**Muhammad Abdullah**

**SE(3A) | 19F-0916**

DS COURSE

ASSIGNMENT # 4

**QUESTION # 1:**

**PROGRAM:**

#include<iostream>

#include<climits>

using namespace std;

void swap(int \*x, int \*y)

{

int temp = \*x;

\*x = \*y;

\*y = temp;

}

class Heap

{

int \*Heap\_Array;

int capacity;

int heap\_size;

public:

Heap(int cap)

{

heap\_size = 0;

capacity = cap;

Heap\_Array = new int[cap];

}

void insertKey(int k)

{

if (heap\_size == capacity)

{

cout << "\nOverflow: Could not insertKey\n";

return;

}

heap\_size++;

int i = heap\_size - 1;

Heap\_Array[i] = k;

while (i != 0 && Heap\_Array[parent(i)] > Heap\_Array[i])

{

swap(&Heap\_Array[i], &Heap\_Array[parent(i)]);

i = parent(i);

}

}

void deleteKey(int i)

{

decreaseKey(i, INT\_MIN);

extractMin();

}

void Heapify(int i)

{

int l = left(i);

int r = right(i);

int smallest = i;

if (l < heap\_size && Heap\_Array[l] < Heap\_Array[i])

smallest = l;

if (r < heap\_size && Heap\_Array[r] < Heap\_Array[smallest])

smallest = r;

if (smallest != i)

{

swap(&Heap\_Array[i], &Heap\_Array[smallest]);

Heapify(smallest);

}

}

int parent(int i)

{

return (i - 1) / 2;

}

int left(int i)

{

return (2 \* i + 1);

}

int right(int i)

{

return (2 \* i + 2);

}

int extractMin()

{

if (heap\_size <= 0)

return INT\_MAX;

if (heap\_size == 1)

{

heap\_size--;

return Heap\_Array[0];

}

int root = Heap\_Array[0];

Heap\_Array[0] = Heap\_Array[heap\_size - 1];

heap\_size--;

Heapify(0);

return root;

}

void decreaseKey(int i, int new\_val)

{

Heap\_Array[i] = new\_val;

while (i != 0 && Heap\_Array[parent(i)] > Heap\_Array[i])

{

swap(&Heap\_Array[i], &Heap\_Array[parent(i)]);

i = parent(i);

}

}

int getMin()

{

return Heap\_Array[0];

}

void IN\_order(int AR[], int i)

{

while (AR[i] != 0000 )

{

cout << " " << AR[i] << " ";

i++;

}

}

};

int main()

{

int AR[18] = { 10,23,4,5,22,1,3,55,7,76,54,45,33,99,22,56,77,0000 };

Heap heap(18);

for (int i = 0; i < 18; i++)

{

heap.insertKey(AR[i]);

}

heap.IN\_order(AR, 0);

cout << endl << endl;

system("pause");

}

**A picture containing text, monitor, screenshot, indoor

Description automatically generated**

**QUESTION # 2:**

**PROGRAM:**

#include<iostream>

#include<climits>

using namespace std;

void swap(int \*x, int \*y)

{

int temp = \*x;

\*x = \*y;

\*y = temp;

}

class Heap

{

int \*Heap\_Array;

int capacity;

int heap\_size;

public:

Heap(int cap)

{

heap\_size = 0;

capacity = cap;

Heap\_Array = new int[cap];

}

void insertKey(int k)

{

if (heap\_size == capacity)

{

cout << "\nOverflow: Could not insertKey\n";

return;

}

heap\_size++;

int i = heap\_size - 1;

Heap\_Array[i] = k;

while (i != 0 && Heap\_Array[parent(i)] > Heap\_Array[i])

{

swap(&Heap\_Array[i], &Heap\_Array[parent(i)]);

i = parent(i);

}

}

void deleteKey(int i)

{

decreaseKey(i, INT\_MIN);

extractMin();

}

void Heapify(int i)

{

int l = left(i);

int r = right(i);

int smallest = i;

if (l < heap\_size && Heap\_Array[l] < Heap\_Array[i])

smallest = l;

if (r < heap\_size && Heap\_Array[r] < Heap\_Array[smallest])

smallest = r;

if (smallest != i)

