**LAB SESSION 9: AVL (Adelson -Velsky and Landis) Trees**

**AIM**: To implement AVL trees and perform the listed operations on such trees.

**PROBLEM DEFINITION:**

Develop a C program to create an AVL tree and perform the following operations

1. Insertion of a new element

2. Deletion of an existing element

3. Searching for a given element

4. Find the maximum and minimum value

**THEORY**:

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors. AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

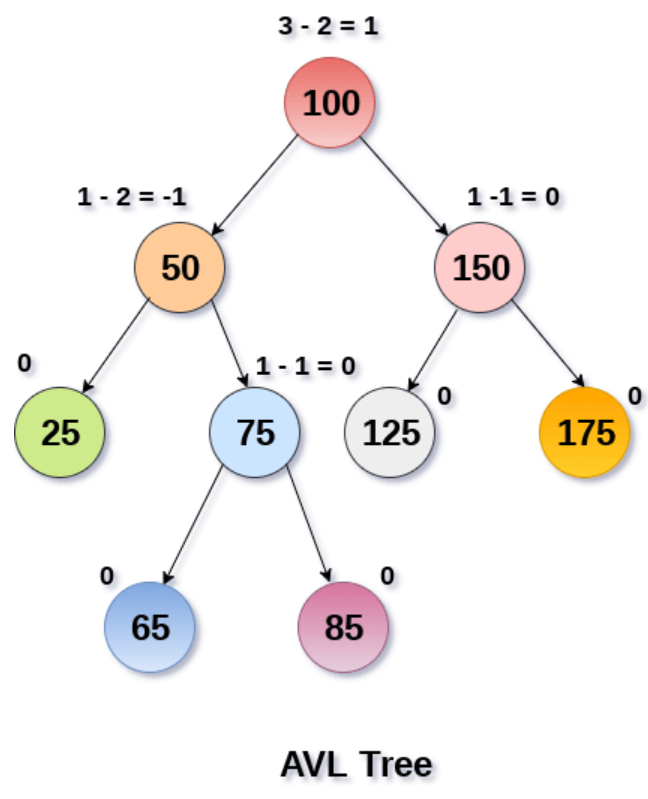
Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree. If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height. If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree. If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



Operations on AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

|  |  |  |
| --- | --- | --- |
| **SN** | **Operation** | **Description** |
| 1 | [Insertion](https://www.javatpoint.com/insertion-in-avl-tree) | Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations. |
| 2 | [Deletion](https://www.javatpoint.com/deletion-in-avl-tree) | Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree |

**ALGORITHMS**:

1. Insertion of a new element

1. Initialize a variable taller to TRUE.
2. If pptr is NULL, create a new node with the key ikey, set its left and right children to NULL, set its balance factor to 0, and set taller to TRUE.
3. Otherwise, if ikey is less than pptr’s key, recursively insert ikey into pptr’s left subtree.
4. If taller is TRUE, call insert\_left\_check on pptr and taller.
5. Otherwise, if ikey is greater than pptr’s key, recursively insert ikey into pptr’s right subtree.
6. If taller is TRUE, call insert\_right\_check on pptr and taller.
7. Otherwise, print “Duplicate key”.
8. Return pptr.

2. Deletion of an existing element

1. Initialize a variable shorter to FALSE.
2. If pptr is NULL, print “Key not present” and return pptr.
3. Otherwise, if dkey is less than pptr’s key, recursively delete dkey from pptr’s left subtree.
4. If shorter is TRUE, call del\_left\_check on pptr and shorter.
5. Otherwise, if dkey is greater than pptr’s key, recursively delete dkey from pptr’s right subtree.
6. If shorter is TRUE, call del\_right\_check on pptr and shorter.
7. Otherwise, if dkey is equal to pptr’s key, do the following:
   * If pptr has two children, set succ to the leftmost child of pptr’s right subtree.
   * While succ has a left child, set succ to its left child.
   * Set pptr’s key to succ’s key.
   * Recursively delete succ’s key from pptr’s right subtree.
   * If shorter is TRUE, call del\_right\_check on pptr and shorter.
   * Otherwise, if pptr has one child, set tmp to pptr, set pptr to pptr’s child, and free tmp.
   * Otherwise, set pptr to NULL.
   * Set shorter to TRUE.
8. Return pptr.

