UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERÍA
ESTRUCTURA DE DATOS
CATEDRÁTICO: ING. EDGAR RENE ORNELIS HOILS
TUTOR ACADÉMICO: ELIAN SAUL ESTRADA URBINA

MANUAL TÉCNICO
RENTA DE ACTIVOS

ENNER ESAÍ MENDIZABAL CASTRO
CARNÉ: 202302220
SECCIÓN: A

GUATEMALA, 16 DE DICIEMBRE DEL 2,024

ÍNDICE

INTRODUCCIÓN

Este documento técnico tiene como objetivo proporcionar una descripción detallada de la arquitectura, diseño e implementación del programa Renta de Activos. Se presentarán las soluciones técnicas empleadas, los tipos de datos abstractos utilizados y la lógica de programación subyacente a cada una de sus funcionalidades. Con esto, se busca facilitar la comprensión del sistema a desarrolladores interesados en replicar o extender sus capacidades.

OBJETIVOS

1. GENERAL

1.1. Documentar de manera exhaustiva las soluciones técnicas adoptadas en la implementación del programa Renta de Activos.

2. ESPECÍFICOS

2.1. Detallar los Tipos de Datos Abstractos (TDA) utilizados en la construcción del programa y su justificación.

2.2. Describir la lógica de implementación de las estructuras de datos y algoritmos empleados, así como su relación con las funcionalidades del sistema.

ALCANCES DEL SISTEMA

Este manual se centra en la arquitectura interna del programa Renta de Activos, proporcionando una visión detallada de su diseño y desarrollo. Se abordan aspectos como:

- Estructura: Organización general del código y relación entre los diferentes módulos.
- Algoritmos: Descripción de los algoritmos utilizados para resolver los problemas planteados.
- Tipos de datos: Definición y uso de los TDA empleados para representar la información.

- Tecnologías: Herramientas y lenguajes de programación utilizados en el desarrollo.

Este documento está dirigido a desarrolladores con conocimientos básicos de programación y un interés particular en la arquitectura de software y el diseño de algoritmos.

ESPECIFICACIÓN TÉCNICA

- **REQUISITOS DE HARDWARE**
  - Resolución mínima de 1024x768
  - 8 GB de memoria RAM
  - Dispositivos de entrada y salida: Pantalla, Ratón y Teclado.
  - 3.5GB de almacenamiento libre en disco
- **REQUISITOS DE SOFTWARE**
  - Sistema operativo: Windows, macOS o Linux
  - Compilador de C++/C. (Si se tiene Clion, no es necesario)
  - Entorno de desarrollo. Se sugiere utilizar un entorno de desarrollo integrado (IDE) como Clion, aunque cualquier editor de texto con soporte para C++ puede ser utilizado.

LOGICA Y DESCRIPCIÓN DE LA SOLUCIÓN

Para la creación del programa Renta Activos, se utilizaron varias estructuras utilizando las clases, usando convenciones para tener una mejor calidad de código creando *headers*, las cuales son:

Lista Circular Doblemente enlazada

Una lista circular doblemente enlazada es una estructura de datos lineal en la que cada elemento, que se llama nodo, contiene, además de los datos, dos punteros: uno que referencia al nodo siguiente y otro que apunta al nodo anterior. En esta estructura, a diferencia de las listas normales, el último nodo apunta al primero, y el primero al último, formando así un círculo.
Para este proyecto, se usó este tipo de lista para guardar las transacciones que se harían durante la ejecución del programa Renta Activos y sus métodos principales son estos:

Constructor

- **Lista()**: Inicializa una lista circular doblemente enlazada vacía, estableciendo los punteros inicio y fin en nullptr y el tamaño en 0.

```
10      //Constructor
11      Lista::Lista(){
12          this->tamano = 0;
13          this->inicio = nullptr;
14          this->fin = nullptr;
15      }
```

Operaciones de la lista

- **agregarNodo(Transaccion* transaccion):** Añade un nuevo nodo al final de la lista creando un nuevo NodoTransaccion con la transacción proporcionada. Si la lista está vacía, el nuevo nodo se convierte tanto en el inicio como en el fin de la lista, apuntándose a sí mismo como anterior y siguiente. Si la lista no está vacía, se actualizan los punteros del último

nodo y del nuevo nodo para mantener la circularidad y el enlace doble.

```
//Función para agregar los nuevos nodos
void Lista::agregarNodo(Transaccion *transaccion){
    NodoTransaccion *nuevo=new NodoTransaccion(transaccion);

    if (this->tamano <= 0){//Si es que no hay nada en la lista circular
        this->inicio=nuevo; //El inicio es igual al nodo nuevo
        this->fin=nuevo; //El final es igual al nodo nuevo
        nuevo->setAnterior(this->inicio); //El anterior del nodo nuevo es él mismo
        nuevo->setSiguiente(this->fin); //El siguiente del nodo nuevo es él mismo
    }else{
        this->fin->setSiguiente(nuevo);  // El último nodo apunta al nuevo
        nuevo->setAnterior(this->fin);   // El nuevo nodo apunta al anterior último
        nuevo->setSiguiente(this->inicio);  // El nuevo nodo apunta al inicio
        this->inicio->setAnterior(nuevo);   // El primer nodo apunta al nuevo como anterior
        this->fin = nuevo;  // El nuevo ahora es el final
    }
    this->tamano++; //Aumento el tamaño del nodo
}
```

