Digital Juke box using Splay Trees:

Key words
Splay tree,splay,insert,zig and zag,rank

I Abstract—>

Data structures is a popular method used for organising data, where it deals with searching for the best ways of the storing and managing it, data structures also involve how can we store a cluster of data without any loss ,the algorithms for the various operations involved in this data storage and management , and of how efficient they are in saving memory space and time.Music is an important part of our life , it is a form of art , it relaxes us it is a form ofcommunication ,Many people struggle with communicating with others but can say how they feel with a song that they write or have heard, making it easier to express themselves without having fear or intimidation. In this paper we will discuss about how people can listen to their favourite music where they go ,whether it is a bar , a restaurant or a mall , the project will talk about the creating a “Digital Jukebox” by implementing splay trees and stacks using C language . It deals with how customers can listen to there favourite music and how using splay trees allows room for more flexibility .The project also explains why the splay trees are being considered , the basic operations ,advantages /disadvantages and a working code/model .

I. Introduction—>

Data structures are used in almost all applications and enhance the accuracy of operations. There are two main function of using data structures that is putting data in and removing data from the application incorporating it. The most common data structures are linked list, arrays, queues, stacks, trees and graphs. It is very helpful in managing memory better in applications using data structures. The splay tree is a type of binary search tree. Unlike other variants like the AVL tree, the red-black tree, or the scapegoat tree, the splay tree is not always balanced. Instead, it is optimized so that elements that have been recently acessed are quick to access again. A Digital Jukebox is a software for managing music files. Also called a "digital music manager," "music manager," or just "jukebox," it lets users organize MP3 and other audio files into playlists and play the songs ,in this juke box the splay tree data structure will be used.The splay property makes the splay tree property is similar in nature to a stack. A stack has the Last-In-First-Out (LIFO) property, so the most recent item added is the quickest to access. Splay trees are similar in that when you add a new item, it becomes the root of the tree no matter what. But they take it a step further. Even when an item is simply searched for, it becomes the new root of the tree.Splay trees are used instead of stacks because it is the optimal choice for the current application that is being considered .The splay trees make it easier to search and makes the access to recently searched node faster .The digital juke box requires us
it easier for the user to listen to their favourite songs ,this can be done by using the splay tree which makes the most frequently accessed song as the root .This will further help in dividing songs into categories like popular / most listened to etc. In case of an AVL tree ,the shape of the tree is constrained at all times such that it is balanced ,meaning that the height of tree never exceeds O(log n).But in case of splay trees the shape is not constrained and varies based on what searches are performed .Another advantage that splay trees have on the AVL trees is that they are on an average faster.The main idea of splay tree is to bring the recently accessed item to root of the tree, this makes the recently searched item to be accessible in O(1) time if accessed again. In a situation where there exists millions of keys ,but only a few are frequently accessed ,then the splay trees can be considered an ideal choice as all splay tree operations run in O(logn) time on an average,n being the number of entries.The splay tree’s main advantage is that it keeps the most recently accessed nodes at the top of the tree, decreasing time for subsequent searches. .Splay trees also have low memory overhead, which makes them attractive for memory-sensitive programs. The biggest disadvantage of splay trees is that they can be linear. This is extremely bad for performance, though it is also extremely rare for this case to occur.

Basic Operations of a splay tree—>

A. Splaying

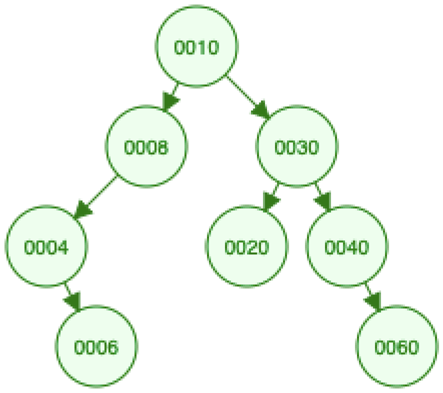
Splaying is what keeps the splay tree roughly balanced. Inorder to splay a node in the tree ,the splaying steps are repeatedly performed on it until it rises to the top and becomes the root. The type of splaying step performed ,will depend on the following 3 possibilities

1) The node’s parent is the root
2) The node is the left child of a right child (or the right child of a left child)
3) The node is the left child of a left child (or the right child of a right child)

