure 118

After insertion of the word,

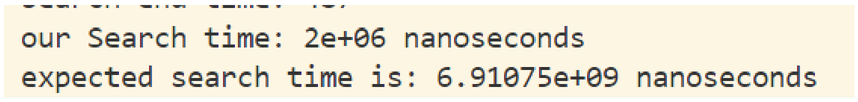


```
medicine ,audience ,attached ,barbados ,regarded ,canberra ,calliber
```

Figure 119

The word 'calliber' was added to the node with the same ascii value and since that node already existed, the word was pushed into the vector.

The time taken for the word to be searched is,



```
our Search time: 2e+06 nanoseconds
expected search time is: 6.91075e+09 nanoseconds
```

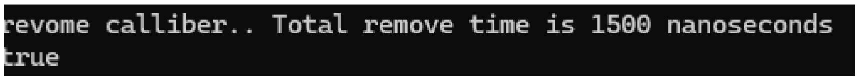
Figure 120

As we can see in the figure, the time taken for deletion of the word is **0.0002s** which is less than the expected time **6.910s**

Delete Function

Now we will be deleting the same word which were inserted in each dataset based RBT.

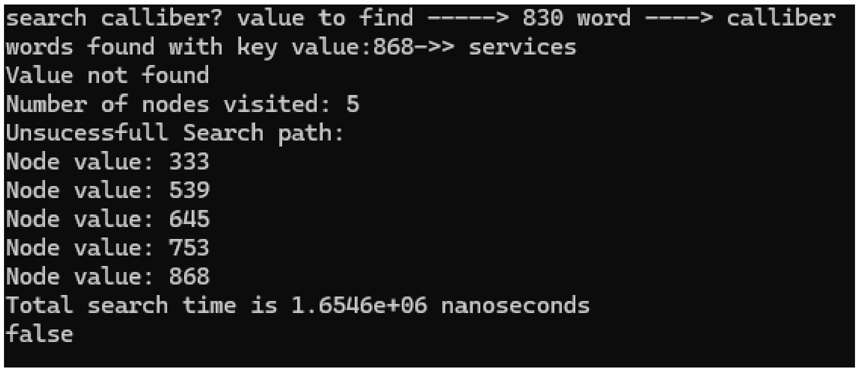
- Small Dataset Skiplist:
  We inserted the word 'calliber' in the Skiplist for this dataset. Now we will be deleting the word. The node had the word previously in its corresponding vector. After deleting the word we inserted,



```
revome calliber.. Total remove time is 1500 nanoseconds
true
```

Figure 121

Now we check if the word is deleted from the skiplist or not, we search the same word in the skip list and see the result, if false then the word was deleted.

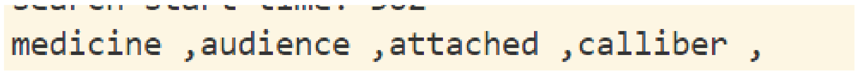


```
search calliber? value to find -----> 830 word ----> calliber
words found with key value:868->> services
Value not found
Number of nodes visited: 5
Unsucessfull Search path:
Node value: 333
Node value: 539
Node value: 645
Node value: 753
Node value: 868
Total search time is 1.6546e+06 nanoseconds
false
```

Figure 122

As we can see in the figure, the time taken for deletion of the word is **0.00016546s**

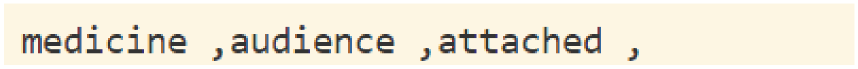
- Small Dataset RBT:
  We inserted the word 'calliber' in the tree for this dataset. Now we will be deleting the word. The node had the word previously in its corresponding vector,



```
medicine ,audience ,attached ,calliber ,
```

Figure 123

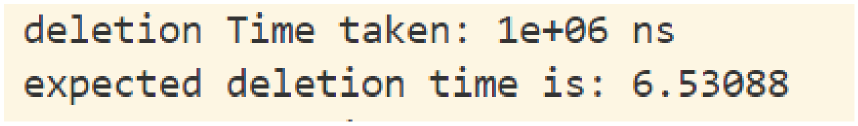
After deleting the word we inserted,



```
medicine ,audience ,attached ,
```

Figure 124

The time taken for the word to be deleted is,



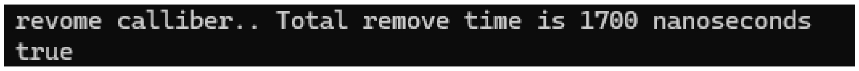
```
deletion Time taken: 1e+06 ns
expected deletion time is: 6.53088
```

