Name	Shubhan Singh
UID no.	2022300118
Experiment No.	6

Program 1	
PROBLEM STATEMENT :	*Implement an ADT for storing an AVL Tree and performing given operations on it.*
ALGORITHM:	1. createNode - Allocate memory for a new AVLNode. - Calculate the length of the input string and allocate memory for the 'data' in the new node. - Copy the input data to the 'data' field. - Initialize 'left' and 'right' pointers to NULL. - Set the 'height' of the new node to 0. - Return the new node. 2. displayAVLtree - If 'root' is NULL, return. - Recursively traverse the left subtree by calling displayAVLTree(root->left). - Print the data of the current node 'root'. - Recursively traverse the right subtree by calling displayAVLTree(root->right). 3. getHeight

- If 'node' is NULL, return -1 (height of a NULL node).
- If 'node' has both left and right children, return the maximum of the heights of its children plus 1.
- If 'node' has only a right child, return the height of the right child plus 1.
- If 'node' has only a left child, return the height of the left child plus 1.
- If 'node' has no children, return 0 (height of a leaf node).

4. rotateRight

- Store 'node->left->right' in 'moved'.
- Store 'node->left' in 'newroot'.
- Update 'newroot->right' to point to 'node'.
- Update 'node->left' to point to 'moved'.
- Return 'newroot'.

5. rotateLeft

- Store 'node->right->left' in 'moved'.
- Store 'node->right' in 'newroot'.
- Update 'newroot->left' to point to 'node'.
- Update 'node->right' to point to 'moved'.
- Return 'newroot'.

6. getBalance

- If both 'node->left' and 'node->right' are NULL, return 0.
- If 'node->right' is NULL, return '-1 - node->left->height'.
- If 'node->left' is NULL, return 'node->right->height + 1'.

	- Otherwise, return 'node->right->height - node->left->height'. 7. insertNode - If the root is NULL (empty tree), create a new node with 'str' and set it as the root node. - Otherwise, start at the root node and follow the appropriate path to insert 'str'. - Compare 'str' with the data in the current node. - If 'str' is less than or equal to the current node's data, move to the left child. - If 'str' is greater, move to the right child. - Repeat the comparison and traversal until you find an empty slot to insert 'str'. - Create a new node with 'str' and insert it in the found empty slot (either as the left or right child, depending on the comparison result). - After inserting the node, calculate the height of each node in the path back to the root. - Check for any imbalances in the path by comparing the balance factor (difference in height of left and right subtrees) at each node. - If an imbalance is detected (balance factor is 2 or -2), perform the necessary rotations to restore the AVL property. - Update the heights of the affected nodes after rotations. - Continue checking and fixing imbalances until the root is reached. - Finally, update the height of the root node and check for imbalances again to ensure the entire tree remains balanced. 8. delete - Initialize an array of AVLNode* 'nodestack' to keep track of visited nodes, and variables 'current', 'parent', 'index', and 'd'. - Start at the root of the tree and traverse it to find the node to be deleted while storing visited nodes in 'nodestack'.

	- During traversal, compare 'data' with the data in the current node: - If 'data' is less than the current node's data, move to the left child. - If 'data' is greater, move to the right child. - Repeat the comparison and traversal until you find the node with the matching data or reach a NULL node. - Determine whether the node to be deleted is the left or right child of 'parent' and set 'd' accordingly (0 for left, 1 for right). - If the node to be deleted has both left and right children (neither child is NULL), perform the following steps: - Find the in-order predecessor (the largest node in the left subtree, i.e., the rightmost node in the left subtree). - Copy the data from the in-order predecessor to the node to be deleted. - Update 'current' to point to the in-order predecessor (the node with the data you just copied). - Continue the deletion process for 'current'. - If the node to be deleted has one or no children, handle three cases: 1. If both left and right children are NULL (a leaf node), set 'parent's left or right pointer (depending on 'd') to NULL and free 'current'. 2. If 'current' has a left child (but not a right child), set 'parent's left or right pointer (depending on 'd') to point to 'current's left child and free 'current'. 3. If 'current' has a right child (but not a left child), set 'parent's left or right pointer (depending on 'd') to point to 'current's right child and free 'current'. - After deleting the node, traverse the 'nodestack' in reverse order and check each node for imbalances. - Calculate the height of each node and check for imbalances at each node. - If an imbalance is detected (balance factor is 2 or -2), perform the necessary rotations to restore the AVL property. - Update the heights of the affected nodes after rotations. - Continue checking and fixing imbalances until the root is reached. - Finally, update the height of the root node and check for imbalances again to ensure the entire tree remains

