Universidad de san Carlos de Guatemala

Facultad de ingeniería

Escuela de ciencias y sistemas

Manual Tecnico

Renta de activos

Víctor Hugo Velasquez

202100054

Estructura de datos / sección A

GitHub: vjr-velasquez

Introducción

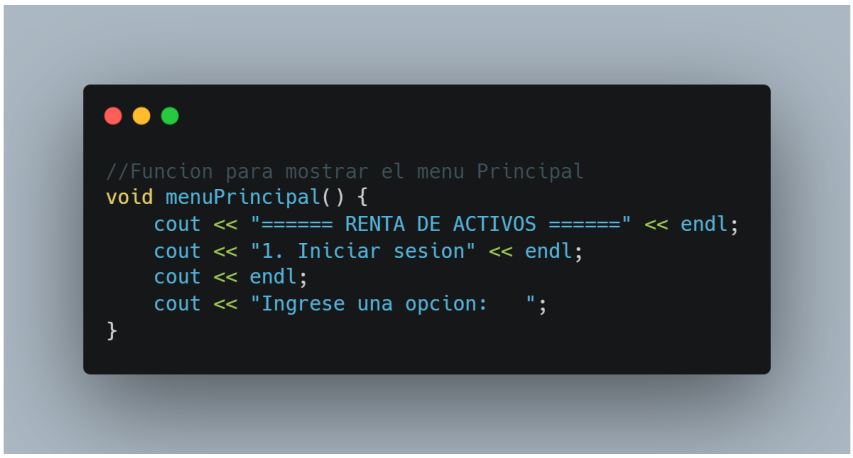
El presente **Manual Técnico** tiene como objetivo proporcionar una guía detallada sobre el funcionamiento interno, implementación y estructura del sistema **"Renta de Activos"**, desarrollado como parte del curso **Estructuras de Datos "A"** en la **Universidad de San Carlos de Guatemala**. Este sistema fue implementado en **C++** y está diseñado para operar como una aplicación en consola que permite la gestión eficiente de activos disponibles para la renta entre usuarios.

El sistema utiliza diversas **estructuras de datos avanzadas**, tales como:

- **Matriz Dispersa**: Organiza la información de los usuarios según departamentos y empresas.
- **Árbol AVL**: Almacena y gestiona los activos disponibles de cada usuario, garantizando un acceso eficiente y equilibrado.
- **Lista Circular Doblemente Enlazada**: Mantiene un historial ordenado de las transacciones realizadas en el sistema.

Fases del uso del programa

En esta primera sección de código tenemos lo que es una función con el menú principal en donde se le indica al usuario desde la consola cual es la opción que debe de elegir



```
//Funcion para mostrar el menu Principal
void menuPrincipal() {
    cout << "====== RENTA DE ACTIVOS ======" << endl;
    cout << "1. Iniciar sesion" << endl;
    cout << endl;
    cout << "Ingrese una opcion:   ";
}
```

En este segundo fragmento de código se encuentra lo que es el registro de los usuarios que solo el administrador puede crear

```
// Funcion del admin para crear usuario
void registroUsario() {
    string nombreUsuario, contrasena, departamento, empresa;
    system("cls");
    cout << " ================ CREACION DE USUARIO ================ " <<endl;
    cout << endl;
    cout << "Ingrese el nombre de usuario: ";
    cin >> nombreUsuario;
    cout << "Ingrese la contraseña: ";
    cin >> contrasena;
    cout << "Ingrese el departamento: ";
    cin >> departamento;
    cout << "Ingrese la empresa: ";
    cin >> empresa;
    matriz.agregarNodo(empresa,departamento,Usuario(nombreUsuario,contrasena)); // Estamos agregando a
la matriz
}
```