{

swap(&Heap\_Array[i], &Heap\_Array[smallest]);

Heapify(smallest);

}

}

int parent(int i)

{

return (i - 1) / 2;

}

int left(int i)

{

return (2 \* i + 1);

}

int right(int i)

{

return (2 \* i + 2);

}

int extractMin()

{

if (heap\_size <= 0)

return INT\_MAX;

if (heap\_size == 1)

{

heap\_size--;

return Heap\_Array[0];

}

int root = Heap\_Array[0];

Heap\_Array[0] = Heap\_Array[heap\_size - 1];

heap\_size--;

Heapify(0);

return root;

}

void decreaseKey(int i, int new\_val)

{

Heap\_Array[i] = new\_val;

while (i != 0 && Heap\_Array[parent(i)] > Heap\_Array[i])

{

swap(&Heap\_Array[i], &Heap\_Array[parent(i)]);

i = parent(i);

}

}

int getMin()

{

return Heap\_Array[0];

}

void IN\_order(int AR[], int i)

{

while (AR[i] != 0000 )

{

cout << " " << AR[i] << " ";

i++;

}

}

};

void max\_heap(int \*a, int m, int n) {

int j, t;

t = a[m];

j = 2 \* m;

while (j <= n) {

if (j < n && a[j + 1] > a[j])

j = j + 1;

if (t > a[j])

break;

else if (t <= a[j]) {

a[j / 2] = a[j];

j = 2 \* j;

}

}

a[j / 2] = t;

return;

}

void build\_maxheap(int \*a, int n) {

int k;

for (k = n / 2; k >= 1; k--) {

max\_heap(a, k, n);

}

}

int main()

{

int AR[18] = { 10,23,4,5,22,1,3,55,7,76,54,45,33,99,22,56,77,0000 };

Heap heap(18);

for (int i = 0; i < 18; i++)

{

heap.insertKey(AR[i]);

}

cout << "Min Heap\n";

heap.IN\_order(AR, 0);

build\_maxheap(AR, 18);

cout << "\n\nMax Heap\n";

for (int i = 1; i < 18; i++) {

cout << " " << AR[i] << " ";

}

cout << endl << endl;

system("pause");

}

**A picture containing text, screenshot, electronics, computer

Description automatically generated**

**QUESTION # 3:**

**PROGRAM:**

#include <iostream>

using namespace std;

void MaxHeapify(int arr[], int i, int n)

{

int l = 2 \* i + 1;

int r = 2 \* i + 2;

int largest = i;

if (l < n && arr[l] > arr[i])

largest = l;

if (r < n && arr[r] > arr[largest])

largest = r;

if (largest != i)

{

swap(arr[i], arr[largest]);

MaxHeapify(arr, largest, n);

}

}

void convertMaxHeap(int arr[], int n)

{

for (int i = (n - 2) / 2; i >= 0; --i)

MaxHeapify(arr, i, n);

}

void printArray(int\* arr, int size)

{

for (int i = 0; i < size; ++i)

printf("%d ", arr[i]);

}

int main()

{

int arr[] = { 10,23,4,5,22,1,3,55,7,76,54,45,33,99,22,56,77. };

int n = sizeof(arr) / sizeof(arr[0]);

printf("Min Heap : ");

printArray(arr, n);

convertMaxHeap(arr, n);

printf("\nMax Heap : ");

printArray(arr, n);

cout << endl << endl;

system("pause");

}

**A picture containing text, monitor, screenshot, indoor

Description automatically generated**

**QUESTION # 4:**

**PROGRAM:**

#include <iostream>

using namespace std; // AVL Tree Formation

struct Node // Node Construction

{

int Data;

Node \*Left\_Node;

Node \*Right\_Node;

};

class AVL\_Tree // Class for Tree Formation

{

public:

AVL\_Tree()

{

Root = NULL;

}

Node \*Root;

Node \*Insertion(Node \*Current, int data) // Insertion of Nodes

{

if (Current == NULL)

{

Current = new Node;

Current->Data = data;

Current->Left\_Node = NULL;

Current->Right\_Node = NULL;

return Current;

}

else if (data < Current->Data) // If Data is lesser than Root

{

Current->Left\_Node = Insertion(Current->Left\_Node, data);

