Ejercicios de Arboles Binarios, AVL y Huffman en C++

Ejercicio 5: Implementación de un Árbol AVL

Solución:

```
int height(tree* N) {
    if (N == nullptr) return 0;
    return N->height;
}

int max(int a, int b) {
    return (a > b) ? a : b;
}

tree* rightRotate(tree* y) {
    tree* x = y->left;
    tree* 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;
}

tree* leftRotate(tree* x) {
    tree* y = x->right;
```

```
    tree* 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(tree* N) {

    if (N == nullptr) return 0;

    return height(N->left) - height(N->right);

}


tree* insert(tree* node, int key) {

    if (node == nullptr) return(createLeaf(key));


    if (key < node->data)

        node->left = insert(node->left, key);

    else if (key > node->data)

        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->data)
        return rightRotate(node);

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

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

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

    return node;
}
```

Ejercicio 6: Eliminación en un Árbol AVL

Solución:

```
tree* deleteNode(tree* root, int key) {
    if (root == nullptr) return root;

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

            if (temp == nullptr) {
                temp = root;
                root = nullptr;
            } else
                *root = *temp;
            delete temp;
        } else {
            tree* temp = minValueNode(root->right);
            root->data = temp->data;
            root->right = deleteNode(root->right, temp->data);
```

```
    }
}

if (root == nullptr) 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;
```

```
}
```

Ejercicios de Arboles Binarios, AVL y Huffman en C++

Ejercicio 7: Implementación de Árbol de Huffman

Solución:

```
struct MinHeapNode {
    char data;
    unsigned freq;
    MinHeapNode *left, *right;
    MinHeapNode(char data, unsigned freq) {
        left = right = nullptr;
        this->data = data;
        this->freq = freq;
    }
};

struct compare {
    bool operator()(MinHeapNode* l, MinHeapNode* r) {
        return (l->freq > r->freq);
    }
};

void printCodes(struct MinHeapNode* root, string str) {
    if (!root) return;
    if (root->data != '$') cout << root->data << ": " << str << "\n";
    printCodes(root->left, str + "0");
```

```
    printCodes(root->right, str + "1");
}


void HuffmanCodes(char data[], int freq[], int size) {
    struct MinHeapNode *left, *right, *top;
    priority_queue<MinHeapNode*, vector<MinHeapNode*>, compare> minHeap;


    for (int i = 0; i < size; ++i)
        minHeap.push(new MinHeapNode(data[i], freq[i]));


    while (minHeap.size() != 1) {
        left = minHeap.top();
        minHeap.pop();
        right = minHeap.top();
        minHeap.pop();


        top = new MinHeapNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        minHeap.push(top);
    }


    printCodes(minHeap.top(), "");
}
```

Ejercicios de Arboles Binarios, AVL y Huffman en C++

Ejercicio 8: Codificación de Huffman

Solución:

```
void HuffmanCodes(char data[], int freq[], int size) {
    struct MinHeapNode *left, *right, *top;
    priority_queue<MinHeapNode*, vector<MinHeapNode*>, compare> minHeap;

    for (int i = 0; i < size; ++i)
        minHeap.push(new MinHeapNode(data[i], freq[i]));

    while (minHeap.size() != 1) {
        left = minHeap.top();
        minHeap.pop();
        right = minHeap.top();
        minHeap.pop();

        top = new MinHeapNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        minHeap.push(top);
    }

    printCodes(minHeap.top(), "");
}
```

Ejercicios de Arboles Binarios, AVL y Huffman en C++

Ejercicio 9: Decodificación de Huffman

Solución:

```
string decodeHuffman(struct MinHeapNode* root, string str) {
    string ans = "";
    struct MinHeapNode* curr = root;
    for (int i = 0; i < str.size(); i++) {
        if (str[i] == '0')
            curr = curr->left;
        else
            curr = curr->right;

        if (!curr->left && !curr->right) {
            ans += curr->data;
            curr = root;
        }
    }
    return ans;
}
```

Ejercicios de Arboles Binarios, AVL y Huffman en C++

Ejercicio 10: Optimización del Árbol de Huffman

Solución:

```
void optimizeHuffman() {
    // Para manejar grandes volúmenes de datos, una posible optimización
    // puede ser usar tablas hash para almacenar los códigos de Huffman
    // y evitar la reconstrucción del árbol cada vez que se quiera decodificar.
    // Además, se puede usar técnicas como compresión de tablas o codificación
    // más eficiente para largas secuencias de bits.
}
```

Ejercicio 11: Búsqueda en un Árbol AVL

Solución:

```
tree* search(tree* root, int key) {
    if (root == nullptr || root->data == key)
        return root;
    if (key < root->data)
        return search(root->left, key);
    return search(root->right, key);
}
```

Ejercicios de Arboles Binarios, AVL y Huffman en C++

Ejercicio 12: Altura de un Árbol AVL

Solución:

```
int height(tree* N) {
    if (N == nullptr)
        return 0;
    return N->height;
}
```