APS COURSE PROJECT

Implementation of Persistent Segment tree

Member 1 : **Aditya Gupta** (2019201067)
Member 2 : **Deeksha Sahu** (2019201068)

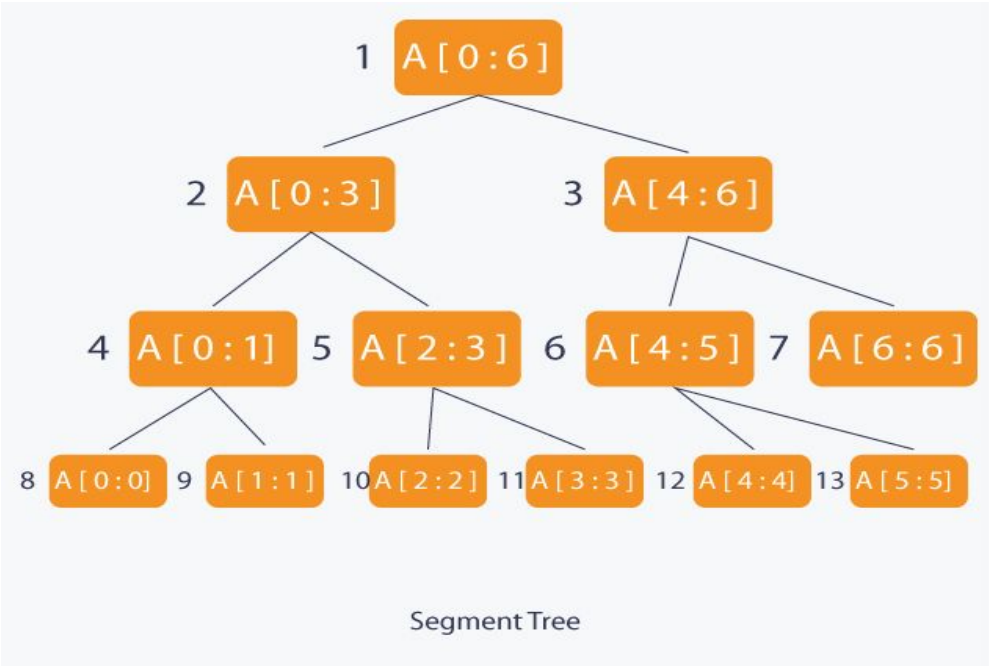
1. Introduction

1.1. Segment Tree Data Structure

The **segment tree** is a highly versatile data structure, based upon the divide-and-conquer paradigm, which can be thought of as a tree of intervals of an underlying array, constructed so that queries on ranges of the array as well as modifications to the array's elements may be efficiently performed.

1.2 Implementation

Since a **Segment Tree** is a binary tree, a simple linear array can be used to represent the Segment Tree. Before building the Segment Tree, one must figure what needs to be stored in the Segment Tree's node?.
For example, if the question is to find the sum of all the elements in an array from indices L to R, then at each node (except leaf nodes) the sum of its children nodes is stored.



Segment Tree

A Segment Tree can be built using **recursion (bottom-up approach)**. Start with the leaves and go up to the root and update the corresponding changes in the nodes that are in the path from leaves to root. Leaves represent a single element. In each step, the data of two children nodes are used to form an internal parent node. Each internal node will represent a union of its children's intervals. Merging may be different for different questions. So, recursion will end up at the root node which will represent the whole array.

1.2.1 Structure to represent a Segment tree Node

```
struct Node
{
   Int data;
   Struct Node* left;
   Struct Node* left;
};
```

1.2.2 Operations on segment tree

1.2.2.1 construction

To ***construct*** a **segment tree** is to initialize it so that it represents some array; we can query and modify this array later but first we need to construct a valid segment tree. We can construct a segment tree either top-down or bottom-up. Top-down construction is recursive; we attempt to fill in the root node, which leads to a recursive call on each of the two children, and so on; the base cases are the leaf nodes, which may be immediately filled in with the corresponding values from the array. After the two recursive calls made on behalf of a non-leaf node return, that node's value is set to the minimum of the values stored at the children.