Current = Balance\_Tree(Current);

}

else if (data >= Current->Data) // If Data is greater than Root

{

Current->Right\_Node = Insertion(Current->Right\_Node, data);

Current = Balance\_Tree(Current);

}

return Current;

}

int Height\_of\_Tree(Node \*Current) // Checking Height of the Tree

{

int Height = 0;

if (Current != NULL)

{

int Left\_SubTree = Height\_of\_Tree(Current->Left\_Node);

int Right\_SubTree = Height\_of\_Tree(Current->Right\_Node);

if (Left\_SubTree >= Right\_SubTree)

return (Left\_SubTree + 1); // Returning Left Sub Tree if Greater

else

return (Right\_SubTree + 1); // Returning Right Sub Tree if Greater

}

else

return 0;

}

int Difference(Node \*Current) // Checking Difference of Left and Right Sub Tree

{

if (Root != NULL)

{

int Left\_SubTree = Height\_of\_Tree(Current->Left\_Node);

int Right\_SubTree = Height\_of\_Tree(Current->Right\_Node);

int Balance\_Factor = Left\_SubTree - Right\_SubTree;

return Balance\_Factor; // Returing BF

}

else

cout << endl << "Tree is Empty !" << endl;

}

Node \*Balance\_Tree(Node \*Current) // Checking Balance of the Tre

{

if (Root != NULL)

{

int Balance\_Factor = Difference(Current);

if (Balance\_Factor > 1)

{

if (Difference(Current->Left\_Node) > 0)

Current = LL\_Rotation(Current); // Criteria for Left Left Rotation

else

Current = LR\_Rotation(Current); // Criteria for Left Right Rotation

}

else if (Balance\_Factor < -1)

{

if (Difference(Current->Right\_Node) < 0)

Current = RR\_Rotation(Current); // Criteria for Right Right Rotation

else

Current = RL\_Rotation(Current); // Criteria for Right Left Rotation

}

}

else

cout << endl << "Tree is Empty !" << endl;

return Current;

}

void Pre\_Order(Node \*Current) // Pre-Order Traversal of the AVL Tree

{

if (Root != NULL)

{

if (Current == NULL)

return;

cout << Current->Data << " ";

Pre\_Order(Current->Left\_Node);

Pre\_Order(Current->Right\_Node);

}

else

cout << endl << "Tree is Empty !" << endl;

}

void In\_Order(Node \*Current) // In-Order Traversal of the AVL Tree

{

if (Root != NULL)

{

if (Current == NULL)

return;

In\_Order(Current->Left\_Node);

cout << Current->Data << " ";

In\_Order(Current->Right\_Node);

}

else

cout << endl << "Tree is Empty !" << endl;

}

void Post\_Order(Node \*Current) // Post-Order Traversal of the AVL Tree

{

if (Root != NULL)

{

if (Current == NULL)

return;

Post\_Order(Current->Left\_Node);

Post\_Order(Current->Right\_Node);

cout << Current->Data << " ";

}

else

cout << endl << "Tree is Empty !" << endl;

}

Node \*LL\_Rotation(Node \*Current) // Left Left Imbalance Rotation

{

Node \*Temp = NULL;

Temp = Current->Left\_Node;

Current->Left\_Node = Current->Right\_Node;

Current->Right\_Node = Current;

return Temp;

}

Node \*LR\_Rotation(Node \*Current) // Left Right Imbalance Rotation

{

Node \*Temp = NULL;

Temp = Current->Left\_Node;

Current->Left\_Node = RR\_Rotation(Current);

return LL\_Rotation(Current);

}

Node \*RR\_Rotation(Node \*Current) // Right Right Imbalance Rotation

{

Node \*Temp = NULL;

Temp = Current->Right\_Node;

Current->Right\_Node = Current->Left\_Node;

Current->Left\_Node = Current;

return Temp;

}

Node \*RL\_Rotation(Node \*Current) // Right Left Imbalance Rotation

{

Node \*Temp = NULL;

Temp = Current->Right\_Node;

Current->Right\_Node = LL\_Rotation(Current);

return RR\_Rotation(Current);

}

void Display\_AVL()

