LAB 7 :

LAB EXCERSIZE :

```
/*1)Modify the solved exercise to find the balance factor for every node in
the binary search tree.*/
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>

struct node
{
int data;
int balance_factor;
struct node *rightLink;
struct node *leftLink;
};

struct node *newNode(int item)
{
struct node *root = (struct node *)malloc(sizeof(struct node));
root->data = item;
root->balance_factor = 0;
root->leftLink = NULL;
root->rightLink = NULL;
return root;
}


void inorder(struct node *root)
{
if (root != NULL)
{
inorder(root->leftLink);
printf("%d\t", root->data);
inorder(root->rightLink);
}
}

void preorder(struct node *root)
```

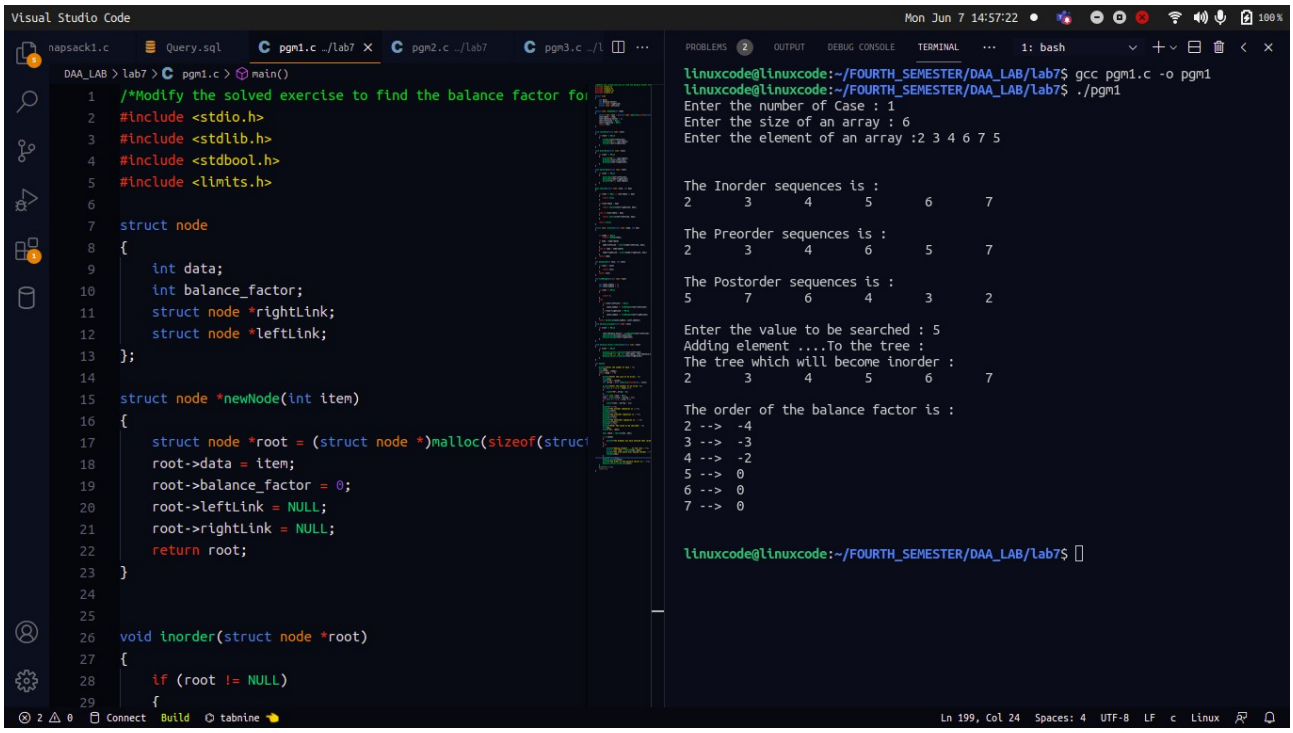
```
{
if (root != NULL)
{
printf("%d\t", root->data);
preorder(root->leftLink);
preorder(root->rightLink);
}
}

void postorder(struct node *root)
{
if (root != NULL)
{
postorder(root->leftLink);
postorder(root->rightLink);
printf("%d\t", root->data);
}
}

bool search(struct node *root, int key)
{
if (root == NULL || root->data == key)
{
return true;
}
if (root->data < key)
{
return search(root->rightLink, key);
}
else if (root->data > key)
{
return search(root->leftLink, key);
}
return false;
}

struct node *insert(struct node *node, int key)
{
if (node == NULL)
return newNode(key);
if (key < node->data)
{
```

```
node->leftLink = insert(node->leftLink, key);
}
else if (key > node->data)
{
node->rightLink = insert(node->rightLink, key);
}
return node;
}

int maxValue(int val1, int val2)
{
if (val1 > val2)
{
return val1;
}
return val2;
}

int findHeight(struct node *root)
{
int count_nodes1 = 1;
int count_nodes2 = 1;

if (root == NULL)
{

return 0;
}
else
{
if (root->leftLink != NULL)
{
count_nodes1 += findHeight(root->leftLink);
}