[CODE]

```
struct node* build_tree(vector<int>& v,int& i,int srt,int end)
{
        if(srt==end)
        {
        // cout<<srt<<" "<<end<<"\n";
        struct node* root=newnode(v[i++]);
        return root;
        }
        int mid=srt+(end-srt)/2;
        struct node* root=newnode(INT_MAX);
        root->left=build_tree(v,i,srt,mid);
        root->right=build_tree(v,i,mid+1,end);
        root->data=min(root->left->data,root->right->data);

        return root;
}
```

1.2.2.1 QUERY

To *query* a segment tree is to use it to determine a function of a range in the underlying array (in this case, the minimum element of that range).When querying a segment tree, therefore, we select a subset of the nodes with the property that the union of their sets of descendants is exactly the interval whose minimum we seek. To do so, we start at the root and recurse over nodes whose corresponding intervals have at least one element in common with the query interval.

[CODE]

```
int range_query(struct node* root,int qs,int qe,int srt,int end)
{

        if(end<qs || srt>qe)
        return INT_MAX;
        else if(srt>=qs && end<=qe)
        return root->data;
        else
        {
        int mid=srt+(end-srt)/2;
        return min(range_query(root->left,qs,qe,srt,mid),
        range_query(root->right,qs,qe,mid+1,end));

        }
}
```

Time Complexity analysis

Construction

Construction requires a constant number of operations on each node of the segment tree; as there are $\mathcal{O}(N)$ nodes, the construction takes linear time. You will notice that this is faster than doing N separate update operations.

Update

A constant number of operations is performed for each node on the path from the root to the leaf associated with the modified element. The number of nodes on this path is bounded by the height of the tree; hence, *per* the conclusions in the Space section above, the time required for an update is $\mathcal{O}(\lg N)$.

Query

Consider the set of selected nodes (marked in yellow in the illustration). It is possible, in the case of a query $f(1, N-1)$, that there are logarithmically many. Can there be more? The answer is no. The simplest way of proving this, perhaps, is by exhibiting an algorithm for selecting the nodes that terminates within $\mathcal{O}(\lg N)$ steps; this is the non-recursive algorithm

alluded to earlier. So a query takes $\mathcal{O}(\lg N)$ time. The proof that the recursive version also takes $\mathcal{O}(\lg N)$ time is left as an exercise for the reader.

1.3. Persistent Data Structure

A persistent data structure is a data structure that always preserves the previous version of itself when it is modified. They can be considered as 'immutable' as updates are not in-place.
There are several classes of Persistence which are explained below :

- **Partial Persistence**

  In this persistence model we may **query any previous version** of the data structure, but we may **only update the latest** version. We have operations read(var, version) and newversion = write(var, val). This definition implies a linear ordering on the versions like patial persistence in version diagrams.

- **Full Persistence**

  In this model, **both updates and queries are allowed on any version** of the data structure. We have operations read(var, version) and newversion = write(var, version, val). The versions form a branching tree as in full persistence in version diagrams.

- **Confluent Persistence**

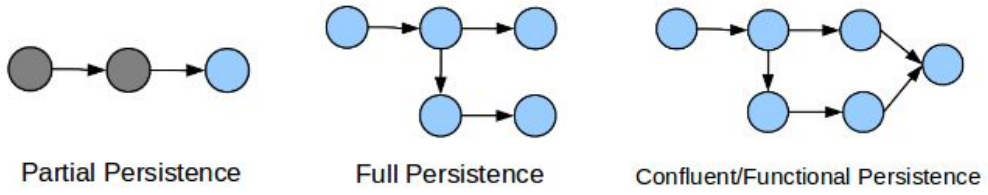
  In this model, **in addition to the previous operation, we allow combination operations to combine input of more than one previous versions to output a new single version**. We have operations read(var, version), newversion = write(var, version, val) and newversion = combine(var, val, version1, version2). Rather than a branching tree, combinations of versions induce a DAG (direct acyclic graph) structure on the version graph, shown in confluent persistence in version diagrams.

- **Functional Persistence**
  This model takes its name from functional programming where objects are immutable. The nodes in this model are likewise immutable: revisions do not alter the existing nodes in the data structure but create new ones instead. Okasaki discusses these as well as other functional data structures in his book @okasaki.