Este fragmento del código indica las opciones que tiene el usuario administrador

```
// Funcion para agregar el menu del admin
void menuAdmin() {
    int opcion;
    while (true) {

        cout << "============ MENU ADMINISTRADOR ============" << endl;
        cout << "1. Registrar un usuario " << endl;
        cout << "2. Reporte de matriz dispersa " << endl;
        cout << "3. Reporte de activos disponibles de un Depto" << endl;
        cout << "4. Reporte de activos disponibles de una empresa" << endl;
        cout << "5. Reporte de transacciones" << endl;
        cout << "6. Reporte de activos de un usuario" << endl;
        cout << "7. Activos rentados por un Usuario" << endl;
        cout << "8. Ordenar Transacciones" << endl;
        cout << "9. Salir" << endl;
        cout << "Ingrese una opcion: ";
        cin >> opcion;

        if (opcion == 1) {
            registroUsario();
        }
        else if(opcion == 9) {
            break;
        }
        else {
            cout << "Opcion no valida. Solo se permite la opcion 1. Por favor intente de nuevo." <<
endl;
        }
    }
}
```

Fragmento de código indicando las opciones que solo le pueden al aparecer al usuario que no es administrador y solo tiene la potestad de hacer transacciones con los activos

```
//Funcion para mostrar el menu de las opciones del usario
void menuOpciones(const string& usuario) {
    int opcion;
    while (true) {
        cout << "========== Bienvenido " << usuario << endl;
        cout << endl;
        cout << "========= MENU DE OPCIONES =========" << endl;
        cout << endl;
        cout << "1. AGREGAR ACTIVO " << endl;
        cout << "2. ELIMINAR ACTIVO " << endl;
        cout << "3. MODIFICAR ACTIVO " << endl;
        cout << "4. RENTAR ACTIVO " << endl;
        cout << "5. ACTIVOS RENTADOS " << endl;
        cout << "6. MIS ACTIVOS RENTADOS " << endl;
        cout << "7. CERRAR SESION " << endl;
        cin >> opcion;
        if (opcion == 1) {
            registroUsario();
        }
        else if(opcion == 7) {
            break;
        }
        else {
            cout << "Opcion no valida. Solo se permite la opcion 1. Por favor intente de nuevo." <<
endl;
        }
    }
}
```

En esta parte del código se le indica la lógica a la hora de ingresar a los distintos tipos de menú en los que el programa se va a ejecutar. Se le hace este tipo de observaciones porque existe un usario ya quemado para el cual la lógica del programa empieza a funcionar

```
// Función para mostrar el menú de login
void menuLogeo() {
    string username, password, depto, company;
    while (true) {

        cout << "============ MENÚ DE LOGIN ============" << endl;
        cout << ">> Ingrese su nombre de usuario: ";
        cin >> username;
        cout << endl << ">> Ingrese su contraseña: ";
        cin >> password;

        if (username == "admin" && password == "edd123") {
            menuAdmin();
        } else {
            cout << endl << ">> Ingrese un Departamento: ";
            cin >> depto;
            cout << endl << ">> Ingrese la empresa a la que pertenece: ";
            cin >> company;
            bool usuarioExistente = matriz.buscarValor(company , depto , username , password);
 /* Aqui buscamos el user en la matriz dispersa */
            cout << usuarioExistente << endl;
            if(usuarioExistente) {
                menuOpciones(username);
            }else {
                cout << "Contra incorrecta ......" << endl;
                cout << "Usuario incorrecto  ......" << endl;
                cout << "Depto incorrecto  ......" << endl;
                cout << "Empresa incorrecto  ......" << endl;
            }
        }
    }
}
```