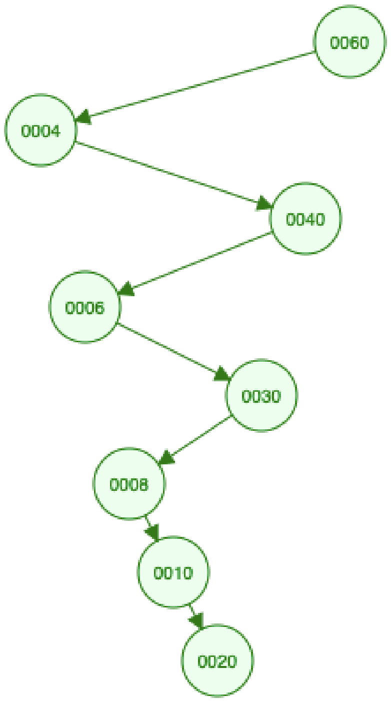
If the node’s parent is the root, then only one rotation is needed to make it the root. That is if the node is the left child of the root, we perform a right rotation, and if the node is the right child of the root, perform a left rotation.

the left rotation or left then right rotation for if N is a node, P is the parent and G is the grandparent ,it rotates N and P right and then N and G left .It does the opposite if the node rigth child of a left child, it does the opposite .

If the node is the left child of a left child, there are also two rotations. First, G and P are rotated right. Then X and P are rotated right. If the node is the right child of a right child, G and P are first rotated left followed by X and P being rotated left.



After Splaying:



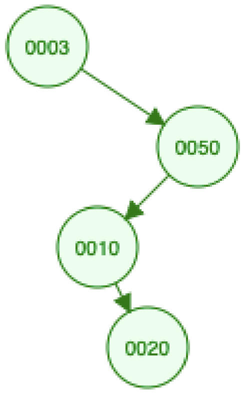
If the node is the left child of a right child,2 rotations are needed to be performed that is either first right rotation and

II. Search

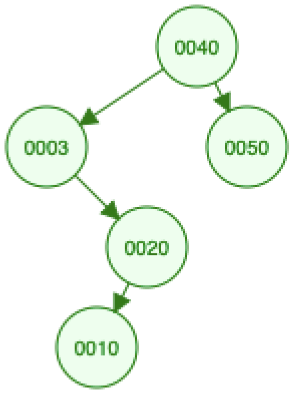
Search is simple once splay is implemented. To search for a node, use binary search down the tree to locate the node. then perform the splay ,and bring this recently accessed node to the root.

III. Insertion

To insert a node, find its appropriate location at the bottom of the tree using binary search. Then perform splay on that node.



After inserting 40:

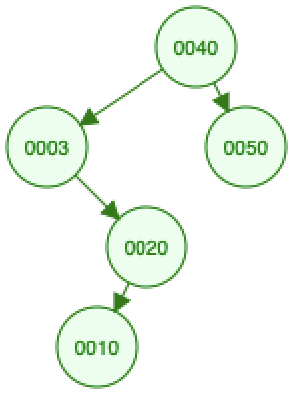


IV. Deletion

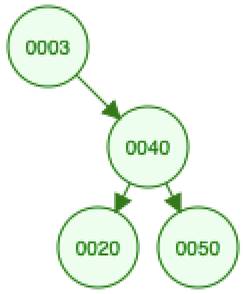
Deletion is the only operation that has some wiggle room for the implementer. After all, there's no obvious node to splay when you're removing a node.

A typical way is that the node to be deleted is first splayed, making it a root node. Then it is deleted.

Regardless, a typical implementation will eventually splay the parent of the deleted node.



After deletion of 10:



SPLAY TREE TIME COMPLEXITY ANALYSIS:

A simple amortized analysis of static splay trees can be carried out using the potential method. Define:

size(r) = the number of nodes in the sub-tree rooted at node r (including r). rank(r) = log2(size(r)).

Φ = the sum of the ranks of all the nodes in the tree.

The value of Φ will be more for a poorly balanced tree than that for a better /well balanced tree

To use the potential method , calculate $\Delta\Phi$: the change in the potential caused by a splay operation. Each case is checked separately. Denote by r^j the rank function after the operation.