The difference between functional persistence and the rest is we have to keep all the structures related to previous versions intact: the only allowed internal operation is to add new nodes. In

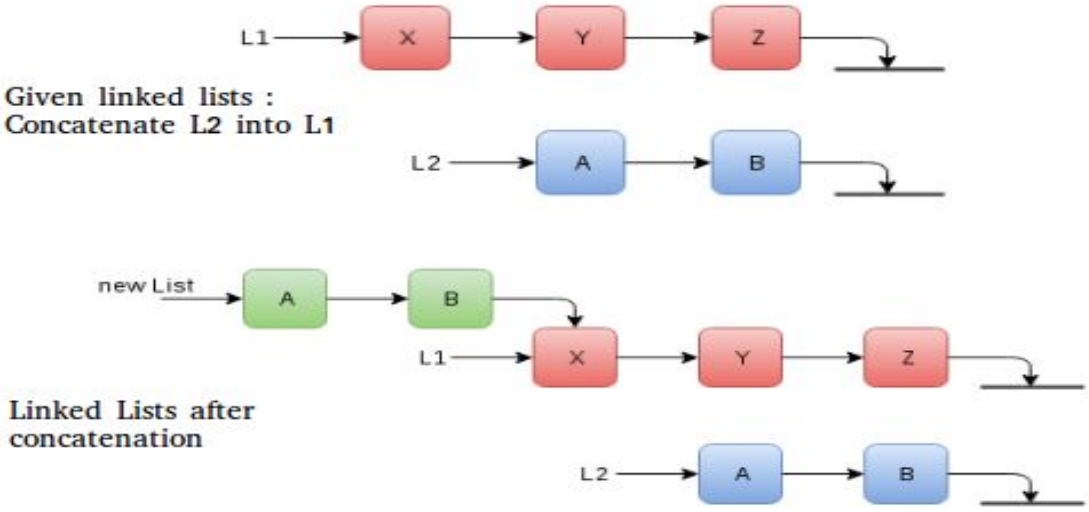
the previous three cases we were allowed anything as long as we were able to implement the interface.

Version Diagrams Following diagrams represent above classes of persistence. Grey node indicates read-only version and Blue node indicates read-write version.



1.3.1. Examples of Persistent Data Structure

- **Linked List Concatenation**
  Consider the problem of concatenating two singly linked lists with n and m as the number of nodes in them. Say $n>m$. We need to keep the versions, i.e., we should be able to original list. One way is to make a copy of every node and do the connections. $O(n+m)$ for traversal of lists and $O(1)$ each for adding $(n+m-1)$ connections. Other way, and a more efficient way in time and space, involves traversal of only one of the two lists and fewer new connections. Since we assume $m<n$, we can pick list with m nodes to be copied. This means $O(m)$ for traversal and $O(1)$ for each one of the m connections. We must copy it otherwise the original form of list wouldn't have persisted.



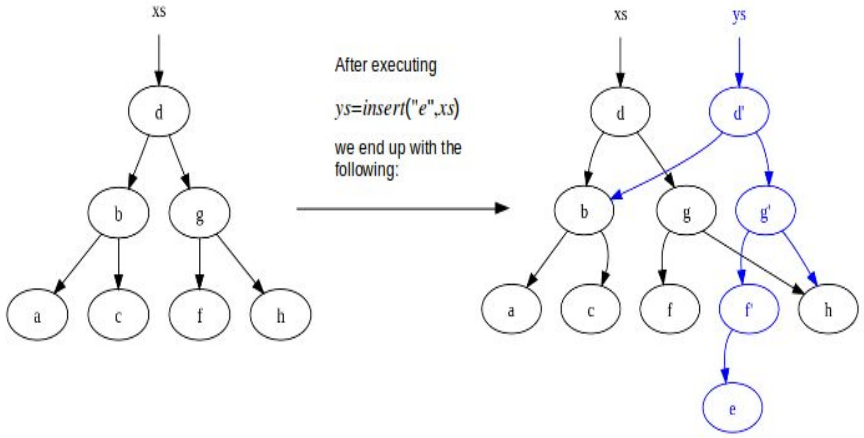
- **Binary Search Tree Insertion**
  Consider the problem of insertion of a new node in a binary search tree. Being a binary

search tree, there is a specific location where the new node will be placed. All the nodes in the path from the new node to the root of the BST will observe a change in structure (cascading). For example, the node for which the new node is the child will now have a new pointer. This change in structure induces change in the complete path up to the root.

This example is taken from Okasaki.

xs=[a,b,c,d,f,g,h]

might be represented by the following binary search tree:



1.3.2. Techniques to implement Persistent Data Structures

For the methods suggested below, updates and access time and space of versions vary with whether we are implementing full or partial persistence.