{

}

};

int main() // Main Code

{

AVL\_Tree AVL;

int Choice = 0, Data = 0, Check = 1;

while (Check == 1) // Menu Driven Code

{

system("cls");

cout << endl << " ================= AVL Tree Formation ==================" << endl;

cout << endl << " Press 1 to Make a Node in AVL Tree !" << endl << endl;

cout << " Press 2 to Traverse in Tree by In-Order Traversal !" << endl;

cout << " Press 3 to Traverse in Tree by Pre-Order Traversal !" << endl;

cout << " Press 4 to Traverse in Tree by Post-Order Traversal !" << endl << endl;

cout << " Press 0 to Exit from System !" << endl;

cout << " ========================================================" << endl;

cout << endl << " Enter Choice : ";

cin >> Choice;

switch (Choice)

{

case 1: // For Insertion

{

int AR[21] = { 1,2,3,6,4,8,6,9,7,11,15,18,12,17,22,21,24,29,28,28,27 };

for (int i = 0; i < 21; i++)

{

AVL.Root = AVL.Insertion(AVL.Root, AR[i]);

}

cout << endl << "Data has been added !" << endl;

cout << endl << endl;

system("pause");

break;

}

case 2: // For In-Order Traversal

{

cout << endl << " In-Order Traversal = ";

AVL.In\_Order(AVL.Root);

cout << endl << endl;

system("pause");

break;

}

case 3: // For Pre-Order Traversal

{

cout << endl << " Pre-Order Traversal = ";

AVL.Pre\_Order(AVL.Root);

cout << endl << endl;

system("pause");

break;

}

case 4: // For Post-Order Traversal

{

cout << endl << " Post-Order Traversal = ";

AVL.Post\_Order(AVL.Root);

cout << endl << endl;

system("pause");

break;

}

case 5: // Displaying Tree

{

cout << endl;

AVL.Display\_AVL();

cout << endl << endl;

system("pause");

break;

}

case 0: // Exiting from Loop

{

cout << endl << "You Have Exited From The System !" << endl;

Check = 0;

break;

}

default:

{

cout << endl << "Invalid Entry !" << endl;

break;

}

}

}

cout << endl << endl;

system("pause");

}

**A picture containing text, monitor, screenshot, indoor

Description automatically generated**

**QUESTION # 5: (SAME FOR THIS ONE TOO)**

**PROGRAM:**

#include <iostream>

using namespace std; // AVL Tree Formation

struct Node // Node Construction

{

int Data;

Node \*Left\_Node;

Node \*Right\_Node;

};

class AVL\_Tree // Class for Tree Formation

{

public:

AVL\_Tree()

{

Root = NULL;

}

Node \*Root;

Node \*Insertion(Node \*Current, int data) // Insertion of Nodes

{

if (Current == NULL)

{

Current = new Node;

Current->Data = data;

Current->Left\_Node = NULL;

Current->Right\_Node = NULL;

return Current;

}

else if (data < Current->Data) // If Data is lesser than Root

{

Current->Left\_Node = Insertion(Current->Left\_Node, data);

Current = Balance\_Tree(Current);

}

else if (data >= Current->Data) // If Data is greater than Root

{

Current->Right\_Node = Insertion(Current->Right\_Node, data);

Current = Balance\_Tree(Current);

}

return Current;

}

int Height\_of\_Tree(Node \*Current) // Checking Height of the Tree

{

int Height = 0;

if (Current != NULL)

{

int Left\_SubTree = Height\_of\_Tree(Current->Left\_Node);

int Right\_SubTree = Height\_of\_Tree(Current->Right\_Node);

if (Left\_SubTree >= Right\_SubTree)

return (Left\_SubTree + 1); // Returning Left Sub Tree if Greater

else

return (Right\_SubTree + 1); // Returning Right Sub Tree if Greater

}

else

return 0;

}

int Difference(Node \*Current) // Checking Difference of Left and Right Sub Tree

{

if (Root != NULL)

{

int Left\_SubTree = Height\_of\_Tree(Current->Left\_Node);

int Right\_SubTree = Height\_of\_Tree(Current->Right\_Node);

int Balance\_Factor = Left\_SubTree - Right\_SubTree;

return Balance\_Factor; // Returing BF

}

else

cout << endl << "Tree is Empty !" << endl;