Figure 125

As we can see in the figure, the time taken for deletion of the word is **0.0001s** which is less than the expected time **6.5308s**

- Medium Dataset Skiplist:
  We inserted the word 'calliber' in the Skiplist for this dataset. Now we will be deleting the word. The node had the word previously in its corresponding vector,

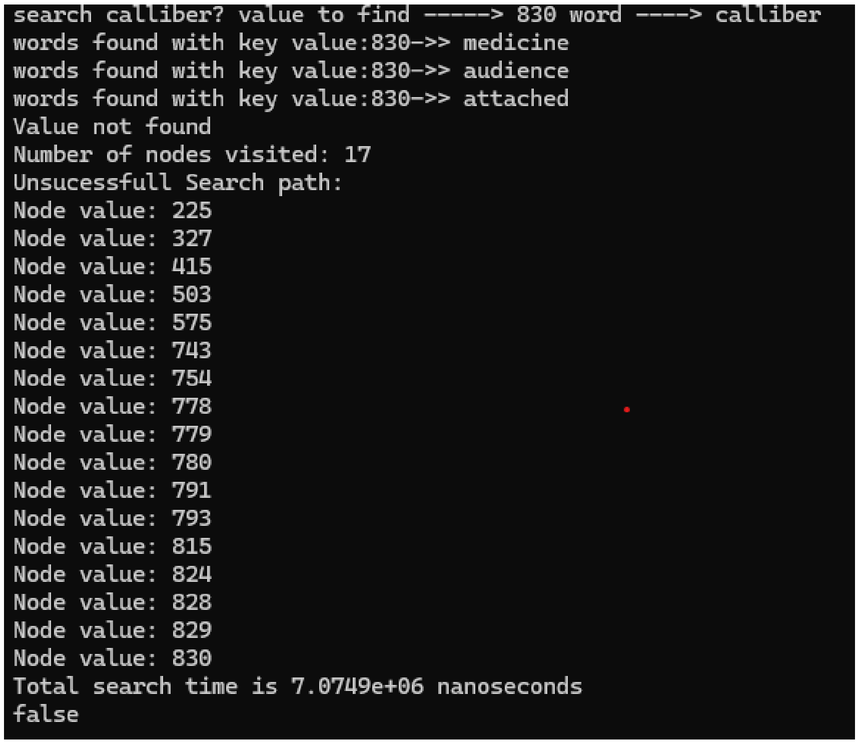
  After deleting the word we inserted,



```
revome calliber.. Total remove time is 1700 nanoseconds
true
```

Figure 126

Now we check if the word is deleted from the skiplist or not, we search the same word in the skip list and see the result, if false then the word was deleted.

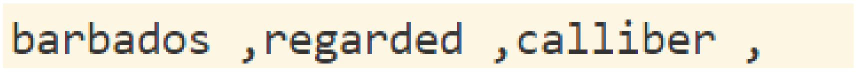


```
search calliber? value to find -----> 830 word ----> calliber
words found with key value:830->> medicine
words found with key value:830->> audience
words found with key value:830->> attached
Value not found
Number of nodes visited: 17
Unsucessfull Search path:
Node value: 225
Node value: 327
Node value: 415
Node value: 503
Node value: 575
Node value: 743
Node value: 754
Node value: 778
Node value: 779
Node value: 780
Node value: 791
Node value: 793
Node value: 815
Node value: 824
Node value: 828
Node value: 829
Node value: 830
Total search time is 7.0749e+06 nanoseconds
false
```

Figure 127

As we can see in the figure, the time taken for deletion of the word is **0.00070749s**

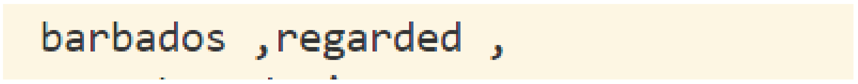
- Medium Dataset RBT:
  We inserted the word 'calliber' in the tree for this dataset. Now we will be deleting the word. The node had the word previously in its corresponding vector,



```
barbados ,regarded ,calliber ,
```

Figure 128

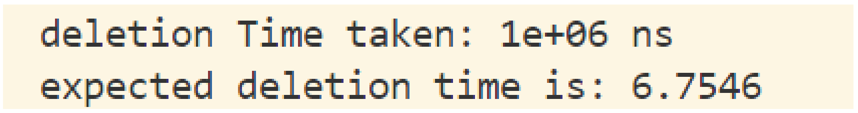
After deleting the word we inserted,



```
barbados ,regarded ,
```