**PROGRAM AND OUTPUT:**

#include<stdio.h>

#include<stdlib.h>

#define TRUE 1

#define FALSE 0

struct node{

    struct node \*lchild;

    int info;

    struct node \*rchild;

    int balance;

};

struct node \*RightRotation(struct node \*pptr){

    struct node \*aptr;

    aptr = pptr->lchild;

    pptr->lchild = aptr->rchild;

    aptr->rchild = pptr;

    return aptr;

}

struct node \*LeftRotation(struct node \*pptr){

    struct node \*aptr;

    aptr = pptr->rchild;

    pptr->rchild = aptr->lchild;

    aptr->lchild = pptr;

    return aptr;

}

struct node \*insert\_leftbalance(struct node \*pptr){

    struct node \*aptr, \*bptr;

    aptr = pptr->lchild;

    if (aptr->balance == 1){

        pptr->balance = 0;

        aptr->balance = 0;

        pptr = RightRotation(pptr);

    }

    else{

        bptr = aptr->rchild;

        switch (bptr->balance){

            case -1:

                pptr->balance = 0;

                aptr->balance = 1;

                break;

            case 1:

                pptr->balance = -1;

                aptr->balance = 0;

                break;

            case 0:

                pptr->balance = 0;

                aptr->balance = 0;

        }

        bptr->balance = 0;

        pptr->lchild = LeftRotation(pptr->lchild);

        pptr = RightRotation(pptr);

    }

    return pptr;

}

struct node \*insert\_rightbalance(struct node \*pptr){

    struct node \*aptr, \*bptr;

    aptr = pptr->rchild;

    if (aptr->balance == -1){

        pptr->balance = 0;

        aptr->balance = 0;

        pptr = LeftRotation(pptr);

    }

    else{

        bptr = aptr->lchild;

        switch (bptr->balance){

            case -1:

                pptr->balance = 1;

                aptr->balance = 0;

                break;

            case 1:

                pptr->balance = 0;

                aptr->balance = -1;

                break;

            case 0:

                pptr->balance = 0;

                aptr->balance = 0;

        }

        bptr->balance = 0;

        pptr->rchild = RightRotation(pptr->rchild);

        pptr = LeftRotation(pptr);

    }

    return pptr;

}

struct node \*insert\_left\_check(struct node \*pptr, int \*ptaller){

    switch (pptr->balance){

        case 0:

            pptr->balance = 1;

            break;

        case 1:

            pptr->balance = 0;

            \*ptaller = FALSE;

            break;

        case -1:

            pptr = insert\_leftbalance(pptr);

            \*ptaller = FALSE;

    }

    return pptr;

}

struct node \*insert\_right\_check(struct node \*pptr, int \*ptaller){

    switch (pptr->balance){

        case 0:

            pptr->balance = -1;

            break;

        case 1:

            pptr->balance = 0;

            \*ptaller = FALSE;

            break;

        case -1:

            pptr = insert\_rightbalance(pptr);

            \*ptaller = FALSE;

    }

    return pptr;

}

struct node \*insert(struct node \*pptr, int ikey){

    static int taller;

    if (pptr == NULL){

        pptr = (struct node \*)malloc(sizeof(struct node));

        pptr->info = ikey;

        pptr->lchild = NULL;

        pptr->rchild = NULL;

        pptr->balance = 0;

        taller = TRUE;

    }

    else if (ikey < pptr->info){

        pptr->lchild = insert(pptr->lchild, ikey);

        if (taller == TRUE)

            pptr = insert\_left\_check(pptr, &taller);

    }

    else if (ikey > pptr->info){

        pptr->rchild = insert(pptr->rchild, ikey);

        if (taller == TRUE)

            pptr = insert\_right\_check(pptr, &taller);

    }

    else{

        printf("Duplicate key\n");

        taller = FALSE;

    }

    return pptr;

}

struct node\* del\_rightbalance(struct node \*pptr,int \*pshorter){

    struct node \*bptr,\*aptr;

    aptr=pptr->rchild;

    if(aptr->balance==0){

        pptr->balance=-1;

        aptr->balance=1;

        \*pshorter=FALSE;

        pptr=LeftRotation(pptr);

    }

    else if(aptr->balance==-1){

        pptr->balance=0;

        aptr->balance=0;

        pptr=LeftRotation(pptr);