}

Node \*Balance\_Tree(Node \*Current) // Checking Balance of the Tre

{

if (Root != NULL)

{

int Balance\_Factor = Difference(Current);

if (Balance\_Factor > 1)

{

if (Difference(Current->Left\_Node) > 0)

Current = LL\_Rotation(Current); // Criteria for Left Left Rotation

else

Current = LR\_Rotation(Current); // Criteria for Left Right Rotation

}

else if (Balance\_Factor < -1)

{

if (Difference(Current->Right\_Node) < 0)

Current = RR\_Rotation(Current); // Criteria for Right Right Rotation

else

Current = RL\_Rotation(Current); // Criteria for Right Left Rotation

}

}

else

cout << endl << "Tree is Empty !" << endl;

return Current;

}

void Pre\_Order(Node \*Current) // Pre-Order Traversal of the AVL Tree

{

if (Root != NULL)

{

if (Current == NULL)

return;

cout << Current->Data << " ";

Pre\_Order(Current->Left\_Node);

Pre\_Order(Current->Right\_Node);

}

else

cout << endl << "Tree is Empty !" << endl;

}

void In\_Order(Node \*Current) // In-Order Traversal of the AVL Tree

{

if (Root != NULL)

{

if (Current == NULL)

return;

In\_Order(Current->Left\_Node);

cout << Current->Data << " ";

In\_Order(Current->Right\_Node);

}

else

cout << endl << "Tree is Empty !" << endl;

}

void Post\_Order(Node \*Current) // Post-Order Traversal of the AVL Tree

{

if (Root != NULL)

{

if (Current == NULL)

return;

Post\_Order(Current->Left\_Node);

Post\_Order(Current->Right\_Node);

cout << Current->Data << " ";

}

else

cout << endl << "Tree is Empty !" << endl;

}

Node \*LL\_Rotation(Node \*Current) // Left Left Imbalance Rotation

{

Node \*Temp = NULL;

Temp = Current->Left\_Node;

Current->Left\_Node = Current->Right\_Node;

Current->Right\_Node = Current;

return Temp;

}

Node \*LR\_Rotation(Node \*Current) // Left Right Imbalance Rotation

{

Node \*Temp = NULL;

Temp = Current->Left\_Node;

Current->Left\_Node = RR\_Rotation(Current);

return LL\_Rotation(Current);

}

Node \*RR\_Rotation(Node \*Current) // Right Right Imbalance Rotation

{

Node \*Temp = NULL;

Temp = Current->Right\_Node;

Current->Right\_Node = Current->Left\_Node;

Current->Left\_Node = Current;

return Temp;

}

Node \*RL\_Rotation(Node \*Current) // Right Left Imbalance Rotation

{

Node \*Temp = NULL;

Temp = Current->Right\_Node;

Current->Right\_Node = LL\_Rotation(Current);

return RR\_Rotation(Current);

}

void Display\_AVL()

{

}

};

int main() // Main Code

{

AVL\_Tree AVL;

int Choice = 0, Data = 0, Check = 1;

while (Check == 1) // Menu Driven Code

{

system("cls");

cout << endl << " ================= AVL Tree Formation ==================" << endl;

cout << endl << " Press 1 to Make a Node in AVL Tree !" << endl << endl;

cout << " Press 2 to Traverse in Tree by In-Order Traversal !" << endl;

cout << " Press 3 to Traverse in Tree by Pre-Order Traversal !" << endl;

cout << " Press 4 to Traverse in Tree by Post-Order Traversal !" << endl << endl;

cout << " Press 0 to Exit from System !" << endl;

cout << " ========================================================" << endl;

cout << endl << " Enter Choice : ";

cin >> Choice;

switch (Choice)

{

case 1: // For Insertion

{

int AR[21] = { 1,2,3,6,4,8,6,9,7,11,15,18,12,17,22,21,24,29,28,28,27 };

for (int i = 0; i < 21; i++)

{

AVL.Root = AVL.Insertion(AVL.Root, AR[i]);

}

cout << endl << "Data has been added !" << endl;

cout << endl << endl;

system("pause");

break;

}

case 2: // For In-Order Traversal

{

cout << endl << " In-Order Traversal = ";