	balanced. 9. freeAVLTree - If 'root' is NULL, return. - Recursively free the left subtree by calling freeAVLTree(root->left). - Recursively free the right subtree by calling freeAVLTree(root->right). - Free the memory for the current node 'root'.
PROGRAM:	

```c
// Create an AVLNode ADT and complete the following functions:
// 1. createNode
// 2. displayAVLTree
// 3. getHeight
// 4. rotateRight
// 5. rotateLeft
// 6. getBalance
// 7. delete
// 8. freeAVLTree

#include <stdio.h>
#include <stdlib.h>
#include<string.h>

typedef struct AVLNode {//description of node of an AVL tree which holds strings
    char* data;
    struct AVLNode* left;
```

```
    struct AVLNode* right;
    int height;
} AVLNode;

int max(int a, int b){//function to find max of two integers
    if(a>=b){return a;}
    return b;
}
int abs(int a){
    if(a>=0){return a;}
    return -1*a;
}
void remove_newline(char* str){//function to remove newline from fgets input
    char* ptr = strchr(str, '\n'); //strchr returns pointer to location where
given character was found
    if (ptr) {
        *ptr = '\0';
    }
}
// complete this function
AVLNode* createNode(char* data){//function to initialise a node containing given
data
    AVLNode* newnode=malloc(sizeof(AVLNode));
    int len=strlen(data);
    newnode->data=malloc((len+1)*sizeof(char));
```

```
    strcpy(newnode->data,data);
    newnode->left=newnode->right=NULL;
    newnode->height=0;
    return newnode;
}

// Display the in-order traversal of the Tree
void displayAVLTree(AVLNode* root){//function for inorder traversal of tree
    if(root==NULL){return;}
    displayAVLTree(root->left);
    printf("%s ",root->data);
    displayAVLTree(root->right);
}

void displaypreorder(AVLNode* root){//function for preorder traversal of tree
    if(root==NULL){return;}
    printf("%s ",root->data);
    displaypreorder(root->left);
    displaypreorder(root->right);
}

void displaytree(AVLNode* root, void (*dispfunc)(AVLNode* a)){//wrapper function
for adding square brackets ar the
//start and end of the traversal outputs;
    printf("[ ");
```

```
    dispfunc(root);
    printf("]\n");
}

int getHeight(AVLNode* node){//function to find height of any given node based
on height of its children
//height of NULL node has been takes as -1 for calculation purposes
    if(node==NULL){return -1;}
    if(node->right!=NULL && node->left!=NULL){
        return max(node->right->height,node->left->height)+1;
    }
    else if(node->left==NULL && node->right!=NULL){
        return node->right->height+1;
    }
    else if(node->left!=NULL && node->right==NULL){
        return node->left->height+1;
    }
    else{return 0;}
}

// function for performing a right rotate
AVLNode* rotateRight(AVLNode* node){
    AVLNode* moved=node->left->right;
    AVLNode* newroot= node->left;
    newroot->right=node;
```

```
    node->left=moved;
    return newroot;
}

// function for performing a left rotate
AVLNode* rotateLeft(AVLNode* node){
    AVLNode* moved=node->right->left;
    AVLNode* newroot= node->right;
    newroot->left=node;
    node->right=moved;
    return newroot;
}
// get balance factor of given node
int getBalance(AVLNode* node){
    if(node->left==node->right){//they can be equal only when both are null
        return 0;
    }
    if(node->right==NULL){
        return -1-node->left->height;
    }
    if(node->left==NULL){
        return node->right->height+1;
    }
    return node->right->height-node->left->height;
}
```

```
AVLNode* fixsubtree(AVLNode* root,int balance){
    if(balance>1){//4 cases of rotations described here.
        if(getBalance(root->right)>=0){
            root=rotateLeft(root);
        }
        else{
            root->right=rotateRight(root->right);
            root=rotateLeft(root);
        }
    }
    else{
        if(getBalance(root->left)<0){
            root=rotateRight(root);
        }
        else{
            root->left=rotateLeft(root->left);
            root=rotateRight(root);
        }
    }
    root->left->height=getHeight(root->left);
    root->right->height=getHeight(root->right);
    root->height=getHeight(root);//set new values of heights after performing