    }

    else{

        bptr=aptr->lchild;

        switch(bptr->balance){

            case 0:

                pptr->balance=0;

                aptr->balance=0;

                break;

            case 1:

                pptr->balance=0;

                aptr->balance=-1;

                break;

            case -1:

                pptr->balance=1;

                aptr->balance=0;

                break;

        }

        bptr->balance=0;

        pptr->rchild = LeftRotation(aptr);

        pptr=RightRotation(pptr);

    }

    return pptr;

}

struct node \*del\_left\_check(struct node \*pptr,int \*pshorter){

    switch(pptr->balance){

        case 0:

            pptr->balance=-1;

            \*pshorter=FALSE;

            break;

        case 1:

            pptr->balance=0;

            break;

        case -1:

            pptr=del\_rightbalance(pptr,pshorter);

            break;

    }

    return pptr;

}

struct node\* del\_leftbalance(struct node \*pptr, int \*pshorter){

    struct node \*bptr, \*aptr;

    aptr = pptr->rchild;

    if (aptr->balance == 0) {

        pptr->balance = 1;

        aptr->balance = -1;

        \*pshorter = FALSE;

        pptr = LeftRotation(pptr);

    }

    else if (aptr->balance == -1){

        pptr->balance = 0;

        aptr->balance = 0;

        pptr = LeftRotation(pptr);

    }

    else {

        bptr = aptr->lchild;

        switch (bptr->balance){

            case 0:

                pptr->balance = 0;

                aptr->balance = 0;

                break;

            case 1:

                pptr->balance = -1;

                aptr->balance = 0;

                break;

            case -1:

                pptr->balance = 0;

                aptr->balance = 1;

                break;

        }

        bptr->balance = 0;

        pptr->rchild = RightRotation(aptr);

        pptr = LeftRotation(pptr);

    }

    return pptr;

}

struct node \*del\_right\_check(struct node \*pptr, int \*pshorter){

    switch (pptr->balance){

        case 0:

            pptr->balance = 1;

            \*pshorter = FALSE;

            break;

        case -1:

            pptr->balance = 0;

            break;

        case 1:

            pptr = del\_leftbalance(pptr, pshorter);

            break;

    }

    return pptr;

}

struct node \*delete(struct node \* pptr, int dkey){

    struct node \* tmp, \* succ;

    static int shorter;

    if(pptr == NULL){

        printf("Key not present\n");

        shorter = FALSE;

        return pptr;

    }

    if(dkey<pptr->info){

        pptr->lchild=delete(pptr->lchild,dkey);

        if(shorter==TRUE){

            pptr=del\_left\_check(pptr,&shorter);

        }

    }

    else if(dkey>pptr->info){

        pptr->rchild = delete(pptr->rchild,dkey);

        if(shorter==TRUE){

            pptr=del\_right\_check(pptr,&shorter);

        }

    }

    else{

        if(pptr->lchild!=NULL && pptr->lchild!=NULL){

            succ=pptr->lchild;

            while(succ->lchild)

                succ=succ->lchild;

            pptr->info=succ->info;

            pptr->rchild=delete(pptr->rchild,succ->info);

            if(shorter==TRUE)

                pptr=del\_right\_check(pptr,&shorter);

        }

        else{

            tmp = pptr;

            if(pptr->lchild!=NULL)

                pptr=pptr->lchild;

            else if(pptr->rchild!=NULL)

                pptr=pptr->rchild;

            else

                pptr=NULL;

            free(tmp);

            shorter=TRUE;

        }

    }

    return pptr;

}

void search(struct node \*ptr, int skey){

    static int found = 0;

    if(ptr == NULL){

        printf("Element not found\n");

        return;

    }

    else if(skey<ptr->info)

        search(ptr->lchild,skey);

    else if(skey>ptr->info)

        search(ptr->rchild,skey);

    else found=1;

    if(found == 0)

        printf("Element not found\n");

    else printf("Element found\n");

}

void minimum(struct node \*ptr){

    if(ptr!=NULL)

        while(ptr->lchild!=NULL)

            ptr=ptr->lchild;

    printf("Minimum element is : %d\n",ptr->info);