AVL.In\_Order(AVL.Root);

cout << endl << endl;

system("pause");

break;

}

case 3: // For Pre-Order Traversal

{

cout << endl << " Pre-Order Traversal = ";

AVL.Pre\_Order(AVL.Root);

cout << endl << endl;

system("pause");

break;

}

case 4: // For Post-Order Traversal

{

cout << endl << " Post-Order Traversal = ";

AVL.Post\_Order(AVL.Root);

cout << endl << endl;

system("pause");

break;

}

case 5: // Displaying Tree

{

cout << endl;

AVL.Display\_AVL();

cout << endl << endl;

system("pause");

break;

}

case 0: // Exiting from Loop

{

cout << endl << "You Have Exited From The System !" << endl;

Check = 0;

break;

}

default:

{

cout << endl << "Invalid Entry !" << endl;

break;

}

}

}

cout << endl << endl;

system("pause");

}

**A picture containing text, monitor, screenshot, indoor

Description automatically generated**

**QUESTION # 6:**

**PROGRAM:**

#include <iostream>

using namespace std;

struct Node // Node Construction

{

int Data;

Node \*Left\_Node;

Node \*Right\_Node;

};

class Bst\_Tree // Class for Tree Formation

{

public:

Bst\_Tree()

{

Root = NULL;

}

Node \*Root;

Node \*Insertion(Node \*Current, int data) // Insertion of Nodes

{

if (Current == NULL)

{

Current = new Node;

Current->Data = data;

Current->Left\_Node = NULL;

Current->Right\_Node = NULL;

return Current;

}

else if (data < Current->Data) // If Data is lesser than Root

{

Current->Left\_Node = Insertion(Current->Left\_Node, data);

}

else if (data >= Current->Data) // If Data is greater than Root

{

Current->Right\_Node = Insertion(Current->Right\_Node, data);

}

return Current;

}

int Count\_Leaf(Node \*Temp) // Counting Leaf Nodes

{

if (Temp == NULL)

return 0;

if (Temp->Left\_Node == NULL && Temp->Right\_Node == NULL)

return 1;

else

return Count\_Leaf(Temp->Left\_Node) + Count\_Leaf(Temp->Right\_Node);

}

};

int main()

{

Bst\_Tree Tree;

int choice = 0, value = 0, i = 1;

while (i != 0)

{

system("cls");

cout << "=================================================" << endl;

cout << " Press 1 to Enter a Node !" << endl;

cout << " Press 2 to Check Total Numbers of Leaf Nodes !" << endl;

cout << " Press 0 To Exit From The System !" << endl;

cout << "=================================================" << endl;

cout << " Enter Choice : ";

cin >> choice;

switch (choice)

{

case 1:

{

cout << endl << "Enter Value for the Node :";

cin >> value;

Tree.Root = Tree.Insertion(Tree.Root, value);

system("pause");

break;

}

case 2:

{

cout << endl;

value = 0;

value = Tree.Count\_Leaf(Tree.Root);

cout << "Total Number of Leaf Nodes Are : " << value << endl;

system("pause");

break;

}

case 0:

{

cout << endl << "You have exited from the system !" << endl;

i = 0;

break;

}

default:

cout << endl << "Invalid Entry !" << endl;

}

}

cout << endl << endl;

system("pause");

}

**A picture containing text, monitor, indoor, electronics

Description automatically generated**

**QUESTION # 7:**

**PROGRAM:**

#include <iostream>

#include<climits>

using namespace std; // AVL + HEAP Tree Formation

void swap(int \*x, int \*y)

{

int temp = \*x;

\*x = \*y;

\*y = temp;

}

class Heap

{

int \*Heap\_Array;

int capacity;

int heap\_size;

public:

Heap(int cap)

{

heap\_size = 0;

capacity = cap;

Heap\_Array = new int[cap];

}

void insertKey(int k)

{

if (heap\_size == capacity)

{

cout << "\nOverflow: Could not insertKey\n";

return;

}

heap\_size++;

int i = heap\_size - 1;

Heap\_Array[i] = k;

while (i != 0 && Heap\_Array[parent(i)] > Heap\_Array[i])