Aca en este fragmento podemos ver la matriz.h en la cual se encuentra todas las funciones para los activos.

```
#ifndef MATRIZDISPERSA_H
#define MATRIZDISPERSA_H

#include <string>
#include <fstream>
#include "usuario.h"

struct NodoMatriz {
    std::string fila;
    std::string columna;
    Usuario valor;
    NodoMatriz* siguienteFila;  // Apunta al siguiente nodo en la misma fila
    NodoMatriz* siguienteColumna; // Apunta al siguiente nodo en la misma columna

    NodoMatriz(const std::string& f, const std::string& c, const Usuario& v)
        : fila(f), columna(c), valor(v), siguienteFila(nullptr), siguienteColumna(nullptr)
}};

class matrizDispersa {
private:
    NodoMatriz* cabeza;

    NodoMatriz* buscarFila(const std::string& fila) const {
        NodoMatriz* actual = cabeza->siguienteFila;
        while (actual != nullptr && actual->fila != fila) {
            actual = actual->siguienteFila;
        }
        return actual;
    }

    NodoMatriz* buscarColumna(const std::string& columna) const {
        NodoMatriz* actual = cabeza->siguienteColumna;
        while (actual != nullptr && actual->columna != columna) {
            actual = actual->siguienteColumna;
        }
        return actual;
    }            cout << "Contra incorrecta ......" << endl;
                cout << "Usuario incorrecto  ......" << endl;
                cout << "Depto incorrecto  ......" << endl;
                cout << "Empresa incorrecto  ......" << endl;
            }
        }
    }
}
```

```cpp
void agregarActivo(const std::string& fila, const std::string& columna , const std::string& user, const
std::string& id, const std::string& nombre, const std::string& descripcion, const int& dias) const {
        NodoMatriz* nodoFila = buscarFila(fila);
        if (nodoFila) {
            NodoMatriz* actual = nodoFila->siguienteColumna;
            while (actual != nullptr) {
                if (actual->columna == columna) {
                    if(actual->valor.user == user) {
                        actual->valor.avl.insertar(Activos("",id,nombre,descripcion,dias,"disponible"));
                        break;
                    }
                }
                actual = actual->siguienteColumna;
            }
        }
    }

    void buscarEliminarActivo(const std::string& fila, const std::string& columna , const std::string&
user, const std::string& id) const {
        NodoMatriz* nodoFila = buscarFila(fila);
        if (nodoFila) {
            NodoMatriz* actual = nodoFila->siguienteColumna;
            while (actual != nullptr) {
                if (actual->columna == columna) {
                    if(actual->valor.user == user) {
                        actual->valor.avl.eliminar(id);
                        break;
                    }
                }
                actual = actual->siguienteColumna;
            }
        }
    }
```

```cpp
void rentarActivo(const std::string& user, const std::string& id, const std::string& estado, const int
dias) const {
        NodoMatriz* filaActual = cabeza->siguienteFila;
        while (filaActual != nullptr) {
            NodoMatriz* columnaActual = filaActual->siguienteColumna;
            while (columnaActual != nullptr) {
                columnaActual->valor.avl.rentarActivo(user,id, estado, dias);
                columnaActual = columnaActual->siguienteColumna;
            }
            filaActual = filaActual->siguienteFila;
        }
    }

    void mostarActivo(const std::string& fila, const std::string& columna, const std::string& user, const
std::string& es) const {
        NodoMatriz* nodoFila = buscarFila(fila);
        if (nodoFila) {
            NodoMatriz* actual = nodoFila->siguienteColumna;
            while (actual != nullptr) {
                if (actual->columna == columna) {
                    if(actual->valor.user == user) {
                        if(es == "rentado") {
                            actual->valor.avl.enOrdenRentado();
                            break;
                        }
                        if(es == "todo") {
                            std::cout << "modificando.....";
                            actual->valor.avl.mostrarActivosEliminar();
                            break;
                        }
                    }
                }
                actual = actual->siguienteColumna;
            }
        }
    }
```

La clase Usuario tiene un constructor que permite inicializar estos atributos cuando se crea un nuevo objeto Usuario. Si no se proporcionan valores, se inicializan con valores predeterminados (cadenas vacías para user y contra, y un árbol AVL vacío para avl).

También hay un método toString que devuelve una representación en forma de cadena del objeto Usuario, mostrando el nombre de usuario y la contraseña.

```
#ifndef USUARIO_H
#define USUARIO_H

#include <string>
#include <iostream>
#include "avl_tree.h"

class Usuario {
public:
    std::string user;
    std::string contra;
    ArbolAVL avl;

    Usuario(const std::string& user = "", const std::string& contra = "", const ArbolAVL& avl =
ArbolAVL())
    : user(user), contra(contra), avl(avl) {}

    std::string toString() const {
        return "Usuario: " + user + ", Contraseña: " + contra;
    }

    friend std::ostream& operator<<(std::ostream& os, const Usuario& usuario) {
        os << usuario.toString();
        return os;
    }
};

#endif //USUARIO_H
```

Este código define una clase llamada DatosTransaccion que representa los datos de una transacción. Piensa en una transacción como una operación o un evento que ocurre en un sistema, como una compra, una venta, un alquiler, etc. La clase tiene varios atributos que almacenan información sobre la transacción