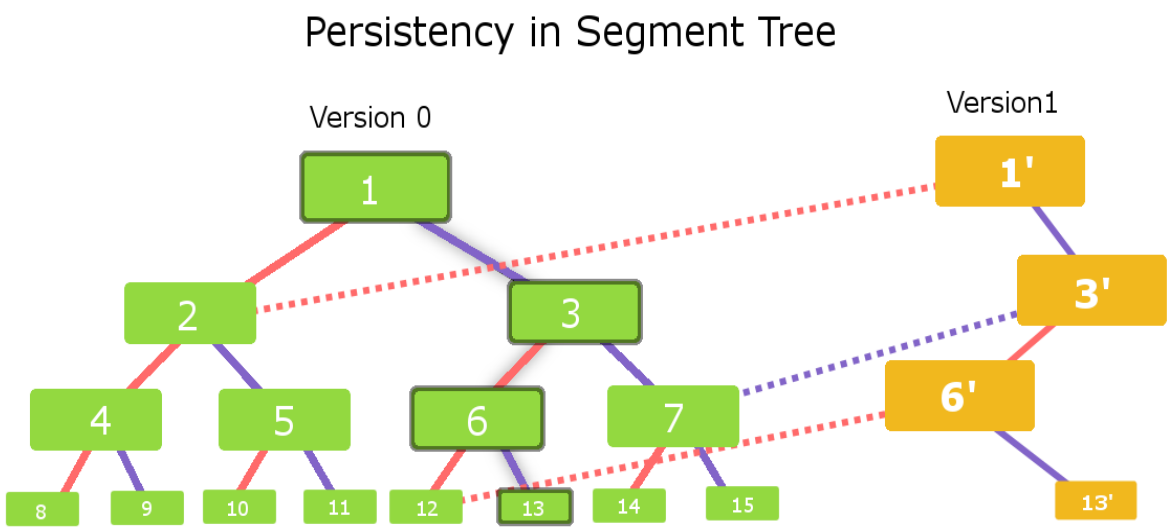
- **Path copying:**
  Make a copy of the node we are about to update. Then proceed for update. And finally, cascade the change back through the data structure, something very similar to what we did in example two above. This causes a chain of updates, until you reach a node no other node points to — the root. How to access the state at time t? Maintain an array of roots indexed by timestamp.
- **Fat nodes:**
  As the name suggests, we make every node store its modification history, thereby making it 'fat'.
- **Nodes with boxes:**
  Amortized $O(1)$

time and space can be achieved for access and updates. This method was given by Sleator, Tarjan and their team. For a tree, it involves using a modification box that can hold:

- one modification to the node (the modification could be of one of the pointers, or to the node's key or to some other node-specific data)
- the time when the mod was applied

2. Persistent Segment Tree

We implemented partial persistent Segment Tree that stores data set. By partial persistent we mean that insertion is only allowed in the latest version whereas we can query any previous version.In this we are creating new nodes for the nodes that are affected due to change in any node in the path from leaf to root.



1.2.2.1 Upgrade

To *update* a segment tree is to modify the value of one element in the underlying array. To do so, we first modify the corresponding leaf node. The other leaf nodes are not affected, since each leaf node is associated only with an individual element. The modified node's parent is affected, since its associated interval contains the modified element, and so is the grandparent, and so on up to the root, but no other nodes are affected.

[CODE]

```
struct node* upgrade(struct node* v1,struct node* v2,int index,int val,int srt,int end)
{
        struct node* prev=v1;
        struct node* temp=newnode(INT_MAX);
        v2=temp;
        int mid=srt+(end-srt)/2;
        if(srt==end)
        {
        temp->data=val;
```

```
        return temp;
        }
        if(index<=mid)
        {
        temp->right=prev->right;
        prev=prev->left;
        temp->left=upgrade(prev,temp,index,val,srt,mid);
        temp->data=min(temp->right->data,temp->left->data);
        }
        else
        {
        temp->left=prev->left;
        prev=prev->right;
        temp->right=upgrade(prev,temp,index,val,mid+1,end);
        temp->data=min(temp->right->data,temp->left->data);
        }
        return temp;

}
```

2.1 Analysis of Space and Time

Time complexity

Since only log N number of nodes are affected for each query, therefore time complexity for each update query is log N

Also **space complexity** is Log N as only log N nodes are to be created newly.