{

swap(&Heap\_Array[i], &Heap\_Array[parent(i)]);

i = parent(i);

}

}

void deleteKey(int i)

{

decreaseKey(i, INT\_MIN);

extractMin();

}

void Heapify(int i)

{

int l = left(i);

int r = right(i);

int smallest = i;

if (l < heap\_size && Heap\_Array[l] < Heap\_Array[i])

smallest = l;

if (r < heap\_size && Heap\_Array[r] < Heap\_Array[smallest])

smallest = r;

if (smallest != i)

{

swap(&Heap\_Array[i], &Heap\_Array[smallest]);

Heapify(smallest);

}

}

int parent(int i)

{

return (i - 1) / 2;

}

int left(int i)

{

return (2 \* i + 1);

}

int right(int i)

{

return (2 \* i + 2);

}

int extractMin()

{

if (heap\_size <= 0)

return INT\_MAX;

if (heap\_size == 1)

{

heap\_size--;

return Heap\_Array[0];

}

int root = Heap\_Array[0];

Heap\_Array[0] = Heap\_Array[heap\_size - 1];

heap\_size--;

Heapify(0);

return root;

}

void decreaseKey(int i, int new\_val)

{

Heap\_Array[i] = new\_val;

while (i != 0 && Heap\_Array[parent(i)] > Heap\_Array[i])

{

swap(&Heap\_Array[i], &Heap\_Array[parent(i)]);

i = parent(i);

}

}

int getMin()

{

return Heap\_Array[0];

}

void IN\_order(int AR[], int i)

{

while (AR[i] != 0000)

{

cout << " " << AR[i] << " ";

i++;

}

}

};

struct Node // Node Construction

{

int Data;

Node \*Left\_Node;

Node \*Right\_Node;

};

class AVL\_Tree // Class for Tree Formation

{

public:

AVL\_Tree()

{

Root = NULL;

}

Node \*Root;

Node \*Insertion(Node \*Current, int data) // Insertion of Nodes

{

if (Current == NULL)

{

Current = new Node;

Current->Data = data;

Current->Left\_Node = NULL;

Current->Right\_Node = NULL;

return Current;

}

else if (data < Current->Data) // If Data is lesser than Root

{

Current->Left\_Node = Insertion(Current->Left\_Node, data);

Current = Balance\_Tree(Current);

}

else if (data >= Current->Data) // If Data is greater than Root

{

Current->Right\_Node = Insertion(Current->Right\_Node, data);

Current = Balance\_Tree(Current);

}

return Current;

}

int Height\_of\_Tree(Node \*Current) // Checking Height of the Tree

{

int Height = 0;

if (Current != NULL)

{

int Left\_SubTree = Height\_of\_Tree(Current->Left\_Node);

int Right\_SubTree = Height\_of\_Tree(Current->Right\_Node);

if (Left\_SubTree >= Right\_SubTree)

return (Left\_SubTree + 1); // Returning Left Sub Tree if Greater

else

return (Right\_SubTree + 1); // Returning Right Sub Tree if Greater

}

else

return 0;

}

int Difference(Node \*Current) // Checking Difference of Left and Right Sub Tree

{

if (Root != NULL)

{

int Left\_SubTree = Height\_of\_Tree(Current->Left\_Node);

int Right\_SubTree = Height\_of\_Tree(Current->Right\_Node);

int Balance\_Factor = Left\_SubTree - Right\_SubTree;

return Balance\_Factor; // Returing BF

}

else

cout << endl << "Tree is Empty !" << endl;

}

Node \*Balance\_Tree(Node \*Current) // Checking Balance of the Tre

{

if (Root != NULL)

{

int Balance\_Factor = Difference(Current);

if (Balance\_Factor > 1)

{

if (Difference(Current->Left\_Node) > 0)

Current = LL\_Rotation(Current); // Criteria for Left Left Rotation

else

Current = LR\_Rotation(Current); // Criteria for Left Right Rotation

}

else if (Balance\_Factor < -1)

{

if (Difference(Current->Right\_Node) < 0)

Current = RR\_Rotation(Current); // Criteria for Right Right Rotation

else

Current = RL\_Rotation(Current); // Criteria for Right Left Rotation

}

}

else

cout << endl << "Tree is Empty !" << endl;

return Current;