Figure 129

The time taken for the word to be deleted is,

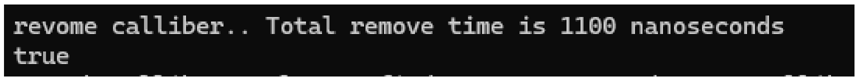


```
deletion Time taken: 1e+06 ns
expected deletion time is: 6.7546
```

Figure 130

As we can see in the figure, the time taken for deletion of the word is **0.0001s** which is less than the expected time **6.7546s**

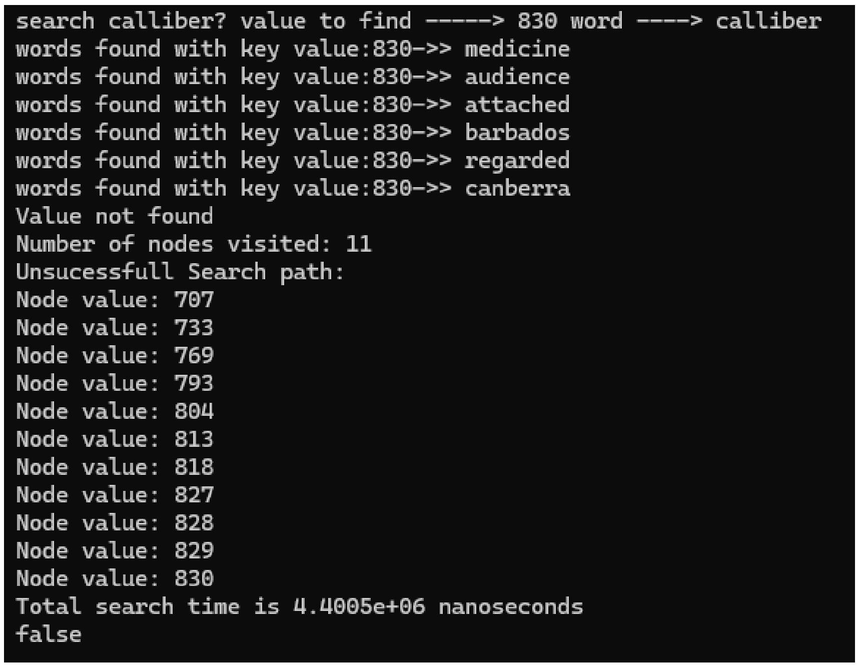
- Large Dataset Skiplist:
  We inserted the word 'calliber' in the Skiplist for this dataset. Now we will be deleting the word. The node had the word previously in its corresponding vector. After deleting the word we inserted,



```
revome calliber.. Total remove time is 1100 nanoseconds
true
```

Figure 131

Now we check if the word is deleted from the skiplist or not, we search the same word in the skip list and see the result, if false then the word was deleted.

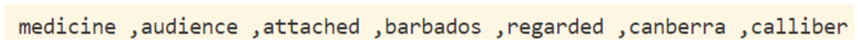


```
search calliber? value to find -----> 830 word ----> calliber
words found with key value:830->> medicine
words found with key value:830->> audience
words found with key value:830->> attached
words found with key value:830->> barbados
words found with key value:830->> regarded
words found with key value:830->> canberra
Value not found
Number of nodes visited: 11
Unsucessfull Search path:
Node value: 707
Node value: 733
Node value: 769
Node value: 793
Node value: 804
Node value: 813
Node value: 818
Node value: 827
Node value: 828
Node value: 829
Node value: 830
Total search time is 4.4005e+06 nanoseconds
false
```

Figure 132

As we can see in the figure, the time taken for deletion of the word is **0.00044005s**

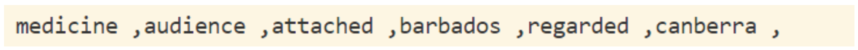
- Large Dataset RBT:
  We inserted the word 'calliber' in the tree for this dataset. Now we will be deleting the word. The node had the word previously in its corresponding vector,



```
medicine ,audience ,attached ,barbados ,regarded ,canberra ,calliber
```

Figure 133

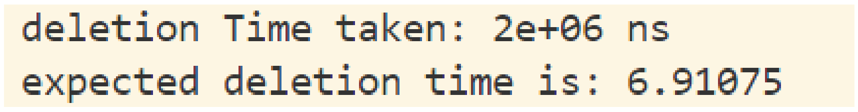
After deleting the word we inserted,



```
medicine ,audience ,attached ,barbados ,regarded ,canberra ,
```