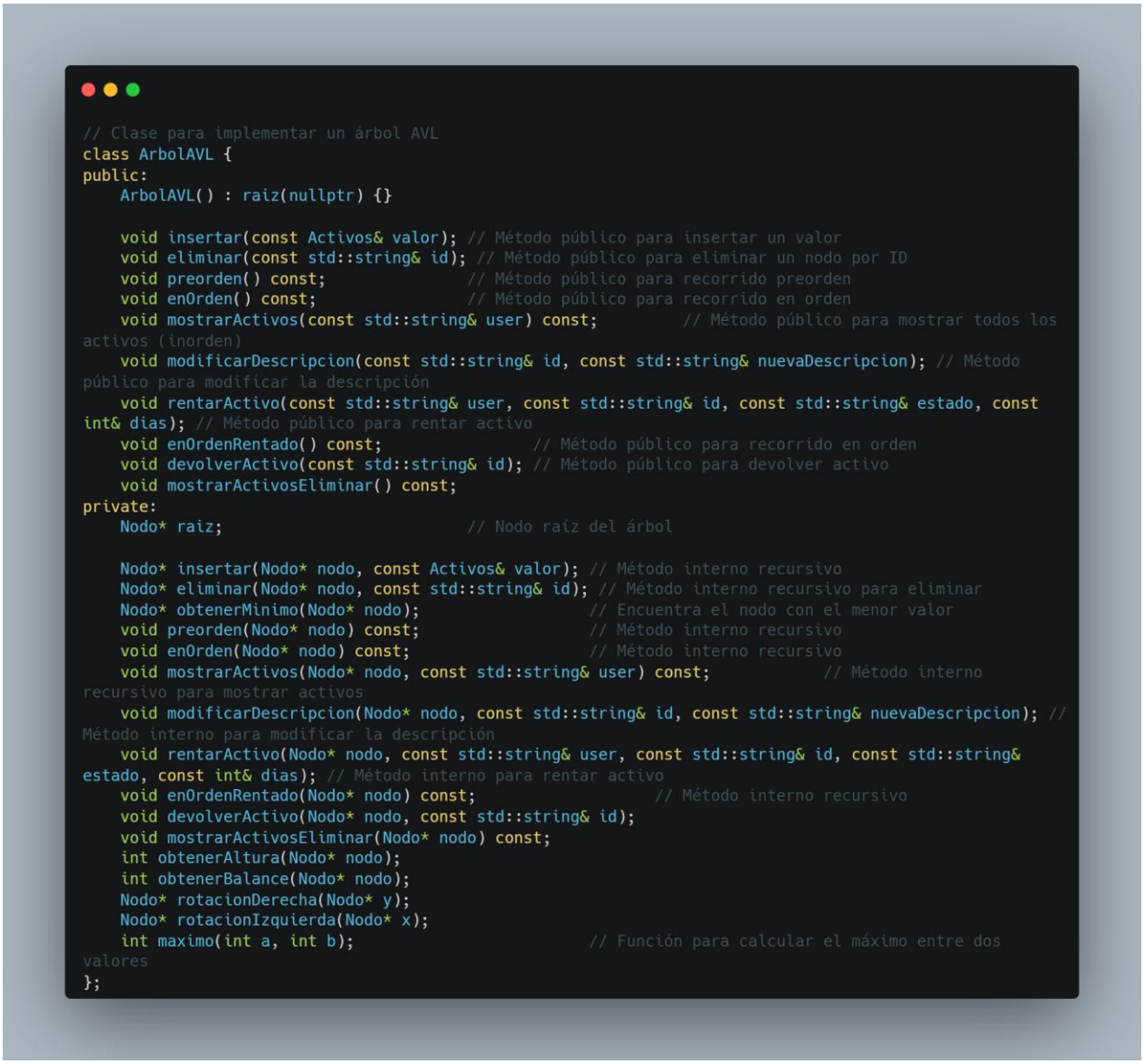
```
#ifndef DATOSTRANSACCION_H
#define DATOSTRANSACCION_H
#include <string>

class DatosTransaccion {
public:
    std::string id;
    std::string id_trans;
    std::string usuario;
    std::string depto;
    std::string empresa;
    std::string fecha;
    int dias;

    // Constructor
    DatosTransaccion(const std::string& id_, const std::string& id_trans_, const std::string& usuario_,
                     const std::string& depto_, const std::string& empresa_, const std::string& fecha_,
int dias_)
        : id(id_), id_trans(id_trans_), usuario(usuario_), depto(depto_), empresa(empresa_),
fecha(fecha_), dias(dias_) {}
};

#endif //DATOSTRANSACCION_H
```