roations
    return root;
```

```
}

void insertNode(AVLNode** root, char* str){
    if(strcmp((*root)->data,str)>=0){//search for correct position to insert
node at
        if((*root)->left!=NULL)
            insertNode(&((*root)->left),str);
        else
            (*root)->left=createNode(str);
    }
    else{
        if((*root)->right!=NULL)
            insertNode(&((*root)->right),str);
        else
            (*root)->right=createNode(str);
    }
    (*root)->height=getHeight((*root));//after inserting, check for imbalance at
all the nodes in the path and fix that
    if(abs(getBalance((*root)))==2){
        (*root)=fixsubtree((*root),getBalance((*root)));
    }
}
// This deletes a node with 'data' into the AVL tree
// Please ensure that your function covers all 4 possible rotation cases
```

```
int delete3cases(AVLNode* current,AVLNode* parent,int d){//this part was
repeated twice in the code,
//so i made a function to make it shorter
    if(current->left==current->right){
        if(d==1)
            parent->right=NULL;
        else
            parent->left=NULL;

        free(current);
    }
    else if(current->left==NULL){
        if(d==1)
            parent->right=current->right;
        else
            parent->left=current->right;
        free(current);
    }
    else if(current->right==NULL){
        if(d==1)
            parent->right=current->left;
        else
            parent->left=current->left;
        free(current);
    }
```

```
    else{return 1;}
    return 0;
}
void deleteNode(AVLNode** root, char* data){
    //the delete function works by finding the node to be deleted, delete it as
it would be deleted in a binary tree,
    //then check all the nodes visited in the path to the deleted node and check
for imbalances and fix them.
    //we use an array to store all the nodes previously visited, we use this
array as a stack, though without formally creating an ADT
    AVLNode* nodestack[(*root)->height + 1];
    AVLNode* current=(*root);
    if(current->left==current->right){
        printf("Cannot delete only node left\n");
        return;
    }
    AVLNode* parent;
    int index=0,cmp,d=0;
    while(1){//find node to be delelted while storing all visited nodes in the
stack
        if(current==NULL){
            printf("Data not found\n");
            return;
        }
        cmp=strcmp(current->data,data);
```

```
        if(cmp>0){
            nodestack[index]=current;
            index++;
            current=current->left;
        }
        else if(cmp<0){
            nodestack[index]=current;
            index++;
            current=current->right;
        }
        else{break;}
    }
    if(index>0){//this condition is used so that the index doesn't become
negative if he
    // root itself was the node to be deleted
        parent=nodestack[index-1];
        if(parent->right==current){d=1;}
        else{d=0;}
    }
    int doelse=0;
    doelse=delete3cases(current,parent,d);//deletes the node if either one or
both children are null,
    //or returns that no children are null
    if(doelse){//in that case, find inorder predecessor, coopy its data to the
node to be deleted,
```

```
    // and then delte the node the data was copied
        AVLNode* temp=current;
        nodestack[index]=current;
        index++;
        current=current->left;
        while(current->right!=NULL){
            nodestack[index]=current;
            index++;
            current=current->right;
        }
        parent=nodestack[index-1];
        if(parent->right==current){d=1;}
        else{d=0;}
        strcpy(temp->data,current->data);
        doelse=delete3cases(current,parent,d);//an inorder predecessor would
have no right child, in that
        //case we don't have to worry about deletion of nodes with 2 children
    }
    index-=2;
    for(int i=index;i>=0;i--){//visit all previously visited nodes, check for
imbalance and restore AVL property
        current=nodestack[index+1];
        parent=nodestack[index];
        current->height=getHeight(current);
        if(abs(getBalance(current))==2){
```