if (root->rightLink != NULL)
{
count_nodes2 += findHeight(root->rightLink);
}
}
return maxValue(count_nodes1, count_nodes2);
}
void balance_assign(struct node *root)
```

```
{
if (root != NULL)
{

root->balance_factor = findHeight(root->leftLink) - findHeight(root-
>rightLink);
balance_assign(root->leftLink);
balance_assign(root->rightLink);
}
}

void balance_factor_traversal(struct node *root)
{
if (root != NULL)
{
balance_factor_traversal(root->leftLink);
printf("%d --> %d \n", root->data, root->balance_factor);
balance_factor_traversal(root->rightLink);
}
}

int main()
{
printf("Enter the number of Case : ");
int temp;
scanf("%d", &temp);
while (temp-- > 0)
{
printf("Enter the size of an array : ");
int size;
scanf("%d", &size);
int *array = (int *)malloc(sizeof(int) * size);

printf("Enter the element of an array :");
for (int i = 0; i < size; i++)
{
scanf("%d", array + i);
}
struct node *root = NULL;
root = insert(root, *(array + 0));
for (int i = 1; i < size; i++)
{
```

```
insert(root, *(array + i));
}
printf("\n\n");
printf("The Inorder sequences is :\n");
inorder(root);
printf("\n\n");
printf("The Preorder sequences is :\n");
preorder(root);
printf("\n\n");
printf("The Postorder sequences is : \n");
postorder(root);
printf("\n\n");
printf("Enter the value to be searched : ");
int key;
scanf("%d", &key);

bool check = search(root, key);

if (!check)
{
printf("The Element you have entered that laready present in Array \n");
}
else
{
printf("Adding element ....To the tree :\n");
struct node *new = insert(root, key);
printf("The tree which will become inorder :\n");
inorder(new);
}
printf("\n\n");
balance_assign(root);
printf("The order of the balance factor is : \n");
balance_factor_traversal(root);
}
printf("\n\n");
return 0;
}
```