$r^j(g) + r^j(x) \quad 2$

[since $r(x)<r(p)$ and $r(x)>r(p)$] $r(x)$

[due to the concavity of the

$\leq 3(r^j(x)-r(x)) - 2$ log function]

Zig-Zag step:

$\Delta\Phi = r^j(g) - r(f) + r^j(p) - r(p) + r^j(x) - r(x)$

$\leq r^j(g) + r^j(p) - 2$

[since $r^j(x)=r(g)$ and $r(x)<r(p)$] $r(x)$

[due to the concavity of the log

$\leq 3(r^j(x)-r(x)) - 2$ function]

The amortized cost of any operation is $\Delta\Phi$ which is added to the actual cost. It can be seen that since the zig-zig or zigzag operation make 2 rotations ,they have an actual cost of 2. Hence amortized cost of the operations

amortized-cost= cost + $\Delta\Phi$

$\leq 3(r^j(x) - r(x))$

When summed over the entire splay operation, this telesco es to 3(rank(root) rank(x)) which is O(log n). The Zig opera- tion adds an amortized cost of 1, but at max there is only one such operation .

So the total *amortized* time for a sequence of m operations is:

$$T_{\text{amortized}}(m) = O(m \log n)$$

To go from the amortized time to the actual time, the decrease in potential is added from the initial state before any operation is done (Φi) to the final state after all operations have been completed (Φf).

$$\Phi_i - \Phi_f = \sum_x \text{rank}_i(x) - \text{rank}_f(x) = O(n \log n)$$

where the last inequality comes from the fact that for every node x, the minimum rank is 0 and the maximum rank is log(n).

Now after bounding the actual time:

Tactual(m)=O(mlogn+nlogn)

Weighted Analysis of Splay tree:

The analysis presented above can be generalized in the following way.

Each node *r is assigned* a weight $w(r)$.

size(r) = the sum of weights of nodes in the sub-tree rooted at node r (including r) is defined . The rank(r) and Φ are defined exactly as above.

The same analysis applies and the amortized cost of a splaying operation is again:

$$\text{rank}(root) - \text{rank}(x) = O(\log W - \log w(x)) = O\left(\log \frac{W}{w(x)}\right)$$

where W is the sum of all weights.

The decrease from the initial to the final potential is bounded by:

$$\Phi_i - \Phi_f \leq \sum_{x \in tree} \log \frac{W}{w(x)}$$

since the maximum size of any single node is W and the minimum is $w(x)$.

Hence the actual time is bounded by:

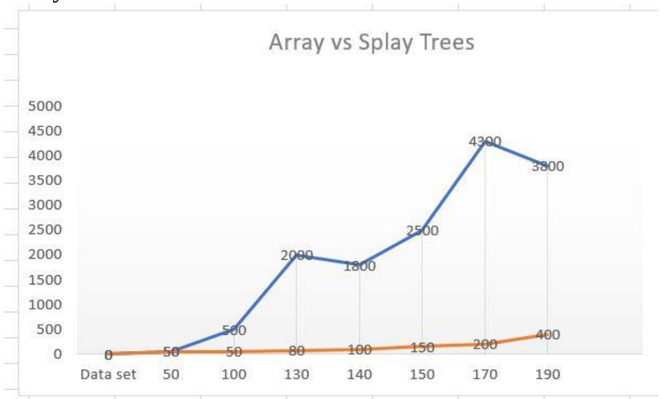
$$O\left(\sum_{x \in sequence} \log \frac{W}{w(x)} + \sum_{x \in tree} \log \frac{W}{w(x)}\right)$$

Reference: https://en.wikipedia.org/wiki/Splay_tree

Graphical Explanation:

We can say that using splay trees is more time saving and reduces time complexity when compared to arrays which are used for creating stacks and queues which increase processing time thus causing function to load in more time and also creating a bad user analysis

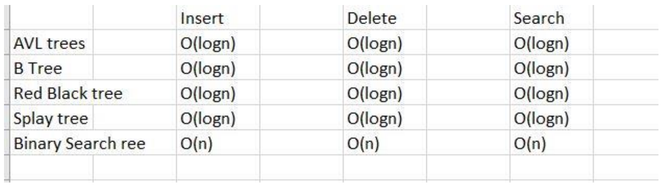
The following graphs show comparative analysis of splay vs arrays:



Tabular Explanation:

The following table shows comparison between different data structures on basis of time complexity:

The following table shows comparison of different search trees on basis of time complexity.