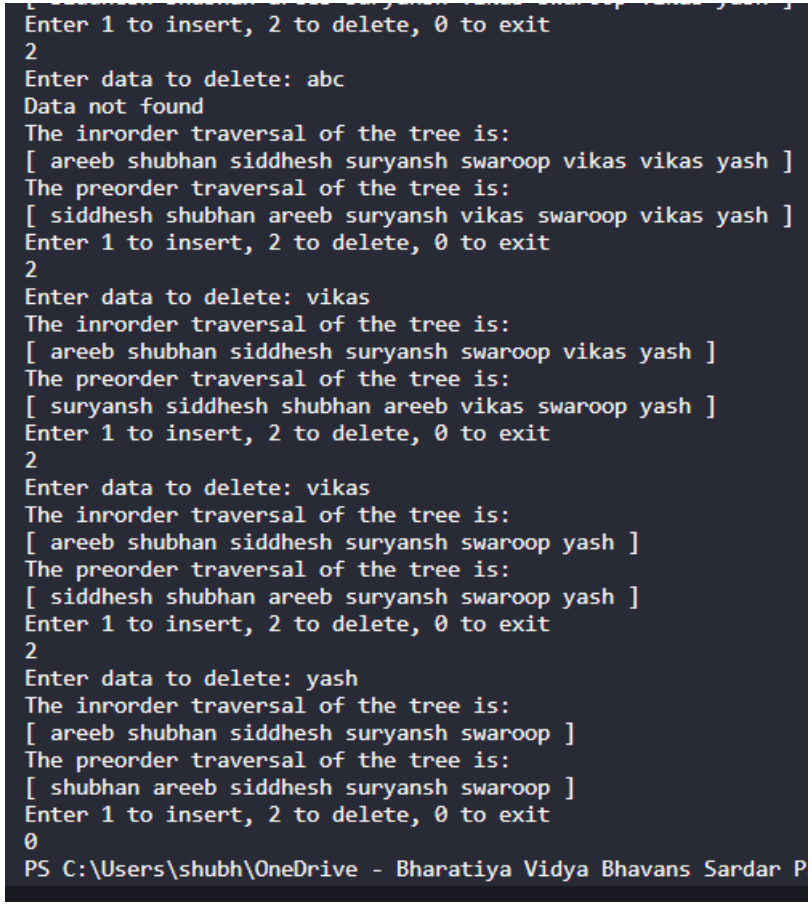
```
            if(parent->right==current)
                parent->right=fixsubtree(current,getBalance(current));
            else
                parent->left=fixsubtree(current,getBalance(current));
        }
    }
    (*root)->height=getHeight((*root));
    (*root)=fixsubtree((*root),getBalance((*root)));
}

// This frees the memory used by the AVL tree
void freeAVLTree(AVLNode* root){
    if(root==NULL){return;}
    if(root->left==NULL && root->right==NULL){
        free(root);
        return;
    }
    freeAVLTree(root->left);
    freeAVLTree(root->right);
}

int main(){
    int n=1;
    printf("Enter first element in tree (string):");
    char str[101];
```

```
    fgets(str,101,stdin);
    remove_newline(str);//fgets also takes the '\n' in the input, so it has to
be removed
    AVLNode* root=createNode(str);//initialises root
    displaytree(root,displayAVLTree);
    while(n){
        printf("Enter 1 to insert, 2 to delete, 0 to exit\n");
        scanf("%d",&n);
        while((getchar())!='\n');//removes newline character from buffer, or
else fgets will red this and ignore the string
        if(n==0){break;}
        if(n==1){
            printf("Enter string to insert:");
            fgets(str,101,stdin);
            remove_newline(str);//takes string
            insertNode(&root,str);//inserts it
            printf("The inrorder traversal of the tree is:\n");//displays
inorder and preorder traversals of the tree
            //both are displayed because inorder traversal would always just be
the sorted order.
            displaytree(root,displayAVLTree);
            printf("The preorder traversal of the tree is:\n");
            displaytree(root,displaypreorder);
        }
        if(n==2){//similar procedure for delete
```

```
            printf("Enter data to delete: ");
            fgets(str,101,stdin);
            remove_newline(str);
            deleteNode(&root,str);
            printf("The inrorder traversal of the tree is:\n");
            displaytree(root,displayAVLTree);
            printf("The preorder traversal of the tree is:\n");
            displaytree(root,displaypreorder);
        }
    }
    freeAVLTree(root);
}
```