OUTPUT :




```c
/*2) Write a program to create the AVL tree by iterative insertion.*/

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

typedef struct node
{
int info;
struct node *left, *right;
} NODE;

struct Stack
{
int top;
unsigned capacity;
NODE **array;
};

struct Stack *createStack(unsigned capacity)
{
struct Stack *stack = (struct Stack *)malloc(sizeof(struct Stack));
```

```
stack->capacity = capacity;
stack->top = -1;
stack->array = (NODE **)malloc(stack->capacity * sizeof(NODE *));
return stack;
}

int isFull(struct Stack *stack)
{
return stack->top == stack->capacity - 1;
}

int isEmpty(struct Stack *stack)
{
return stack->top == -1;
}

void push(struct Stack *stack, NODE *item)
{
if (isFull(stack))
return;
stack->array[++stack->top] = item;
// printf("%d pushed to stack\n", item);
}

NODE *pop(struct Stack *stack)
{
if (isEmpty(stack))
return NULL;
return stack->array[stack->top--];
}

NODE *peek(struct Stack *stack)
{
if (isEmpty(stack))
return NULL;
return stack->array[stack->top];
}

int max(int x, int y)
{
return x > y ? x : y;
}
```

```
int height(NODE *root)
{
if (root == NULL)
return 0;
return 1 + max(height(root->left), height(root->right));
}

int getBalFactor(NODE *root)
{
return height(root->left) - height(root->right);
}

NODE *rightRotate(NODE *y)
{
NODE *x = y->left;
NODE *T2 = x->right;
x->right = y;
y->left = T2;
return x;
}

NODE *leftRotate(NODE *x)
{
NODE *y = x->right;
NODE *T2 = y->left;
y->left = x;
x->right = T2;
return y;
}

NODE *create(NODE *root, int x)
{
struct Stack *stack = createStack(100);
NODE *newnode = (NODE *)malloc(sizeof(NODE));
newnode->info = x;
newnode->right = NULL;
newnode->left = NULL;
NODE *curr = root;
NODE *trail = NULL;
while (curr != NULL)
{
```