}

void Pre\_Order(Node \*Current) // Pre-Order Traversal of the AVL Tree

{

if (Root != NULL)

{

if (Current == NULL)

return;

cout << Current->Data << " ";

Pre\_Order(Current->Left\_Node);

Pre\_Order(Current->Right\_Node);

}

else

cout << endl << "Tree is Empty !" << endl;

}

void In\_Order(Node \*Current) // In-Order Traversal of the AVL Tree

{

if (Root != NULL)

{

if (Current == NULL)

return;

In\_Order(Current->Left\_Node);

cout << Current->Data << " ";

In\_Order(Current->Right\_Node);

}

else

cout << endl << "Tree is Empty !" << endl;

}

void Post\_Order(Node \*Current) // Post-Order Traversal of the AVL Tree

{

if (Root != NULL)

{

if (Current == NULL)

return;

Post\_Order(Current->Left\_Node);

Post\_Order(Current->Right\_Node);

cout << Current->Data << " ";

}

else

cout << endl << "Tree is Empty !" << endl;

}

Node \*LL\_Rotation(Node \*Current) // Left Left Imbalance Rotation

{

Node \*Temp = NULL;

Temp = Current->Left\_Node;

Current->Left\_Node = Current->Right\_Node;

Current->Right\_Node = Current;

return Temp;

}

Node \*LR\_Rotation(Node \*Current) // Left Right Imbalance Rotation

{

Node \*Temp = NULL;

Temp = Current->Left\_Node;

Current->Left\_Node = RR\_Rotation(Current);

return LL\_Rotation(Current);

}

Node \*RR\_Rotation(Node \*Current) // Right Right Imbalance Rotation

{

Node \*Temp = NULL;

Temp = Current->Right\_Node;

Current->Right\_Node = Current->Left\_Node;

Current->Left\_Node = Current;

return Temp;

}

Node \*RL\_Rotation(Node \*Current) // Right Left Imbalance Rotation

{

Node \*Temp = NULL;

Temp = Current->Right\_Node;

Current->Right\_Node = LL\_Rotation(Current);

return RR\_Rotation(Current);

}

void Display\_AVL()

{

}

};

int main() // Main Code

{

AVL\_Tree AVL;

int Choice = 0, Data = 0, Check = 1;

while (Check == 1) // Menu Driven Code

{

system("cls");

cout << endl << " ================= AVL Tree Formation ==================" << endl;

cout << endl << " Press 1 to Make a Node in AVL Tree !" << endl << endl;

cout << " Press 2 to Traverse in Tree by In-Order Traversal !" << endl;

cout << " Press 3 to Traverse in Tree by Pre-Order Traversal !" << endl;

cout << " Press 4 to Traverse in Tree by Post-Order Traversal !" << endl << endl;

cout << " Press 0 to Exit from System !" << endl;

cout << " ========================================================" << endl;

cout << endl << " Enter Choice : ";

cin >> Choice;

switch (Choice)

{

case 1: // For Insertion

{

cout << endl << "Enter Data to Push it in AVL Tree :";

cin >> Data;

AVL.Root = AVL.Insertion(AVL.Root, Data);

cout << endl << endl;

system("pause");

break;

}

case 2: // For In-Order Traversal

{

cout << endl << " In-Order Traversal = ";

AVL.In\_Order(AVL.Root);

cout << endl << endl;

system("pause");

break;

}

case 3: // For Pre-Order Traversal

{

cout << endl << " Pre-Order Traversal = ";

AVL.Pre\_Order(AVL.Root);

cout << endl << endl;

system("pause");

break;

}

case 4: // For Post-Order Traversal

{

cout << endl << " Post-Order Traversal = ";

AVL.Post\_Order(AVL.Root);

cout << endl << endl;

system("pause");

break;

}

case 5: // Displaying Tree

{

cout << endl;

AVL.Display\_AVL();

cout << endl << endl;

system("pause");

break;

}

case 0: // Exiting from Loop

{

cout << endl << "You Have Exited From The System !" << endl;

Check = 0;

break;

}

default:

{

cout << endl << "Invalid Entry !" << endl;

break;

}

}

}

cout << endl << endl;

system("pause");

}