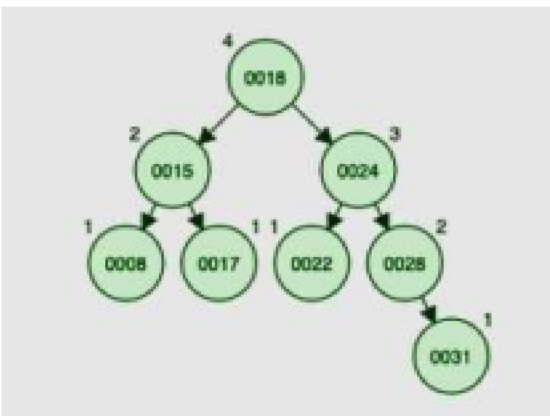
Week 8-LAB B

Lab Questions:

Q1. Delete the following values form the AVL tree in following fig.

1) 24
2) 15
3) 18
4) 22
5) 17



Ans:

```
#include <iostream>
using namespace std;

struct Node {
    int key;
    Node* left;
    Node* right;
    int height;
};

int height(Node* n) {
    if (n == NULL)
        return 0;
    return n->height;
}

Node* newNode(int key) {
    Node* node = new Node();
    node->key = key;
```

```
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return node;
}

Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    return x;
}

Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    return y;
}

int getBalance(Node* n) {
    if (n == NULL)
        return 0;
    return height(n->left) - height(n->right);
}

Node* insert(Node* node, int key) {
    if (node == NULL)
        return newNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
```

```
    else
        return node;

    node->height = 1 + max(height(node->left), height(node->right));

    int balance = getBalance(node);

    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

Node* minValueNode(Node* node) {
    Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}

Node* deleteNode(Node* root, int key) {
    if (root == NULL)
        return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else {
        if ((root->left == NULL) || (root->right == NULL)) {
            Node* temp = root->left ? root->left : root->right;

            if (temp == NULL) {
```

```
            temp = root;
            root = NULL;
        } else
            *root = *temp;

        delete temp;
    } else {
        Node* temp = minValueNode(root->right);
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
}

if (root == NULL)
    return root;

root->height = 1 + max(height(root->left), height(root->right));

int balance = getBalance(root);

if (balance > 1 && getBalance(root->left) >= 0)
    return rightRotate(root);

if (balance > 1 && getBalance(root->left) < 0) {
    root->left = leftRotate(root->left);
    return rightRotate(root);
}

if (balance < -1 && getBalance(root->right) <= 0)
    return leftRotate(root);

if (balance < -1 && getBalance(root->right) > 0) {
    root->right = rightRotate(root->right);
    return leftRotate(root);
}

return root;
}

void preOrder(Node* root) {
    if (root != NULL) {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

```
int main() {
    Node* root = NULL;

    // Insert the values from the AVL tree image
    root = insert(root, 18);
    root = insert(root, 15);
    root = insert(root, 24);
    root = insert(root, 8);
    root = insert(root, 17);
    root = insert(root, 22);
    root = insert(root, 28);
    root = insert(root, 31);

    cout << "Preorder traversal before deletion: ";
    preOrder(root);
    cout << endl;

    // Delete the specified nodes
    root = deleteNode(root, 24);
    root = deleteNode(root, 15);
    root = deleteNode(root, 18);
    root = deleteNode(root, 22);
    root = deleteNode(root, 17);

    cout << "Preorder traversal after deletion: ";
    preOrder(root);
    cout << endl;

    return 0;
}
```

```
Preorder traversal before deletion: 18 15 8 17 24 22 28 31
Preorder traversal after deletion: 28 8 31

Process returned 0 (0x0)   execution time : 0.141 s
Press any key to continue.
```

Q2. AVL Tree is one of the height-balanced binary search trees. Here, balancing of the BST is achieved using an additional field of balance factor associated with each node. Balance factor of a node, N is computed as the difference between the height of the left branch of N and the height of the right branch of N. In AVL tree, the balance factor of each node must be in the range between -1 and +1 (*i.e.* -1, 0, and 1).

In context of AVL tree, write programs for following:

(a) Compute the <u>height</u> of a binary tree/ binary search tree rooted at a node, N

Ans:

```
#include <iostream>
using namespace std;

struct node {
    int key;
    node* left;
    node* right;
};

node* newnode(int key) {
    node* n = new node();
    n->key = key;
    n->left = NULL;
    n->right = NULL;
    return n;
}

int height(node* n) {
    if (n == NULL)
        return 0;
    int leftheight = height(n->left);
    int rightheight = height(n->right);
    return 1 + max(leftheight, rightheight);
}
   int main() {
    node* root = newnode(1);
    root->left = newnode(2);
    root->right = newnode(3);
    root->left->left = newnode(4);
```

```
    root->left->right = newnode(5);

    cout << "height of the tree: " << height(root) << endl;

    return 0;

}
```

```
height of the tree: 3

Process returned 0 (0x0)   execution time : 0.063 s
Press any key to continue.
```

(b) Pre-order traversal of a binary search tree is given to you. Write a program to check whether the binary search tree formed with the given pre-order traversal is a valid AVL tree or not. E.g., if pre-order traversal of a binary search tree is 20, 10, 15, 18, 30, 25, and 40, then your program should display that it is not a valid AVL tree, whereas, your program should display a <u>valid AVL tree</u> if the pre-order traversal is given as 20, 15, 18, 30, 25, and 40.

Ans:

```
#include <iostream>
using namespace std;

struct node {
    int key;
    node* left;
    node* right;
};

node* newnode(int key) {
    node* n = new node();
    n->key = key;
    n->left = NULL;
    n->right = NULL;
    return n;
}

node* constructbst(int preorder[], int* preorderindex, int key, int min, int max, int n) {
    if (*preorderindex >= n)
        return NULL;

    node* root = NULL;

    if (key > min && key < max) {
        root = newnode(key);
        *preorderindex = *preorderindex + 1;
```

```
        if (*preorderindex < n) {
            root->left = constructbst(preorder, preorderindex, preorder[*preorderindex], min, key, n);
        }
        if (*preorderindex < n) {
            root->right = constructbst(preorder, preorderindex, preorder[*preorderindex], key, max, n);
        }
    }
    return root;
}

int height(node* n) {
    if (n == NULL)
        return -1;
    int leftheight = height(n->left);
    int rightheight = height(n->right);
    return 1 + max(leftheight, rightheight);
}

bool isavl(node* root) {
    if (root == NULL)
        return true;

    int leftheight = height(root->left);
    int rightheight = height(root->right);
    int balancefactor = leftheight - rightheight;

    if (balancefactor > 1 || balancefactor < -1)
        return false;

    return isavl(root->left) && isavl(root->right);
}

int main() {
    int preorder1[] = {20, 10, 15, 18, 30, 25, 40};
    int preorder2[] = {20, 15, 18, 30, 25, 40};

    int n1 = sizeof(preorder1) / sizeof(preorder1[0]);
    int n2 = sizeof(preorder2) / sizeof(preorder2[0]);

    int preorderindex1 = 0;
    int preorderindex2 = 0;

    node* root1 = constructbst(preorder1, &preorderindex1, preorder1[0], -1000000, 1000000, n1);
    node* root2 = constructbst(preorder2, &preorderindex2, preorder2[0], -1000000, 1000000, n2);
```

```
    if (isavl(root1))
        cout << "preorder1 forms a valid avl tree" << endl;
    else
        cout << "preorder1 does not form a valid avl tree" << endl;

    if (isavl(root2))
        cout << "preorder2 forms a valid avl tree" << endl;
    else
        cout << "preorder2 does not form a valid avl tree" << endl;

    return 0;
}
```

```
preorder1 does not form a valid avl tree
preorder2 forms a valid avl tree

Process returned 0 (0x0)   execution time : 0.160 s
Press any key to continue.
```

(c) You have been given two AVL trees A and B of height M and N respectively. Write a program to merge the AVL trees A and B into a new AVL tree C.

ANS:

```
#include <iostream>
#include <vector>
using namespace std;

struct node {
    int key;
    node* left;
    node* right;
    int height;
};

node* newnode(int key) {
    node* n = new node();
    n->key = key;
    n->left = NULL;
```

```
    n->right = NULL;
    n->height = 1;
    return n;
}

int getheight(node* n) {
    if (n == NULL)
        return 0;
    return n->height;
}

int updateheight(node* n) {
    return 1 + max(getheight(n->left), getheight(n->right));
}

int getbalance(node* n) {
    if (n == NULL)
        return 0;
    return getheight(n->left) - getheight(n->right);
}

node* rightrotate(node* y) {
    node* x = y->left;
    node* t2 = x->right;

    x->right = y;
    y->left = t2;

    y->height = updateheight(y);
    x->height = updateheight(x);

    return x;
}
```

```
node* leftrotate(node* x) {
    node* y = x->right;
    node* t2 = y->left;

    y->left = x;
    x->right = t2;

    x->height = updateheight(x);
    y->height = updateheight(y);

    return y;
}

node* insert(node* root, int key) {
    if (root == NULL)
        return newnode(key);

    if (key < root->key)
        root->left = insert(root->left, key);
    else if (key > root->key)
        root->right = insert(root->right, key);
    else
        return root;

    root->height = updateheight(root);

    int balance = getbalance(root);

    if (balance > 1 && key < root->left->key)
        return rightrotate(root);

    if (balance < -1 && key > root->right->key)
```

```
        return leftrotate(root);

    if (balance > 1 && key > root->left->key) {
        root->left = leftrotate(root->left);
        return rightrotate(root);
    }

    if (balance < -1 && key < root->right->key) {
        root->right = rightrotate(root->right);
        return leftrotate(root);
    }

    return root;
}

void inorder(node* root, vector<int>& result) {
    if (root == NULL)
        return;
    inorder(root->left, result);
    result.push_back(root->key);
    inorder(root->right, result);
}

vector<int> mergearrays(const vector<int>& arr1, const vector<int>& arr2) {
    vector<int> merged;
    int i = 0, j = 0;

    while (i < arr1.size() && j < arr2.size()) {
        if (arr1[i] < arr2[j])
            merged.push_back(arr1[i++]);
        else
            merged.push_back(arr2[j++]);
    }
```

```
    while (i < arr1.size())
        merged.push_back(arr1[i++]);

    while (j < arr2.size())
        merged.push_back(arr2[j++]);

    return merged;
}

node* sortedarraytoavl(const vector<int>& arr, int start, int end) {
    if (start > end)
        return NULL;

    int mid = (start + end) / 2;
    node* root = newnode(arr[mid]);

    root->left = sortedarraytoavl(arr, start, mid - 1);
    root->right = sortedarraytoavl(arr, mid + 1, end);

    root->height = updateheight(root);

    return root;
}

node* mergetrees(node* root1, node* root2) {
    vector<int> inorder1, inorder2;

    inorder(root1, inorder1);
    inorder(root2, inorder2);

    vector<int> merged = mergearrays(inorder1, inorder2);
```

```
    return sortedarraytoavl(merged, 0, merged.size() - 1);
}

void printinorder(const vector<int>& inorder, const string& tree_name) {
    cout << "inorder traversal of " << tree_name << ":" << endl;
    for (int i = 0; i < inorder.size(); ++i)
        cout << inorder[i] << " ";
    cout << endl;
}

int main() {
    node* root1 = NULL;
    node* root2 = NULL;

    root1 = insert(root1, 20);
    root1 = insert(root1, 10);
    root1 = insert(root1, 30);

    root2 = insert(root2, 25);
    root2 = insert(root2, 35);
    root2 = insert(root2, 15);

    vector<int> inorder1;
    inorder(root1, inorder1);
    printinorder(inorder1, "tree a");

    vector<int> inorder2;
    inorder(root2, inorder2);
    printinorder(inorder2, "tree b");

    node* root3 = mergetrees(root1, root2);

    vector<int> inorder3;
```

```
    inorder(root3, inorder3);
    printinorder(inorder3, "tree c");

    return 0;
}
```

```
inorder traversal of tree a:
10 20 30
inorder traversal of tree b:
15 25 35
inorder traversal of tree c:
10 15 20 25 30 35