RESULT:

```
PS C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Ins
tute Of Technology\C programs\data structures\exp_6_2022300118_compsB_
ithiterativedelete }
Enter first element in tree (string):shubhan
[ shubhan ]
Enter 1 to insert, 2 to delete, 0 to exit
1
Enter string to insert:swaroop
The inrorder traversal of the tree is:
[ shubhan swaroop ]
The preorder traversal of the tree is:
[ shubhan swaroop ]
Enter 1 to insert, 2 to delete, 0 to exit
1
Enter string to insert:vikas
The inrorder traversal of the tree is:
[ shubhan swaroop vikas ]
The preorder traversal of the tree is:
[ swaroop shubhan vikas ]
Enter 1 to insert, 2 to delete, 0 to exit
1
Enter string to insert:suryansh
The inrorder traversal of the tree is:
[ shubhan suryansh swaroop vikas ]
The preorder traversal of the tree is:
[ swaroop shubhan suryansh vikas ]
Enter 1 to insert, 2 to delete, 0 to exit
1
Enter string to insert:yash
The inrorder traversal of the tree is:
[ shubhan suryansh swaroop vikas yash ]
The preorder traversal of the tree is:
[ swaroop shubhan suryansh vikas yash ]
Enter 1 to insert, 2 to delete, 0 to exit
1
Enter string to insert:nakshatra
The inrorder traversal of the tree is:
[ nakshatra shubhan suryansh swaroop vikas yash ]
The preorder traversal of the tree is:
[ swaroop shubhan nakshatra suryansh vikas yash ]
Enter 1 to insert, 2 to delete, 0 to exit
1
Enter string to insert:adil
The inrorder traversal of the tree is:
[ adil nakshatra shubhan suryansh swaroop vikas yash ]
The preorder traversal of the tree is:
[ swaroop shubhan nakshatra adil suryansh vikas yash ]
Enter 1 to insert, 2 to delete, 0 to exit
1
Enter string to insert:siddhesh
The inrorder traversal of the tree is:
[ adil nakshatra shubhan siddhesh suryansh swaroop vikas yash ]
The preorder traversal of the tree is:
[ swaroop shubhan nakshatra adil suryansh siddhesh vikas yash ]
Enter 1 to insert, 2 to delete, 0 to exit
1
Enter string to insert:rahul
The inrorder traversal of the tree is:
[ adil nakshatra rahul shubhan siddhesh suryansh swaroop vikas yash ]
The preorder traversal of the tree is:
[ swaroop shubhan nakshatra adil rahul suryansh siddhesh vikas yash ]
Enter 1 to insert, 2 to delete, 0 to exit
2
Enter data to delete: rahul
The inrorder traversal of the tree is:
[ adil nakshatra shubhan siddhesh suryansh swaroop vikas yash ]
The preorder traversal of the tree is:
[ suryansh shubhan nakshatra adil siddhesh swaroop vikas yash ]
Enter 1 to insert, 2 to delete, 0 to exit
2
Enter data to delete: adil
The inrorder traversal of the tree is:
[ nakshatra shubhan siddhesh suryansh swaroop vikas yash ]
The preorder traversal of the tree is:
[ siddhesh shubhan nakshatra suryansh swaroop vikas yash ]
Enter 1 to insert, 2 to delete, 0 to exit
1
Enter string to insert:vikas
The inrorder traversal of the tree is:
[ nakshatra shubhan siddhesh suryansh swaroop vikas vikas yash ]
The preorder traversal of the tree is:
[ suryansh siddhesh shubhan nakshatra vikas swaroop vikas yash ]
Enter 1 to insert, 2 to delete, 0 to exit
1
Enter string to insert:areeb
The inrorder traversal of the tree is:
[ areeb nakshatra shubhan siddhesh suryansh swaroop vikas vikas yash ]
The preorder traversal of the tree is:
[ suryansh shubhan nakshatra areeb siddhesh vikas swaroop vikas yash ]
Enter 1 to insert, 2 to delete, 0 to exit
2
Enter data to delete: nakshatra
The inrorder traversal of the tree is:
[ areeb shubhan siddhesh suryansh swaroop vikas vikas yash ]
The preorder traversal of the tree is:
[ siddhesh shubhan areeb suryansh vikas swaroop vikas yash ]
Enter 1 to insert, 2 to delete, 0 to exit
2
Enter data to delete: abc
Data not found
```



