

Student Information

- Name: [ATUL KUMAR]
- Student ID: [590011190]
- Branch: [MCA]
- Batch: [B1]
- Instructor: [Dr. Sourbh Kumar]
- Date: [07/11/2024]

Lab Experiment: 10

Subject: Data Structures Lab

Semester: 1st

Batch: 1 & 2

MCA

Assignment 1st: AVL Tree Implementation

Definition: An AVL Tree is a self-balancing binary search tree. For any node in the tree, the height difference between its left and right subtrees is at most one.

Tasks:

- Implement insertion in an AVL tree. Ensure that after each insertion, the tree remains balanced using rotations (left rotation, right rotation, left-right rotation, right-left rotation).
- Implement deletion in an AVL tree with the necessary rebalancing steps.

Testing: Insert and delete a series of values, displaying the tree structure after each operation.

Solution:--

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

// Node structure for AVL Tree
typedef struct Node {
    int key;
    struct Node* left;
    struct Node* right;
    int height;
} Node;

// Utility function to get the height of a node
int height(Node* node) {
    return node ? node->height : 0;
}

// Utility function to get the balance factor of a node
int getBalance(Node* node) {
    return node ? height(node->left) - height(node->right) : 0;
}

// Function to create a new node
Node* createNode(int key) {
```

```

Node* node = (Node*)malloc(sizeof(Node));
node->key = key;
node->left = node->right = NULL;
node->height = 1;
return node;
}

```

// Right rotation

```

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

```

```

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

```

```

y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));
x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));

```

```

return x;
}

```

// Left rotation

```

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

```

```

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

```

```

x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));
y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));

```

```

return y;
}

```

// Insert a key and perform rotations to balance the tree

```

Node* insert(Node* node, int key) {
if (!node) return createNode(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 + (height(node->left) > height(node->right) ? 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;
}

// Function to find the node with the smallest value
Node* minValueNode(Node* node) {
Node* current = node;
while (current->left)
current = current->left;
return current;
}

// Delete a node and balance the tree
Node* deleteNode(Node* root, int key) {
if (!root) 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 || !root->right) {
Node* temp = root->left ? root->left : root->right;
free(root);
root = temp;
} else {

```

```

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

if (!root) return root;

root->height = 1 + (height(root->left) > height(root->right) ?
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;
}

// Utility function to print the tree (in-order traversal)
void printInOrder(Node* root) {
if (root) {
printInOrder(root->left);
printf("%d ", root->key);
printInOrder(root->right);
}
}

int main() {
Node* root = NULL;

// Test Insertion
root = insert(root, 10);
root = insert(root, 20);
root = insert(root, 30);
root = insert(root, 40);

```

```

root = insert(root, 50);
root = insert(root, 25);

printf("In-Order traversal after insertions: ");
printInOrder(root);
printf("\n");

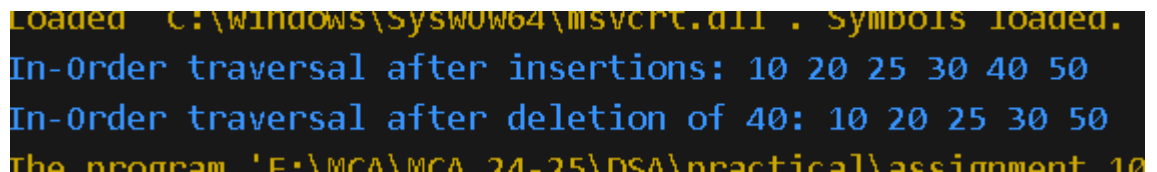
// Test Deletion
root = deleteNode(root, 40);

printf("In-Order traversal after deletion of 40: ");
printInOrder(root);
printf("\n");

return 0;
}

```

Output:--



```

Loaded C:\Windows\SysWow64\msvcrt.dll . Symbols loaded.
In-Order traversal after insertions: 10 20 25 30 40 50
In-Order traversal after deletion of 40: 10 20 25 30 50
The program 'E:\MCA\MCA_24-25\DSA\practical\assignment 10

```

Assignment 2nd: Heap Sort Implementation

Definition: Heap sort is a comparison-based sorting algorithm that uses a binary heap (typically a max-heap).

Tasks:

- Build a max heap from an array of unsorted elements.
- Implement heap sort by repeatedly removing the root element (maximum value) and re-heapifying the tree.

Testing: Demonstrate heap sort with an example array, showing each step and the final sorted output.

Solution:--

```

#include <stdio.h>

// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to "heapify" a subtree rooted at index `i` in an array of size `n`
void heapify(int arr[], int n, int i) {
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child

    // Check if left child exists and is greater than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

```

```

// Check if right child exists and is greater than the largest so far
if (right < n && arr[right] > arr[largest])
largest = right;