	Insert	Delete	Search
AVL trees	O(logn)	O(logn)	O(logn)
B Tree	O(logn)	O(logn)	O(logn)
Red Black tree	O(logn)	O(logn)	O(logn)
Splay tree	O(logn)	O(logn)	O(logn)
Binary Search ree	O(n)	O(n)	O(n)

Method to be Implemented:

1) Owner inputs the song which he wants to add to the jukebox.
2) The jukebox inserts the song into the playlist.
3) User using the jukebox searches for the song which he wants to listen.
4) Splay operation is carried to find the song efficiently.
5) The output is displayed to the screen.
6) The user needs to pay the price to listen to that song.
7) The user can enjoy his song.
8) As soon as song gets over program ends.

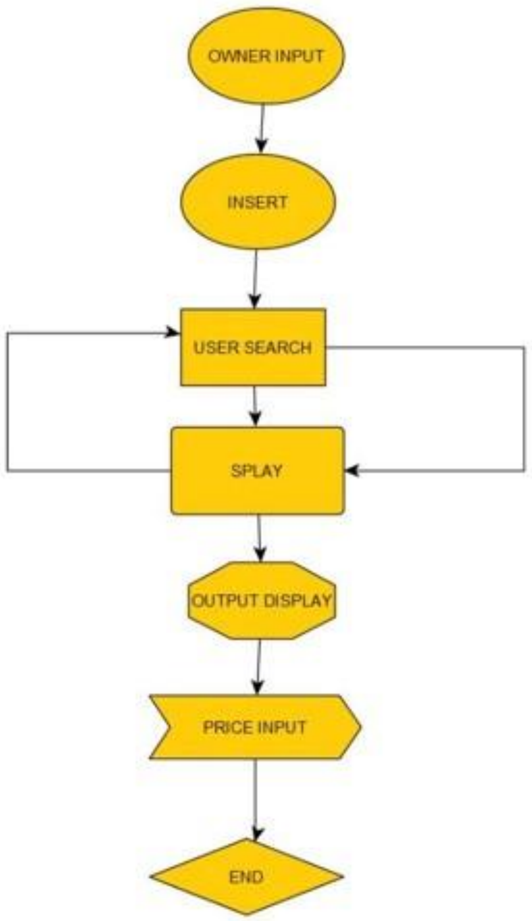


fig-Flow chart for working

Reference: https://www.geeksforgeeks.org/avl-tree-set-1-insertion

Code Implemented:

note:the following code has been implemented in dev c++

For creating a node:

```
#include<stdio.h>
#include<stdlib.h>
//An AVL tree node
struct node
{
int song;
struct node *left, *right;
};
/* Helper function that allocates a new node with the given
key and
NULL left and right pointers. */
struct node* newNode(int song)
{
struct node* node = (struct node*)malloc(sizeof(struct
node));
node->song       = song;
node->left = node->right = NULL; return (node);
}
//A utility function to right rotate subtre e rooted with y
//See the diagram given above.
struct node *rightRotate(struct node *x)
{
struct node *y = x->left; x->left = y->right; y->right = x;
return y;
}
```

```
//A utility function to left rotate subtree rooted with x
//See the diagram given above.
struct node *leftRotate(struct node *x)
{
struct node *y = x->right; x->right = y->left; y->left = x;
return y;
}
```

For splay:

