1. AVL Tree Insertion

main.c

Share Run

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

struct Node {
    int data;
    struct Node *left, *right;
    int height;
};

// Function to get the height of a node
int height(struct Node* node) {
    if (node == NULL) return 0;
    return node->height;
}

// Function to get the balance factor of a node
int balanceFactor(struct Node* node) {
    if (node == NULL) return 0;
    return height(node->left) - height(node->right);
}

// Right rotation (used for left-left case)
struct Node* rightRotate(struct Node* y) {
    struct Node* x = y->left;
    struct 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 (used for right-right case)
struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct 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));
```

Output

```
/tmp/WOSM7Zz8OH.o
In-order Traversal: 10 20 30


=== Code Execution Successful ===
```

main.c

Share Run

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

    return y;
}

// Left-Right rotation (used for left-right case)
struct Node* leftRightRotate(struct Node* node) {
    node->left = leftRotate(node->left);
    return rightRotate(node);
}

// Right-Left rotation (used for right-left case)
struct Node* rightLeftRotate(struct Node* node) {
    node->right = rightRotate(node->right);
    return leftRotate(node);
}

// Insert a node and balance the tree
struct Node* insert(struct Node* node, int data) {
    if (node == NULL) {
        struct Node* newNode = malloc(sizeof(struct Node));
        newNode->data = data;
        newNode->left = newNode->right = NULL;
        newNode->height = 1;
        return newNode;
    }

    if (data < node->data) {
        node->left = insert(node->left, data);
    } else if (data > node->data) {
        node->right = insert(node->right, data);
    } else {
        return node;
    }

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

    int balance = balanceFactor(node);

    if (balance > 1 && data < node->left->data) {
        return rightRotate(node);
    }
    if (balance < -1 && data > node->right->data) {
```

Output

```
/tmp/WOSM7Zz8OH.o
In-order Traversal: 10 20 30


=== Code Execution Successful ===
```

main.c

Share Run

```c
    int balance = balanceFactor(node);

    if (balance > 1 && data < node->left->data) {
        return rightRotate(node);
    }
    if (balance < -1 && data > node->right->data) {
        return leftRotate(node);
    }
    if (balance > 1 && data > node->left->data) {
        return leftRightRotate(node);
    }
    if (balance < -1 && data < node->right->data) {
        return rightLeftRotate(node);
    }

    return node;
}

// Print the tree (In-order Traversal)
void inOrder(struct Node* root) {
    if (root != NULL) {
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);
    }
}

// Main function
int main() {
    struct Node* root = NULL;

    // Inserting nodes
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30); // Causes Left-Left case (rotation)

    printf("In-order Traversal: ");
    inOrder(root);  // Should print the nodes in sorted order
    printf("\n");

    return 0;
}
```

Output

```
/tmp/WOSM7Zz8OH.o
In-order Traversal: 10 20 30


=== Code Execution Successful ===
```

1. AVL Tree Deletion

main.c

Share Run

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

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

// Get height of the node
int height(struct Node *N) {
    if (N == NULL) return 0;
    return N->height;
}

// Get maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Right rotate subtree rooted with y
struct Node* rightRotate(struct Node *y) {
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

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

    // Return new root
    return x;
}

// Left rotate subtree rooted with x
struct Node* leftRotate(struct Node *x) {
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    // Perform rotation
```

Output

```
/tmp/t3ophXwTZn.o
Preorder traversal before deletion:
10 20 5 25 30
Preorder traversal after deletion:
10 20 5 25 30


=== Code Execution Successful ===
```

main.c

Share Run

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

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

    // Return new root
    return y;
}

// Get balance factor of node N
int getBalance(struct Node *N) {
    if (N == NULL) return 0;
    return height(N->left) - height(N->right);
}

// Recursive function to delete a node
struct Node* deleteNode(struct Node* root, int key) {
    // Step 1: Perform normal BST delete
    if (root == NULL) return root;

    // If key to be deleted is smaller than the root's key, then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If key to be deleted is greater than the root's key, then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // If key is the same as root's key, then this is the node to be deleted
    else {
        // Node with only one child or no child
        if ((root->left == NULL) || (root->right == NULL)) {
            struct Node *temp = root->left ? root->left : root->right;

            // No child case
            if (temp == NULL) {
                temp = root;
                root = NULL;
            } else // One child case
                *root = *temp;

            free(temp);
```

Output

```
/tmp/t3ophXwTZn.o
Preorder traversal before deletion:
10 20 5 25 30
Preorder traversal after deletion:
10 20 5 25 30


=== Code Execution Successful ===
```

main.c

Share Run

```
            free(temp);
        } else {
            // Node with two children: Get the inorder successor (smallest in the right subtree
            )
            struct Node* temp = root->right;
            while (temp->left != NULL) {
                temp = temp->left;
            }

            // Copy the inorder successor's content to this node
            root->key = temp->key;

            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->key);
        }
    }

    // If the tree had only one node, return
    if (root == NULL) return root;

    // Step 2: Update height of this ancestor node
    root->height = max(height(root->left), height(root->right)) + 1;

    // Step 3: Get the balance factor of this node (to check whether it became unbalanced)
    int balance = getBalance(root);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < root->left->key)
        return rightRotate(root);

    // Right Right Case
    if (balance < -1 && key > root->right->key)
        return leftRotate(root);

    // Left Right Case
    if (balance > 1 && key > root->left->key) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Left Case
    if (balance < -1 && key < root->right->key) {
```

Output

/tmp/t3ophXwTZn.o
Preorder traversal before deletion:
10 20 5 25 30
Preorder traversal after deletion:
10 20 5 25 30