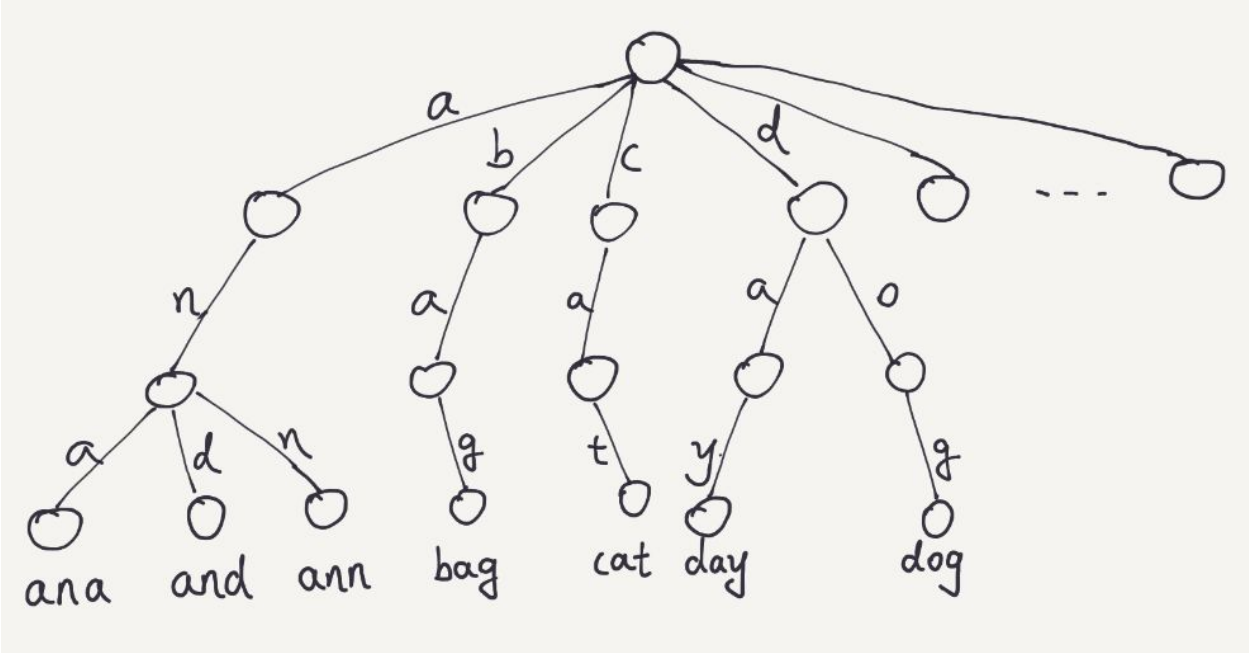
Thus for n update queries both space and time complexity becomes $O(N \log_2 N)$

Implementation of Persistent Binary Trie

1. Introduction

1.1.Trie Data Structure

A **Trie** is a special data structure used to store strings that can be visualized like a graph. It consists of nodes and edges. Each node consists of at max 26 children and edges connect each parent node to its children. These 26 pointers are nothing but pointers for each of the 26 letters of the English alphabet A separate edge is maintained for every edge.



1.1. Binary Trie Data Structure

Every node of a binary trie consists of **two branches, one for 0 and other for 1**. A Trie node field **Flag** is used to distinguish the node as end of word node.

1.2.1 Structure to represent a Binary Trie Node

```
class Trie
{
        public:
       Trie *left;
       Trie *right;
       bool flag;
}
```

1.2.2 Operations on Binary Trie

1.2.2.1 Insert

Every character of the input key is inserted as an individual Trie node. Note that the *children* is an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array *children*. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of the key, and mark end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word. The key length determines Trie depth.

[CODE]

```
Trie * insert(Trie*root,string x)
{
        Trie* node=root;
        if(node==NULL)
        node=init();
        root=node;
        for(long long int i=0;i<x.length();i++)
        {

        if(x[i]==0)
        {
        if(node->left==NULL)
        node->left=init();
        node=node->left;
        }
        else
        {
        if(node->right==NULL)
        node->right=init();
        node=node->right;
        }
        }
        return root;
}
```

1.2.2.2 Find

[CODE]

```
int find(Trie*root,string s)
{
        for(int i=0;i<s.length();i++)
        {
        if(s[i]==0)
        {
        if(root!=NULL)
        root=root->left;
        else
        return 0;
        }
        else
        {
        if(root!=NULL)
        root=root->right;
        else
        return 0;
        }
        }
        return 1;
}
```

Time Complexity analysis

Insert and search costs **O(key_length)**,

however the memory requirements of Trie is

O(ALPHABET_SIZE * key_length * N)