Process returned 0 (0x0)   execution time : 0.210 s
Press any key to continue.
```

(d) It is desired to <u>delete entire sub-tree rooted</u> at an intermediate node N in an AVL tree, A. It can be done in single step by making left and right children of N as NULL. Call this modified tree as A'. Certainly, A' will not be a valid AVL tree. If possible, apply known AVL rotations or your own designed rotations on A' so that it will become a valid AVL tree.

ANS:

```
#include <iostream>

#include <cmath>

using namespace std;



class Node {

public:

   int key;

   Node* left;

   Node* right;

   int height;



   Node(int value) {

      key = value;
```

```
        left = NULL;
        right = NULL;
        height = 1;
    }
};

int getHeight(Node* node) {
    return node ? node->height : 0;
}

int getBalance(Node* node) {
    return node ? getHeight(node->left) - getHeight(node->right) : 0;
}

Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;
    x->right = y;
    y->left = T2;
    y->height = 1 + max(getHeight(y->left), getHeight(y->right));
    x->height = 1 + max(getHeight(x->left), getHeight(x->right));
    return x;
}

Node* leftRotate(Node* x) {
```

```
    Node* y = x->right;
    Node* T2 = y->left;
    y->left = x;
    x->right = T2;
    x->height = 1 + max(getHeight(x->left), getHeight(x->right));
    y->height = 1 + max(getHeight(y->left), getHeight(y->right));
    return y;
}

Node* rebalance(Node* node) {
    if (!node) return NULL;
    int balance = getBalance(node);
    if (balance > 1) {
        if (getBalance(node->left) < 0)
            node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    if (balance < -1) {
        if (getBalance(node->right) > 0)
            node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}
```

```
Node* deleteSubtree(Node* root, Node* nodeToDelete) {
    if (!root) return NULL;
    if (root == nodeToDelete) {
        nodeToDelete->left = NULL;
        nodeToDelete->right = NULL;
        return nodeToDelete;
    }
    root->left = deleteSubtree(root->left, nodeToDelete);
    root->right = deleteSubtree(root->right, nodeToDelete);
    root->height = 1 + max(getHeight(root->left), getHeight(root->right));
    return rebalance(root);
}

void printInOrder(Node* node) {
    if (!node) return;
    printInOrder(node->left);
    cout << node->key << " ";
    printInOrder(node->right);
}

Node* insert(Node* node, int key) {
    if (!node) return new Node(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
```

```
        node->right = insert(node->right, key);
    node->height = 1 + max(getHeight(node->left), getHeight(node->right));
    return rebalance(node);
}

int main() {
    Node* root = NULL;
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 10);
    root = insert(root, 25);

    cout << "Original AVL Tree Inorder: ";
    printInOrder(root);
    cout << endl;

    Node* nodeToDelete = root->left; // Assuming we want to delete subtree rooted at 20
    deleteSubtree(root, nodeToDelete);

    cout << "After Deleting Subtree Inorder: ";
    printInOrder(root);
    cout << endl;

    return 0;
```

```
}
```

```
Original AVL Tree Inorder: 10 20 25 30 40
After Deleting Subtree Inorder: 20 30 40

Process returned 0 (0x0)   execution time : 0.172 s
Press any key to continue.
```

(e) Let us consider, E as an element stored in a node N of an AVL tree, A. Considering an update operation, which updated the element E by ±Δ. With updated value as E±Δ at node N, A may not be a valid AVL tree. Call the updated tree as A`. Write a program to make the tree A` as <u>a valid AVL tree</u>.

Ans:

```
#include <iostream>
using namespace std;

struct node {
    int key;
    node* left;
    node* right;
    int height;
};

node* newnode(int key) {
    node* n = new node();
    n->key = key;
    n->left = NULL;
    n->right = NULL;
    n->height = 1;
    return n;
}

int getheight(node* n) {
```

```
    if (n == NULL)
        return 0;
    return n->height;
}

int updateheight(node* n) {
    return 1 + max(getheight(n->left), getheight(n->right));
}

int getbalance(node* n) {
    if (n == NULL)
        return 0;
    return getheight(n->left) - getheight(n->right);
}

node* rightrotate(node* y) {
    node* x = y->left;
    node* t2 = x->right;

    x->right = y;
    y->left = t2;

    y->height = updateheight(y);
    x->height = updateheight(x);

    return x;
}

node* leftrotate(node* x) {
    node* y = x->right;
    node* t2 = y->left;

    y->left = x;
```

```
    x->right = t2;

    x->height = updateheight(x);
    y->height = updateheight(y);

    return y;
}

node* insert(node* root, int key) {
    if (root == NULL)
        return newnode(key);

    if (key < root->key)
        root->left = insert(root->left, key);
    else if (key > root->key)
        root->right = insert(root->right, key);
    else
        return root;

    root->height = updateheight(root);

    int balance = getbalance(root);

    if (balance > 1 && key < root->left->key)
        return rightrotate(root);

    if (balance < -1 && key > root->right->key)
        return leftrotate(root);

    if (balance > 1 && key > root->left->key) {
        root->left = leftrotate(root->left);
        return rightrotate(root);
    }
```

```
    if (balance < -1 && key < root->right->key) {
        root->right = rightrotate(root->right);
        return leftrotate(root);
    }

    return root;
}

node* deletekey(node* root, int key) {
    if (root == NULL)
        return root;

    if (key < root->key)
        root->left = deletekey(root->left, key);
    else if (key > root->key)
        root->right = deletekey(root->right, key);
    else {
        if ((root->left == NULL) || (root->right == NULL)) {
            node* temp = root->left ? root->left : root->right;
            if (temp == NULL) {
                temp = root;
                root = NULL;
            } else
                *root = *temp;
            delete temp;
        } else {
            node* temp = root->right;
            while (temp->left != NULL)
                temp = temp->left;
            root->key = temp->key;
            root->right = deletekey(root->right, temp->key);
        }
```

```
    }

    if (root == NULL)
        return root;

    root->height = updateheight(root);

    int balance = getbalance(root);

    if (balance > 1 && getbalance(root->left) >= 0)
        return rightrotate(root);

    if (balance < -1 && getbalance(root->right) <= 0)
        return leftrotate(root);

    if (balance > 1 && getbalance(root->left) < 0) {
        root->left = leftrotate(root->left);
        return rightrotate(root);
    }

    if (balance < -1 && getbalance(root->right) > 0) {
        root->right = rightrotate(root->right);
        return leftrotate(root);
    }

    return root;
}

node* updatekey(node* root, int oldkey, int newkey) {
    root = deletekey(root, oldkey);
    root = insert(root, newkey);
    return root;
}
```

```
void inorder(node* root) {
    if (root == NULL)
        return;
    inorder(root->left);
    cout << root->key << " ";
    inorder(root->right);
}

int main() {
    node* root = NULL;

    root = insert(root, 20);
    root = insert(root, 10);
    root = insert(root, 30);
    root = insert(root, 25);
    root = insert(root, 5);

    cout << "inorder before update: ";
    inorder(root);
    cout << endl;

    int oldkey = 10;
    int delta = 11;
    int newkey = oldkey + delta;  // update operation

    root = updatekey(root, oldkey, newkey);

    cout << "inorder after update: ";
    inorder(root);
    cout << endl;

    return 0;
```

}

```
inorder before update: 5 10 20 25 30
inorder after update: 5 20 21 25 30

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```