Este código define un árbol AVL que permite realizar operaciones de inserción, eliminación, modificación y recorrido de manera eficiente. Los métodos de rotación aseguran que el árbol se mantenga balanceado, lo que garantiza un rendimiento óptimo para las operaciones de búsqueda y actualización.



```
// Clase para implementar un árbol AVL
class ArbolAVL {
public:
    ArbolAVL() : raiz(nullptr) {}

    void insertar(const Activos& valor); // Método público para insertar un valor
    void eliminar(const std::string& id); // Método público para eliminar un nodo por ID
    void preorden() const;              // Método público para recorrido preorden
    void enOrden() const;               // Método público para recorrido en orden
    void mostrarActivos(const std::string& user) const;         // Método público para mostrar todos los
activos (inorden)
    void modificarDescripcion(const std::string& id, const std::string& nuevaDescripcion); // Método
público para modificar la descripción
    void rentarActivo(const std::string& user, const std::string& id, const std::string& estado, const
int& dias); // Método público para rentar activo
    void enOrdenRentado() const;               // Método público para recorrido en orden
    void devolverActivo(const std::string& id); // Método público para devolver activo
    void mostrarActivosEliminar() const;
private:
    Nodo* raiz;                         // Nodo raíz del árbol

    Nodo* insertar(Nodo* nodo, const Activos& valor); // Método interno recursivo
    Nodo* eliminar(Nodo* nodo, const std::string& id); // Método interno recursivo para eliminar
    Nodo* obtenerMinimo(Nodo* nodo);                  // Encuentra el nodo con el menor valor
    void preorden(Nodo* nodo) const;                  // Método interno recursivo
    void enOrden(Nodo* nodo) const;                   // Método interno recursivo
    void mostrarActivos(Nodo* nodo, const std::string& user) const;           // Método interno
recursivo para mostrar activos
    void modificarDescripcion(Nodo* nodo, const std::string& id, const std::string& nuevaDescripcion); //
Método interno para modificar la descripción
    void rentarActivo(Nodo* nodo, const std::string& user, const std::string& id, const std::string&
estado, const int& dias); // Método interno para rentar activo
    void enOrdenRentado(Nodo* nodo) const;                  // Método interno recursivo
    void devolverActivo(Nodo* nodo, const std::string& id);
    void mostrarActivosEliminar(Nodo* nodo) const;
    int obtenerAltura(Nodo* nodo);
    int obtenerBalance(Nodo* nodo);
    Nodo* rotacionDerecha(Nodo* y);
    Nodo* rotacionIzquierda(Nodo* x);
    int maximo(int a, int b);                         // Función para calcular el máximo entre dos
valores
};
```

```cpp
// Implementación de la función maximo
int ArbolAVL::maximo(int a, int b) {
    return (a > b) ? a : b;
}

// Implementación de obtenerAltura
int ArbolAVL::obtenerAltura(Nodo* nodo) {
    return nodo ? nodo->altura : 0;
}

// Implementación de obtenerBalance
int ArbolAVL::obtenerBalance(Nodo* nodo) {
    return nodo ? obtenerAltura(nodo->izquierdo) - obtenerAltura(nodo->derecho) : 0;
}

// Implementación de la rotación a la derecha
Nodo* ArbolAVL::rotacionDerecha(Nodo* y) {
    Nodo* x = y->izquierdo;
    Nodo* T2 = x->derecho;

    x->derecho = y;
    y->izquierdo = T2;

    y->altura = maximo(obtenerAltura(y->izquierdo), obtenerAltura(y->derecho)) + 1;
    x->altura = maximo(obtenerAltura(x->izquierdo), obtenerAltura(x->derecho)) + 1;

    return x;
}
```

```cpp
// Método público para insertar un valor
void ArbolAVL::insertar(const Activos& valor) {
    raiz = insertar(raiz, valor); // Llama al método recursivo privado
}

// Implementación del recorrido en orden recursivo, filtrando por estado "disponible"
void ArbolAVL::enOrden(Nodo* nodo) const {
    if (nodo) {
        enOrden(nodo->izquierdo); // Recorrer el subárbol izquierdo
        if (nodo->valor.estado == "disponible") { // Mostrar solo si el estado es "disponible"
            std::cout << "ID: " << nodo->valor.id
                      << ", Nombre: " << nodo->valor.nombre
                      << ", Descripcion: " << nodo->valor.descripcion
                      << ", Días: " << nodo->valor.dias
                      << ", Estado: " << nodo->valor.estado << std::endl;
        }
        enOrden(nodo->derecho); // Recorrer el subárbol derecho
    }
}

// Método público para el recorrido en orden
void ArbolAVL::enOrden() const {
    enOrden(raiz); // Llama al método recursivo privado
}
```