=== Code Execution Successful ===

main.c

Share Run

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

    return root;
}

// A utility function to print preorder traversal of the tree
void preOrder(struct Node *root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Helper function to create a new node
struct Node* newNode(int key) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
    node->left = node->right = NULL;
    node->height = 1; // New node is initially at height 1
    return node;
}

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

    // Insert nodes
    root = newNode(10);
    root->left = newNode(20);
    root->right = newNode(30);
    root->left->left = newNode(5);
    root->left->right = newNode(25);

    printf("Preorder traversal before deletion:\n");
    preOrder(root);
    printf("\n");

    // Delete a node
    root = deleteNode(root, 20);

    printf("Preorder traversal after deletion:\n");
```

Output

```
/tmp/t3ophXwTZn.o
Preorder traversal before deletion:
10 20 5 25 30
Preorder traversal after deletion:
10 20 5 25 30


=== Code Execution Successful ===
```

main.c

Share Run

```c

// A utility function to print preorder traversal of the tree
void preOrder(struct Node *root) {
    if (root != NULL) {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Helper function to create a new node
struct Node* newNode(int key) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
    node->left = node->right = NULL;
    node->height = 1; // New node is initially at height 1
    return node;
}

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

    // Insert nodes
    root = newNode(10);
    root->left = newNode(20);
    root->right = newNode(30);
    root->left->left = newNode(5);
    root->left->right = newNode(25);

    printf("Preorder traversal before deletion:\n");
    preOrder(root);
    printf("\n");

    // Delete a node
    root = deleteNode(root, 20);

    printf("Preorder traversal after deletion:\n");
    preOrder(root);
    printf("\n");

    return 0;
}
```

Output

```
/tmp/t3ophXwTZn.o
Preorder traversal before deletion:
10 20 5 25 30
Preorder traversal after deletion:
10 20 5 25 30


=== Code Execution Successful ===
```

2. Heap Sort Implementation

main.c

```c
#include <stdio.h>

void heapify(int arr[], int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i >= 0; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; 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("Unsorted array: ");
```

Output

```
/tmp/KS9upjXAYh.o
Unsorted array: 12 11 13 5 6 7
Sorted array: 5 6 7 11 12 13


=== Code Execution Successful ===
```

main.c

```c
        largest = right;

    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i >= 0; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; 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("Unsorted array: ");
    printArray(arr, n);

    heapSort(arr, n);

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

    return 0;
}
```

Output

```
/tmp/KS9upjXAYh.o
Unsorted array: 12 11 13 5 6 7
Sorted array: 5 6 7 11 12 13


=== Code Execution Successful ===
```

3. Priority Queue using Heap

main.c

Share Run

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

struct PriorityQueue {
    int* heap;
    int size;
    int capacity;
};

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void heapify(struct PriorityQueue* pq, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

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

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

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

void insert(struct PriorityQueue* pq, int value) {
    if (pq->size == pq->capacity) {
        printf("Priority Queue is full\n");
        return;
    }

    pq->heap[pq->size] = value;
    int i = pq->size;
    pq->size++;

    // Fix the max heap property if it is violated
    while (i != 0 && pq->heap[(i - 1) / 2] < pq->heap[i]) {
        swap(&pq->heap[i], &pq->heap[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

int extractMax(struct PriorityQueue* pq) {
    if (pq->size <= 0) {
        printf("Priority Queue is empty\n");
        return -1;
    }
    if (pq->size == 1) {
        pq->size--;
        return pq->heap[0];
    }

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

    // Call heapify to restore heap property
    heapify(pq, 0);

    return root;
}

void printPriorityQueue(struct PriorityQueue* pq) {
    for (int i = 0; i < pq->size; i++) {
        printf("%d ", pq->heap[i]);
    }
    printf("\n");
}

int main() {
    struct PriorityQueue pq;
    pq.size = 0;
    pq.capacity = 10;
    pq.heap = (int*)malloc(pq.capacity * sizeof(int));

    insert(&pq, 10);
    insert(&pq, 20);
    insert(&pq, 5);
    insert(&pq, 30);
```

Output

```
/tmp/qC67OCCeZR.o
Priority Queue (Max-Heap): 30 20 5 10 15
Extracted max: 30
Priority Queue after extraction: 20 15 5 10


=== Code Execution Successful ===
```

main.c

Share Run

```c
    int root = pq->heap[0];
    pq->heap[0] = pq->heap[pq->size - 1];
    pq->size--;

    // Call heapify to restore heap property
    heapify(pq, 0);

    return root;
}

void printPriorityQueue(struct PriorityQueue* pq) {
    for (int i = 0; i < pq->size; i++) {
        printf("%d ", pq->heap[i]);
    }
    printf("\n");
}

int main() {
    struct PriorityQueue pq;
    pq.size = 0;
    pq.capacity = 10;
    pq.heap = (int*)malloc(pq.capacity * sizeof(int));

    insert(&pq, 10);
    insert(&pq, 20);
    insert(&pq, 5);
    insert(&pq, 30);
    insert(&pq, 15);

    printf("Priority Queue (Max-Heap): ");
    printPriorityQueue(&pq);

    printf("Extracted max: %d\n", extractMax(&pq));

    printf("Priority Queue after extraction: ");
    printPriorityQueue(&pq);

    free(pq.heap);

    return 0;
}
```

Output

```
/tmp/qC670CCeZR.o
Priority Queue (Max-Heap): 30 20 5 10 15
Extracted max: 30
Priority Queue after extraction: 20 15 5 10


=== Code Execution Successful ===
```