- **obtenerNodoEnPosicion(int posicion)**: Retorna un puntero al nodo que se encuentra en la posición especificada. Realiza una busqueda iterativa.

```
NodoTransaccion Lista::obtenerNodoEnPosicion(int posicion){
    if (inicio == nullptr || inicio == fin) return nullptr;
    NodoTransaccion* aux=this->inicio;
    int contador=0;
    while (contador<posicion){
        aux=aux->getSiguiente();
        contador++;
    }
    return *aux;
}
```

- **imprimirNodos():** Imprime la información contenida en cada nodo de la lista. Recorre la lista circular y muestra el ID, nombre y descripción de cada nodo. Maneja el caso de una lista vacía

```
void Lista::imprimirNodos() {
    if (inicio == nullptr || fin == nullptr) {
        std::cout << "No tiene activos rentados" << std::endl;
        return;
    }
    int contador = 1;
    NodoTransaccion* actual = inicio; // Nodo que recorrerá la lista
    do {
        // Imprime la información
        std::cout<<std::to_string(contador)<<". " << actual->getId()<< " - "<<actual->getNombre
        actual = actual->getSiguiente(); // Avanzar al siguiente nodo
        contador++;
    } while (actual != inicio); // Continuar hasta regresar al inicio
    std::cout << "------------------------------------------------------------" << std::endl;
}
```

- **ordenarAscendente()**: Ordena la lista de forma ascendente según el idTransaccion de los objetos Transaccion almacenados en cada nodo. Utiliza un algoritmo de burbuja modificado para intercambiar punteros a los nodos.

```
73      //Función para odenar la lista de forma ascendente
74  >   void Lista::ordenarAscendente(){...}
```

- **ordenarDescendente()**: Igual a ordenarAscendente(), pero ordena la lista de forma descendente según el idTransaccion (básicamente solo cambio un símbolo)

```
98      //Función para odenar la lista de forma ascendente
99  >   void Lista::ordenarDescendente(){...}
```

- **generarReporte()**: La función recorre la lista y crea nodos en el grafo .dot con la información de cada transacción, conectándolos con flechas bidireccionales para representar el doble enlace.

```
125     //Función para graficar la lista circular doblemente enlazada
126 >   void Lista::generarReporte(){...}
```

- **agregarNodoActivo(std::string id, std::string descripcion, std::string nombre):** Añade un nuevo nodo a la lista, pero en este caso, el nodo se crea directamente con un id, una descripcion y un nombre, en lugar de un objeto Transaccion completo. Esto debido a que se utilizará para guardar la información de los activos de los usuarios (reutilizando la estructura).

```
169 void Lista::agregarNodoActivo(std::string id,std::string descripcion,std::string nombre){
170     NodoTransaccion *nuevo = new NodoTransaccion( id: descripcion, descripcion: id, nombre);
171     if (this->tamano <= 0){//Si es que no hay nada en la lista circular
172         this->inicio=nuevo; //El inicio es igual al nodo nuevo
173         this->fin=nuevo; //El final es igual al nodo nuevo
174         nuevo->setAnterior(this->inicio); //El anterior del nodo nuevo es él mismo
175         nuevo->setSiguiente(this->fin); //El siguiente del nodo nuevo es él mismo
176     }else{
177         this->fin->setSiguiente(nuevo);  // El último nodo apunta al nuevo
178         nuevo->setAnterior(this->fin);   // El nuevo nodo apunta al anterior último
179         nuevo->setSiguiente(this->inicio);  // El nuevo nodo apunta al inicio
180         this->inicio->setAnterior(nuevo);   // El primer nodo apunta al nuevo como anterior
181         this->fin = nuevo;  // El nuevo ahora es el final
182     }
183     this->tamano++; //Aumento el tamaño del nodo
184 }
```

- **eliminarNodoPorId(std::string id):** Elimina un nodo de la lista buscando por su ID.

```
186     //Función para eliminar un nodo por su ID
187 >   void Lista::eliminarNodoPorId(std::string id){...}
```

- **generarReporteActivosRentados(std::string user):** Genera un reporte gráfico DOT similar a generarReporte(), pero hecho para los activos rentados.

```
227    //Función para generar un reporte para los activos rentados
228    void Lista::generarReporteActivosRentados(std::string user){...}
```

Matriz Dispersa

Una matriz dispersa es una matriz en la que la mayoría de sus elementos no ocupan espacio, lo que la distingue de un arreglo, en donde la mayoría de los elementos ya apartan especio de memoria. Para este proyecto se usó este tipo de matriz como base para guardar toda la información de los usuarios entre encabezados que permitirían ubicar cada uno fácilmente.