Este código define una aplicación de gestión de activos donde los usuarios pueden iniciar sesión, agregar, eliminar, modificar y rentar activos. También hay un administrador que puede registrar nuevos usuarios y generar varios reportes. La aplicación utiliza una matriz dispersa para almacenar los activos y una lista doble circular para almacenar las transacciones.

```cpp
#include <iostream>
#include <string>
#include <random>
#include <ctime>
#include "matrizDispersa.h"
#include "listaDobleCircular.h"

matrizDispersa matriz; // Creacion de matriz
ListaDobleCircular lisD;
using namespace std;

std::string generarID(int longitud) {
    const std::string caracteres =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    std::mt19937 generador(rd()); // Generador de números aleatorios
    std::uniform_int_distribution<> distribucion(0, caracteres.size() - 1);

    std::string id;
    for (int i = 0; i < longitud; ++i) {
        id += caracteres[distribucion(generador)];
    }

    return id;
}

//Funcion para mostar mis activos rentados
void misActivosRentados(const string& usuario, const string& depto, const string& empresa) {
    cout << "============ MIS ACTIVOS RENTADOS ============ " << endl;
    matriz.mostarActivo(empresa, depto, usuario, "rentado");
    cout << "========================================== " << endl;
}
```

```cpp
//Funcion para mostrar los activos rentados
void activosRentados(const string& usuario, const string& depto, const string& empresa)
{   int opcion = 0;
    cout << "============ ACTIVOS RENTADOS ============ " << endl;
    matriz.mostarActivoRentados(usuario);
    cout << "========================================== " << endl;
    cout << "1. Para devolver Activo" << endl;
    cin >> opcion;
    if(opcion == 1) {
        std:: string id = "";
        cout << "Ingrese Id que quiere devolver: " << endl;
        cin >> id;
        matriz.devolverActivo(id);
    }
}

//Funcion para mostrar el menu Principal
void menuPrincipal() {
    cout << "====== RENTA DE ACTIVOS ======" << endl;
    cout << "1. Iniciar sesion" << endl;
    cout << endl;
    cout << "Ingrese una opcion:   ";
}
```

```cpp
//Funcion para rentar un activo
void rentarActivo(const string& usuario, const string& depto, const string& empresa) {
    // Obtener el tiempo actual como un objeto time_t
    std::time_t tiempoActual = std::time(nullptr);
    // Convertir el tiempo a una estructura de tiempo local
    std::tm* tiempoLocal = std::localtime(&tiempoActual);
    // Formatear la fecha en una cadena
    char buffer[11]; // Espacio suficiente para "YYYY-MM-DD\0"
    std::strftime(buffer, sizeof(buffer), "%Y-%m-%d", tiempoLocal);
    // Guardar la fecha en una variable tipo std::string
    std::string fechaActual = buffer;
    int longitud = 15; // Longitud del ID
    std::string id_trans = generarID(longitud);
    std:: string id = "";
    int dias = 0;
    cout << "======RENTANDO UN ACITVO======" << endl;
    matriz.mostrarActivos(usuario);
    cout << "================================" << endl;
    cout << "Ingrese Id que quiere rentar: " << endl;
    cin >> id;
    cout << "Cuantos dias: " << endl;
    cin >> dias;
    matriz.rentarActivo(usuario, id, "rentado", dias);
    lisD.agregar(DatosTransaccion(id,id_trans,usuario,depto,empresa,fechaActual,dias));
}
```