}

void maximum(struct node \*ptr){

    if(ptr!=NULL)

        while(ptr->rchild!=NULL)

            ptr=ptr->rchild;

    printf("Maximum element is : %d\n",ptr->info);

}

int main(){

    struct node \*root = NULL;

    int s, key;

    do{

        printf("\n1. Insertion of a new element\n"

                "2. Deletion of an existing element\n"

                "3. Searching for a given element\n"

                "4. Find minimum and maximum element\n"

                "5. Exit\n");

        printf("Enter your option: ");

        scanf("%d", &s);

        switch(s){

            case 1: printf("Insert: ");

                    scanf("%d", &key);

                    root = insert(root, key);

                    break;

            case 2: printf("Delete: ");

                    scanf("%d", &key);

                    root = delete(root, key);

                    break;

            case 3: printf("Enter the element: ");

                    scanf("%d", &key);

                    search(root, key);

                    break;

            case 4: minimum(root);

                    maximum(root);

                    break;

            case 5: break;

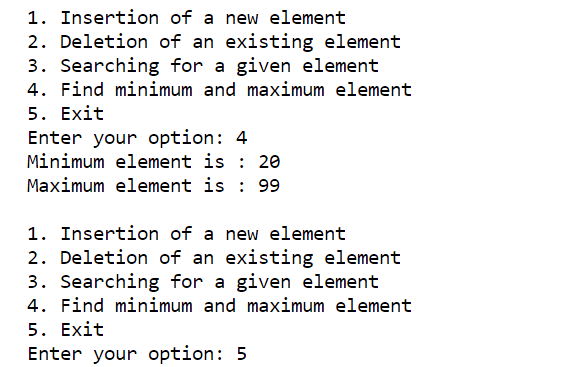
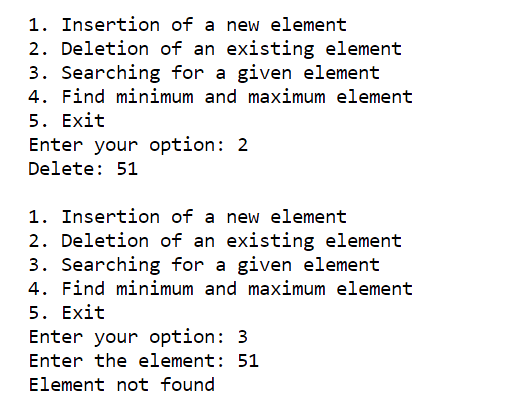
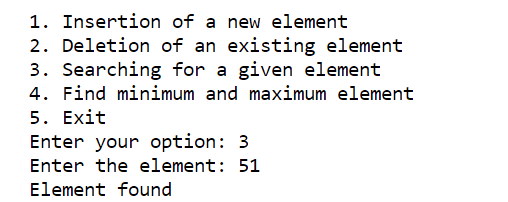
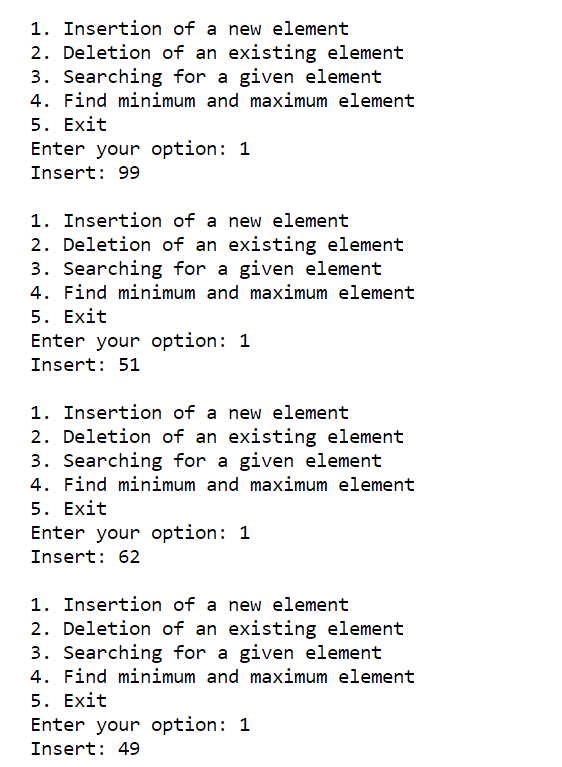
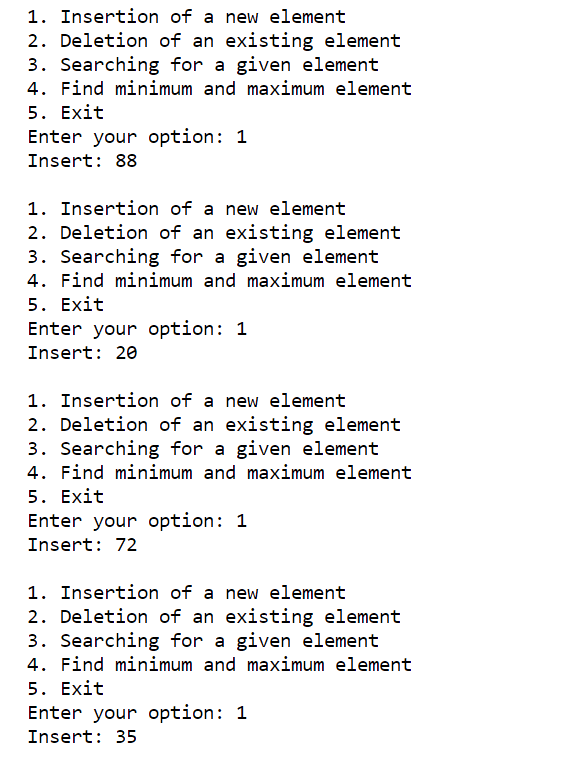
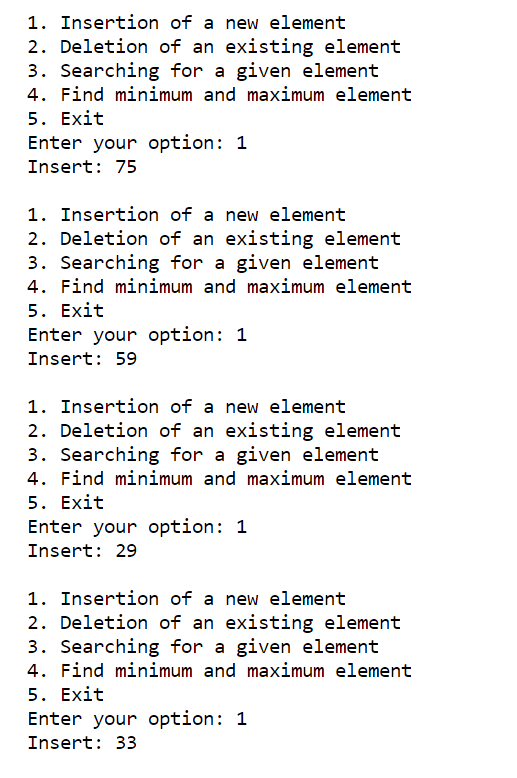
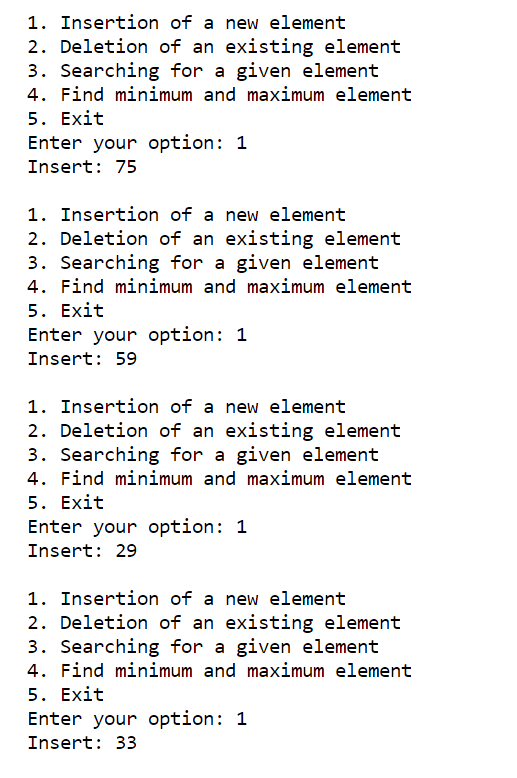
            default: printf("Invalid input\n");

        }

    }while(s != 5);

    return 0;

}



**CONCLUSION**: AVL was successfuly understood and implemented.