It is done for searching the song by using splay tree search

```
//This function brings the key at root if key is present in
tree.
//If key is not present, then it brings the last accessed item
at
//root. This function modifies the tree and returns the new
root
struct node *splay(struct node *root, int song)
{
// Base cases: root is NULL or key is present at root
if (root == NULL || root->song == song) return root;
// Key lies in left subtree
if (root->song > song)//the following process operates
through the recursive loops
{
//Key is not in tree, we are done if (root->left == NULL)
return root;
//Zig-Zig (Left Left)
if (root->left->song > song)
{
// First recursively bring the key as root of left-left
root->left->left = splay(root->left->left, song);
// Do first rotation for root, second rotation is done after
else
root = rightRotate(root);
}
else if (root->left->song < song) // Zig-Zag (Left Right)
{
// First recursively bring the key as root of left-right
root->left->right = splay(root->left->right, song);
// Do first rotation for root->left if (root->left->right !=
NULL)
root->left = leftRotate(root->left);
}
// Do second rotation for root
return (root->left == NULL)? root:rightRotate(root);
}
else // Key lies in right subtree
{
//Key is not in tree, we are done if (root->right == NULL)
return root;
//Zag-Zig (Right Left)
if (root->right->song > song)
{
// Bring the key as root of right-left
root->right->left = splay(root->right->left, song);
// Do first rotation for root->right if (root->right->left !=
NULL)
root->right = rightRotate(root->right);
}
else if (root->right->song < song)// Zag-Zag (Right Right)
{
// Bring the key as root of right-right and do first rotation
root->right->right = splay(root->right->right, song);
//the use. of right right twice?
root = leftRotate(root);
}
// Do second rotation for root
return (root->right == NULL)? root:leftRotate(root);
}
}
//The search function for Splay tree. Note that this function
//returns the new root of Splay Tree. If key is present in tree
//then, it is moved to root.
struct node *search(struct node *root, int song)
{
return splay(root, song);
}
```

explanation—>

To perform a splay operation , there is need to carry out sequence of Splay steps, which moves the node closer to the root. The recently accessed nodes are kept closer to the root so that tree remains balanced.

Each Splay step depends on three factors

X is left or right child of its parent node P. Check P is root node or not, if not. P is left or right child of its parent G.

Results of Splaying

The result is a binary tree, with the left subtree having all keys less than the root, and the right subtree having keys greater than the root.

The resulted tree is more balanced than the original tree.

If an operation near the root is done, the tree can become less balanced.

For Insertion:

For inserting the song inside the tree

```
// Function to insert a new key k in splay tree with given
root
struct node *insert(struct node *root, int k)
{
// Simple Case: If tree is empty
if (root == NULL) return newNode(k);
//Bring the closest leaf node to root root = splay(root, k);
//If key is already present, then return
if (root->song == k) return root;
//Otherwise allocate memory for newnode
struct node *newnode  = newNode(k);
//If root's key is greater, make root as right child
//of newnode and copy the left child of root to newnode
if (root->song > k)
{
newnode->right = root; newnode->left = root->left; root-
>left = NULL;
}
//If root's key is smaller, make root as left child
//of newnode and copy the right child of root to newnode
else
{
newnode->left = root; newnode->right = root->right; root-
>right = NULL;
}
return newnode; // newnode becomes new
}
```

explanation—>

To insert a node in to the tree

1)Insert a node normally in to the tree.

2)Splay the newly inserted node to the top of the tree

For Deleting:

For removing some song that has been outdated the following code can be used which would first involve the search operation and then will remove the node.

```
struct node* delete_key(struct node *root, int song)
{
struct node *temp;
if (!root)
return NULL;
//Splay the given key root = splay(root, key);
//If key is not present, then
//return root
if (song != root->song) return root;
//If key is present
//If left child of root does notexist
//make root->right as root
if (!root->left)
{
temp = root;
root = root->right;
}
//Else if left child exits
else
{
temp = root;
/*Note: Since key == root->key,
so after Splay(key, root- >lchild),the tree we get wIll have
no right child tree4
and maximum node in left subtree will get splayed*/
// New root
root = splay(root->left, song);
//Make right child of previous root as
//new root's right child root->right = temp->right;
}
//free the previous root node, that is,
//the node containing the key
free(temp);
//return root of the new Splay Tree
return(root);
}
```

explanation-->

1)Access the node to be deleted bringing it to the root.

2) Delete the root leaving two subtrees L left and R right.

3)Find the largest element in L, thus the root of L will have no right child.

4)Make R the right child of L's root

VI CONCLUSION —>

We have developed a model Digital jukebox , that uses splay trees unlike the conventional mechanical jukebox (which has limited practical applications)and the digital jukeboxes that use stack and queue data structure .We have also explained its implementation in c language .The work suggested why splay trees are considered the optimal solution for the implementation of the digital juke box ,and not any other data structure.The splay trees implementation of juke box on execution advocated that the given model is accurate and functional.This project also demonstrated that, by using the stack data struc- ture transactions could be allowed, which would further aide it in practical applications (in a restaurant etc).Thus with the help of splay trees we have achieved a more accurate and flexible digital jukebox.'