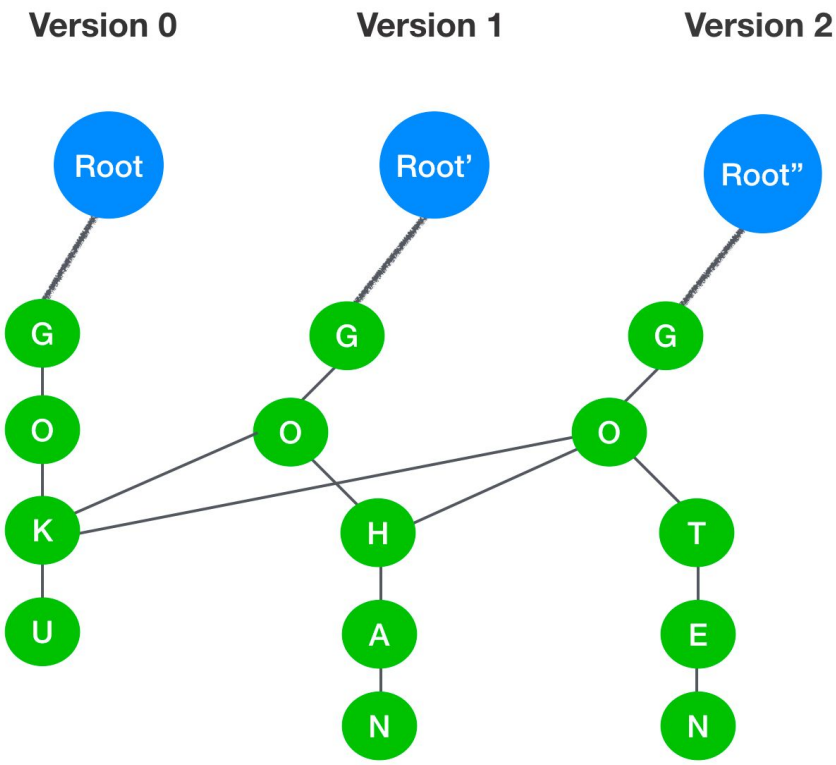
where N is the number of keys in Trie

Space Complexity analysis

Space complexity is determined by the number of nodes present in the trie or sum of length of each word stored in the trie.

2. Persistent Binary Trie

Our aim is to apply persistency in Trie and also to ensure that it does not take more than the standard trie searching i.e. **O(length_of_key)**. We will also analyze the extra space complexity that persistency causes over the standard Space Complexity of a Trie.



Consider this as a sample that shows how persistency is implemented in string based trie, this can be extended for binary trie also.

2.1 Upgrade

This code shows how persistency is implemented in Binary Trie.

[CODE]

```
Trie * upgrade(Trie *root1,Trie *root2,string s)
{
        if(root2==NULL)
        root2=init();
        Trie *node=root2;
        for(int i=0;i<s.length();i++)
        {
        if(s[i]==0)
        {
        if(root1!=NULL)
```

```
        {
                root2->right=root1->right;
                root1=root1->right;
        }
        root2->left=init();
        root2=root2->left;

        }
        else
        {
        if(root1!=NULL)
        {
                root2->left=root1->left;
                root1=root1->left;
        }
        root2->right=init();
        root2=root2->right;
        }
        }
return node;

}
```

2.2 Space Complexity Analysis of Persistent binary trie

Obviously, persistency in data structures comes with a trade of space and we will be consuming more memory in maintaining the different versions of the trie.

Now, let us visualize the worst case – for insertion, we are creating **O(length_of_key)** nodes and each newly created node will take a space of **O(sz)** to store its children.

Hence, the space complexity is **O(length_of_key * sz).**

2.3 Time Complexity Analysis of Persistent binary trie

we will be visiting all the **X**(length of key) number of nodes in the trie while inserting; So, we will be visiting the X number of states and at each state we will be doing **O(sz)** amount of work by liking the sz children of the previous version with the current version for the newly created trie nodes.

Hence, Time Complexity of insertion becomes **O(length_of_key * sz)**.

But the searching time is still linear over the length of the key to be searched .

Hence, the time complexity of searching a key is still **O(length_of_key)** just like a

standard trie.

3. Code

The code of this project can be downloaded from this GitHub link.
https://github.com/Aditya-crypto/APS-Project.git

4. References

- **Persistent Data Structures (Video Lecture by Eric Demaine) :**

  https://www.youtube.com/watch?v=DQY3CQK4Ar0&t=2036s

- **https://www.geeksforgeeks.org/persistent-data-structures/**
- **http://wcipeg.com/wiki/Segment_tree**
- **https://www.geeksforgeeks.org/persistent-trie-set-1-introduction/**