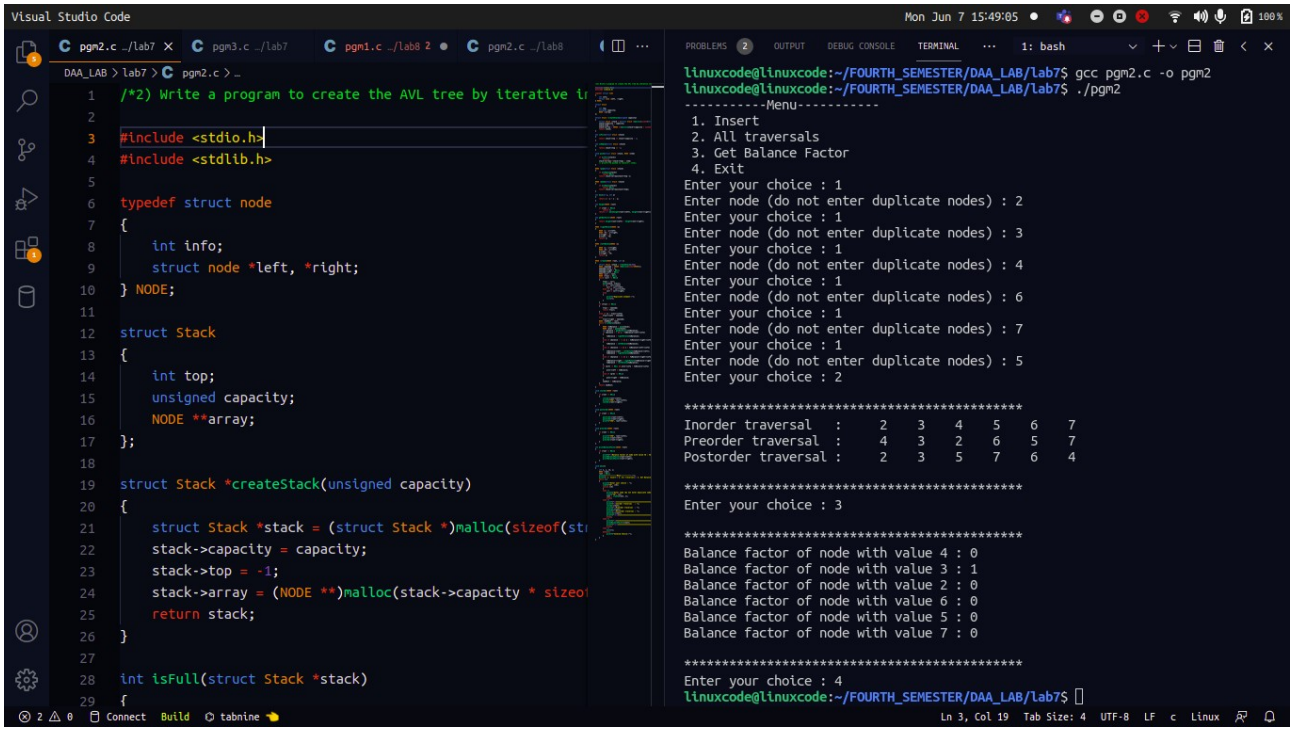
```
trail = curr;
push(stack, trail);
if (x < curr->info)
curr = curr->left;
else if (x > curr->info)
curr = curr->right;
else
{
printf("Duplicate element\n");
exit(0);
}
}
if (trail == NULL)
{
trail = newnode;
return trail;
}
else if (x < trail->info)
trail->left = newnode;
else
trail->right = newnode;
NODE *newRoot = root;
while (!isEmpty(stack))
{
NODE *toBalance = pop(stack);
NODE *prev = peek(stack);
int balance = getBalFactor(toBalance);
if (balance > 1 && x < toBalance->left->info)
{
toBalance = rightRotate(toBalance);
}
else if (balance < -1 && x > toBalance->right->info)
{
toBalance = leftRotate(toBalance);
}
else if (balance > 1 && x > toBalance->left->info)
{
toBalance->left = leftRotate(toBalance->left);
toBalance = rightRotate(toBalance);
}
else if (balance < -1 && x < toBalance->right->info)
{
```

```
toBalance->right = rightRotate(toBalance->right);
toBalance = leftRotate(toBalance);
}
if (prev != NULL && prev->info > toBalance->info)
{
prev->left = toBalance;
}
else if (prev != NULL)
{
prev->right = toBalance;
}
newRoot = toBalance;
}
return newRoot;
}

void inorder(NODE *root)
{
if (root != NULL)
{
inorder(root->left);
printf("%5d", root->info);
inorder(root->right);
}
}

void postorder(NODE *root)
{
if (root != NULL)
{
postorder(root->left);
postorder(root->right);
printf("%5d", root->info);
}
}

void preorder(NODE *root)
{
if (root != NULL)
{
printf("%5d", root->info);
preorder(root->left);
```

```
preorder(root->right);
}
}

int printBalanceFactor(NODE *root)
{
if (root != NULL)
{
printf("\nBalance factor of node with value %d : %d", root->info,
getBalFactor(root));
printBalanceFactor(root->left);
printBalanceFactor(root->right);
}
}

void main()
{
int n, x, ch, i;
NODE *root;
root = NULL;
printf("-----------Menu-----------\n");
printf(" 1. Insert\n 2. All traversals\n 3. Get Balance Factor\n 4. Exit\
n");
while (1)
{
printf("Enter your choice : ");
scanf("%d", &ch);
switch (ch)
{
case 1:
printf("Enter node (do not enter duplicate nodes) : ");
scanf("%d", &x);
root = create(root, x);
break;
case 2:
printf("\n*********************************************");
printf("\nInorder traversal : ");
inorder(root);
printf("\nPreorder traversal : ");
preorder(root);
printf("\nPostorder traversal : ");
postorder(root);
```

```
printf("\n\n*****************************************\n");
break;
case 3:
printf("\n*****************************************");
printBalanceFactor(root);
printf("\n\n*****************************************\n");
break;
case 4:
exit(0);
default:
printf("Invalid Choice\n");
}}}
```

OUTPUT :



ANALYSIS PART :

1.

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. So best case time case time complexity is O(logn) and worst case time complexity is O(n).

2.

The height of an AVL tree is always O(logn) where n is the number of nodes in the tree. Thus most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(logn) time.