ACKNOWLEDEGMENT---->

We truly thank our motivator and guide Prof. Santhi K.who provided us with the insight and expertise that greatly assisted the research although they may not agree with all of the interpretaions/conclusions of this paper.

VII REFERENCES----->

[1] https://www.geeksforgeeks.org/avl-tree-set-1-insertion
[2]https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm
[3]https://www.cs.usfca.edu/~galles/visualization/SplayTree.html
[4]https://www.cs.usfca.edu/~galles/visualization/AVLtree.html
[5]**https://stackoverflow.com/questions/7467079/differencbetween-avl-trees-and-splay-trees**
[6] B. Allen and J. I. Munro, Self-organizing binary search trees, J. ACM, 25 (1978), pp. 526– 535.
[7] M. Badoiu, R. Cole, E. D. Demaine, and J. Iacono, A unified access bound on comparisonbased dynamic dictionaries, Theor. Comput. Sci., 382 (2007), pp. 86–96.
[8] A. Blum, S. Chawla, and A. Kalai, Static optimality and dynamic search-optimality in lists and trees, Algorithmica, 36 (2003), pp. 249–260.
[9] P. Chalermsook, M. Goswami, L. Kozma, K. Mehlhorn, and T. Saranurak, Self-adjusting binary search trees: What makes them tick?, in Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings, vol. 9294 of LNCS, Springer, 2015, pp. 300–312.
[10] R. Cole, On the dynamic finger conjecture for splay trees. part II: The proof, SIAM Journal on Computing, 30 (2000), pp. 44–85.
[11] R. Cole, B. Mishra, J. Schmidt, and A. Siegel, On the dynamic finger conjecture for splay trees. part I: Splay sorting log n-block sequences, SIAM Journal on Computing, 30 (2000), pp. 1–43.
[12] E. D. Demaine, D. Harmon, J. Iacono, D. Kane, and M. Pˇatra¸scu, The geometry of binary search trees, in Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2009), New York, New York, January 4–6 2009, pp. 496–505.
[13] E. D. Demaine, D. Harmon, J. Iacono, and M. Pˇatra¸scu, Dynamic optimality—almost, SIAM Journal on Computing, 37 (2007), pp. 240–251. Special issue of selected papers from the 45th Annual IEEE Symposium on Foundations of Computer Science.
[14] D. E. Knuth, The Art of Computer Progr amming, Volume 3: (2Nd Ed.) Sorting and Se arching, Addison Wesley Longm an Publishing Co., Inc., Redwood City, C A, US A, 1998.
[15] J. M. Luc as, Canonical forms for competitive binary se arch tree algorithms, Rutgers University, Department of Computer Science, L abor atory for Computer Science Research, 1988. [16]J. M. Lucas, On the competitiveness of splay trees: Relations to the union-find problem, On-line Algorithms, 7 (1992), pp. 95–124.
[17] K. Mehlhorn, Nearly optimal binary search trees, Act a Informatica, 5 (1975), pp. 287–295.
[18] J. I. Munro, On the competitiveness of line search, in Algorithms-ES A 2000, Springer, 2000, pp. 338–345 .
[19] S. Pettie, Splay trees, d avenport-schinzel sequences, and the deque conjecture, CoRR, abs/0707.2160 (2007).
[20] D. D. Sleator and R. E. T arjan, Self- adjusting binary search trees, J. ACM, 32 (1985), pp. 652–686.
[21] A. Subramanian, An explanation of splaying, J. Algorithms, 20 (1996), pp. 512–525. [22] R. Sundar, On the deque conjecture for the splay algorithm, Combinatorica, 12 (1992), pp. 95– 124.
[23] R. Tarjan, Sequential access in splay trees takes linear time, Combinatoria, 5 (1985), pp. 367– 378.
[24] R. E. Tarjan, Amortized computational complexity, SI AM Journal on Algebraic Discrete Methods, 6 (1985), pp. 306–318.
[25] R. E. Wilber, Lower bounds for accessing binary search trees with rotations, SI AM J. Comput., 18 (1989), pp. 56–67.

Published by:

Dr.Santhi K
Department of Computer Science,Vellore Institute of Technology,Vellore,India

Urjit Divedi
Department of Computer Science,Vellore Insttitute of Technology,Vellore,India

Prakhar Dungarwal
Department of Computer Science,Vellore Institute of Technology,Vellore,India