Figure 134

The time taken for the word to be deleted is,



```
deletion Time taken: 2e+06 ns
expected deletion time is: 6.91075
```

Figure 135

As we can see in the figure, the time taken for deletion of the word is **0.0002s** which is less than the expected time **6.9107s**

The table below shows the time-complexity for each functions for this dataset,

Function-Dataset	**Red Black Tree**	**Skip-List**
Insert-Small	**0.00010s**	0.00012s
Insert-Medium	0.0002s	**0.0001s**
Insert-Large	0.0006s	**0.0001s**
Search-Small	**0.0001s**	0.0032s
Search-Medium	**0.0001s**	0.0028s
Search-Large	**0.0002s**	0.0042s
Delete-Small	**0.00010s**	0.00016s
Delete-Medium	**0.00010s**	0.00070s
Delete-Large	**0.00020s**	0.00040s

Table 2: Time-Complexity for Duplicate Dataset

Space Complexity of Red-Black Trees

Space complexity of the RB tree itself is $\mathbf{O(n)}$, where n is the number of nodes, since we need memory for each node, and the space occupied increases with increasing the nodes
Space complexity of the RB tree insertion,deletion and searching is $\mathbf{O(1)}$ as we don't need any additional memory that grows with the size of the trees, we might use a few extra variables and temporary storage, and the amount of extra space required remains constant, so it is 0(1)

One can ask that wont we require space when inserting a new node so how is O(1) possible? The answer is because we do need some extra space to create and store the new node. However, the amount of space required for this new node is constant and doesn't depend on the size of the tree.

For 'n' number of terms, we're concerned with how the space requirements grow as the input size increases. For operations like insertion, deletion, and searching in a Red-Black Tree, the extra space required remains constant regardless of the number of nodes in the tree.

Space Complexity of Skip-List

The Space complexity of a skip list is typically $\mathbf{O(n)}$ for the average case, where n is the number of elements in the list. This is because, on average, each element in a skip list is expected to be part of O(log n) levels due to the probabilistic nature of the structure1.

However, it's important to note that this is the expected space complexity under average circumstances. In the worst-case scenario, where an element could potentially be part of every level, the space complexity could theoretically reach **O(n log n)**. This would happen if the randomization process resulted in every element being promoted to each level up to the maximum level, which is determined by the logarithm of the number of elements.

In practice, the space complexity is kept linear by limiting the height of the skip list, which effectively caps the number of pointers and thus the space usage. So, while the worst-case space complexity can be higher, the expected and typical space usage for a skip list is linear with respect to the number of elements it contains.

Conclusion

To summarize our project, we will go over the advantages and disadvantages of the two data structures.

Red Black Trees

Advantages

Red-Black Trees maintain a balanced height, ensuring that search, insert, and delete operations have a time complexity of O(log n), where n is the number of nodes in the tree. Due to their balanced nature, Red-Black Trees perform well in practice for various operations.

Red-Black Trees offer predictable performance for search, insert, and delete operations, making them suitable for real-time systems and applications.

Disadvantages

Red-Black Trees are more complex to implement compared to simpler data structures like binary search trees.

They may have a higher memory overhead due to the additional color attribute associated with each node.

While Red-Black Trees maintain balance during insertions and deletions, these operations may involve rotations and color changes, which can increase the overhead compared to simpler data structures.

Skiplist

Advantages

Skip Lists are relatively easier to implement compared to complex balanced tree structures like Red-Black Trees.

Skip Lists offer logarithmic time complexity for search, insert, and delete operations on average, similar to balanced trees.

Disadvantages

Skip Lists may consume more memory compared to traditional linked lists due to the additional levels maintained for efficient search.

The performance of Skip Lists may depend on the random choices made during the insertion process, which can lead to variations in performance.

While Red-Black Trees maintain balance during insertions and deletions, these operations may involve rotations and color changes, which can increase the overhead compared to simpler data structures.

References

Introduction to Algorithms, Thomas H. Cormen, Date Accessed: 9th April 2024
Link: Introduction to Algorithms PDF

English-Dicitionary-Database, Benjihilard, Date Accessed: 9th April 2024
Link: English-Dictionary Database

Google-10000-words, No Author, Date Accessed: 9th April 2024
Link: Google-10000 words