THEORY:	An AVL (Adelson-Velsky and Landis) tree is a self-balancing binary search tree data structure. It is named after its inventors, Georgy Adelson-Velsky and Evgenii Landis, who introduced it in 1962. AVL trees are designed to maintain balance in a binary search tree to ensure efficient operations for searching, insertion, and deletion. Here are some key characteristics and a brief overview of AVL trees: 1. **Balance Factor**: In an AVL tree, each node has a balance factor, which is the height of the right subtree minus the height of the left subtree. The balance factor helps in measuring how balanced or unbalanced a tree is. It can

	be one of {-1, 0, 1} for every node to maintain balance. 2. **Balancing Property**: The balancing property of an AVL tree ensures that the balance factor of every node remains within the range [-1, 0, 1]. When an element is inserted or deleted, and this property is violated, rotations are performed to restore balance. These rotations can be single or double rotations (right and left rotations). 3. **Search Operation**: AVL trees maintain a binary search tree property, which means the left subtree of a node contains values smaller than the node's value, and the right subtree contains values greater than the node's value. This property ensures efficient searching, with a time complexity of O(log n). 4. **Insertion and Deletion**: When new elements are inserted or existing elements are deleted from an AVL tree, the tree may become unbalanced. To maintain the balance, rotations are performed at specific nodes as needed, following the order of insertion or deletion. 5. **Complexity Analysis**: • Search: O(log n) • Insertion: O(log n) (amortized) • Deletion: O(log n) (amortized) 6. **Advantages**: • Provides efficient searching due to the binary search tree property. • Self-balancing, ensuring that the height of the tree is always logarithmic, which results in consistent time complexities. • Predictable performance, which makes it suitable for real-time and critical systems. 7. **Limitations**: • Requires extra storage to maintain balance factors, which increases memory usage. • The overhead of rebalancing may affect performance for a large number of insertions and deletions. 8. **Applications**: • Database systems for indexing and searching. • Implementations of sorted data structures. • In compilers for optimizing expressions and decision trees. • In self-balancing binary search tree libraries and data structures.

CONCLUSION:	AVLtrees are an important data structure useful in creating balanced Binary trees which are faster to search, and searching always takes O(logn) time.