// If the largest is not root
if (largest != i) {
swap(&arr[i], &arr[largest]); // Swap root with the largest child
heapify(arr, n, largest);      // Recursively heapify the affected subtree
}
}

// Function to perform heap sort
void heapSort(int arr[], int n) {
// Build a max heap
for (int i = n / 2 - 1; i >= 0; i--)
heapify(arr, n, i);

// One by one extract elements from heap
for (int i = n - 1; i > 0; i--) {
// Move the current root to the end
swap(&arr[0], &arr[i]);

// Call heapify on the reduced heap
heapify(arr, i, 0);
}
}

// Function to print the array
void printArray(int arr[], int n) {
for (int i = 0; i < n; i++)
printf("%d ", arr[i]);
printf("\n");
}

int main() {
int arr[] = {12, 11, 13, 5, 6, 7};
int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array: ");
printArray(arr, n);

heapSort(arr, n);

printf("Sorted array: ");
printArray(arr, n);

return 0;
}

```

Output:--

```

Original array: 12 11 13 5 6 7
Array after moving max element to position 5: 7 11 12 5 6 13
Array after moving max element to position 4: 6 11 7 5 12 13
Array after moving max element to position 3: 5 6 7 11 12 13
Array after moving max element to position 2: 5 6 7 11 12 13
Array after moving max element to position 1: 5 6 7 11 12 13
Array after moving max element to position 0: 5 6 7 11 12 13
Sorted array: 5 6 7 11 12 13

```

Assignment 3rd: Priority Queue Using Heap

Definition: A priority queue is a data structure that allows elements to be removed based on priority (highest or lowest priority element is removed first).

Tasks:

- Implement a priority queue using a heap structure.
- Implement functions to insert elements with a priority and to remove the highest priority element.

Testing: Insert elements with varying priorities and demonstrate removing elements in priority order.

Solution:--

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

#define MAX_SIZE 100

// Structure to represent a priority queue using a max-heap
typedef struct {
    int data[MAX_SIZE]; // Array to store elements
    int size;           // Current number of elements in the heap
} PriorityQueue;

// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to insert an element into the priority queue
void insert(PriorityQueue* pq, int value) {
    if (pq->size >= MAX_SIZE) {
        printf("Priority queue overflow!\n");
        return;
    }

    // Insert the new element at the end
    int i = pq->size;
    pq->data[i] = value;
    pq->size++;

    // Move the element up to maintain the max-heap property
    while (i != 0 && pq->data[(i - 1) / 2] < pq->data[i]) {
        swap(&pq->data[i], &pq->data[(i - 1) / 2]);
        i = (i - 1) / 2;
    }

    printf("Inserted %d into the priority queue.\n", value);
}
```

```

// Function to heapify a subtree with root at index i
void heapify(PriorityQueue* pq, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < pq->size && pq->data[left] > pq->data[largest])
        largest = left;

    if (right < pq->size && pq->data[right] > pq->data[largest])
        largest = right;

    if (largest != i) {
        swap(&pq->data[i], &pq->data[largest]);
        heapify(pq, largest);
    }
}

// Function to remove the highest priority element (maximum
// value)
int removeMax(PriorityQueue* pq) {
    if (pq->size <= 0) {
        printf("Priority queue underflow!\n");
        return -1;
    }

    if (pq->size == 1) {
        pq->size--;
        return pq->data[0];
    }

    // Store the maximum value and remove it from the heap
    int root = pq->data[0];
    pq->data[0] = pq->data[pq->size - 1];
    pq->size--;

    // Heapify the root to maintain max-heap property
    heapify(pq, 0);

    return root;
}

// Function to print the priority queue
void printQueue(PriorityQueue* pq) {
    for (int i = 0; i < pq->size; i++) {
        printf("%d ", pq->data[i]);
    }
    printf("\n");
}

```



```

int main() {
    PriorityQueue pq;
    pq.size = 0;

    // Insert elements with varying priorities
    insert(&pq, 10);
    insert(&pq, 20);
    insert(&pq, 15);
    insert(&pq, 30);
    insert(&pq, 25);

    printf("Priority queue after insertions: ");
    printQueue(&pq);

    // Remove elements in priority order
    printf("Removed element with highest priority: %d\n",
        removeMax(&pq));
    printf("Priority queue after removing max: ");
    printQueue(&pq);

    printf("Removed element with highest priority: %d\n",
        removeMax(&pq));
    printf("Priority queue after removing max: ");
    printQueue(&pq);

    return 0;
}

```

Output:--

```

Inserted 10 into the priority queue.
Inserted 20 into the priority queue.
Inserted 15 into the priority queue.
Inserted 30 into the priority queue.
Inserted 25 into the priority queue.
Priority queue after insertions: 30 25 15 10 20
Removed element with highest priority: 30
Priority queue after removing max: 25 20 15 10
Removed element with highest priority: 25
Priority queue after removing max: 20 10 15

```

Τηανκυ ψου☐.....