Constructor

- **Matriz():** inicializa las cabeceras de la matriz con valor nulo.

```
    //Constructor de la matriz
11  Matriz::Matriz(){
12      this->cabeceraHorizontal=nullptr;
13      this->cabeceraVertical=nullptr;
14  }
```

Funciones principales

- **insertarValor(std::string elian, std::string cabeH, std::string cabeV, std::string contra, std::string nombreCompleto):** Esta es la función central de toda la matriz. Esta inserta un nuevo usuario en la matriz, recibe el nombre de usuario (elian), la cabecera horizontal (cabeH, departamento), la cabecera vertical (cabeV, empresa), la contraseña (contra) y el nombre completo para crear su nodo que contiene el usuario, por lo tanto, inicializa todo lo relacionado a este. Maneja la inserción de cabeceras si no existen, la inserción del usuario en la posición correcta (basándose en las cabeceras) y el manejo de colisiones (cuando ya existe un nodo en esa posición, preguntando si se inserta adelante o atrás).

```
16     //Función para insertar un nuevo valor (sería como la principal)
       void Matriz::insertarValor(std::string elian, std::string cabeH,
138
```

- **encontrarUsuario(std::string contrasena, std::string user):** Busca un usuario en la matriz por su nombre de usuario y contraseña recorriendo la matriz buscando la coincidencia.

```
139    //Funcion para encontrar usuario a partir de nombre y contraseña
140    NodoMatriz* Matriz::encontrarUsuario(std::string contrasena, std::string user){...}
```

- **generarReporteMatriz() y generarDotGrafica():** Generan un reporte gráfico de la matriz en formato DOT, que luego se convierte a PDF con Graphviz. generarDotGrafica() es la función que crea la cadena de texto con el formato DOT, mientras que generarReporteMatriz() maneja la creación del archivo y la ejecución del comando dot.

```
367    //Funciones para generar el reporte de la matriz
368    void Matriz::generarReporteMatriz(){...}
379    std::string Matriz::generarDotGrafica(){...}
```

- **generarReporteActivosEmpresa(std::string empresa) y concatenarStringPorFila(NodoMatriz* cabV):** Generan un reporte de los activos de una empresa específica. concatenarStringPorFila() crea el string para el archivo dot con la informacion de la empresa, generarReporteActivosEmpresa() maneja la creación del archivo y la ejecucion del comando en la consola.

```
433    //Función para obtener el cuerpo de la cosa esa para la graficación
434    std::string Matriz::concatenarStringPorFila(NodoMatriz* cabV){...}
462    //Función para generar el reporte de todos los activos de una empresa
463    void Matriz::generarReporteActivosEmpresa(std::string empresa){...}
```

- **generarReporteActivosDepartamento(std::string departamento) y concatenarStringPorColumna(NodoMatriz* cabH):** Similar a las anteriores, pero generan un reporte de los activos de un departamento específico.

```
482    //Lo mismo de arriba pero para la columna :)
483    std::string Matriz::concatenarStringPorColumna(NodoMatriz* cabH){...}
511
512    void Matriz::generarReporteActivosDepartamento(std::string departamento){...}
```

- **mostrarActivosUsuario(std::string user, std::string contrasena):** Muestra los activos de un usuario específico mediante la ejeución de otra función dentro del arbol que se encuentra dentro del usuario, el cual posee los activos.

```
533     //Función para mostrar los activos del usuario
534 >   void Matriz::mostrarActivosUsuario(std::string user,std::string contrasena){...}
543
```

- **mostrarActivosRentados(std::string user, std::string contrasena):** Muestra los activos rentados por un usuario específico ejecutando una función dentro de la lista circula doblemente enlazada dentro del usuario.

```
545     //Función para mostrar los activos rentados por un usuario
546 >   void Matriz::mostrarActivosRentados(std::string user,std::string contrasena){...}
```

- **generarReporteActivosRentados(std::string user, std::string contrasena):** Genera un reporte de los activos rentados por un usuario en formato PDF llamando una funcón dentro de la lista que contiene el usuario.

```
557 //Función para crear informe de los activos rentados por un usuario :)
558 void Matriz::generarReporteActivosRentados(std::string user,std::string contrasena){
559     NodoMatriz* nodoUsuario=encontrarUsuario( contrasena, user);
560     if(nodoUsuario==nullptr){
561         std::cout<<"No se encuentra el usuario"<<std::endl;
562     }else{
563         nodoUsuario->getUsuario()->getActivosRentados()->generarReporteActivosRentados( user);
564     }
565 }
```

- **mostrarActivosDisponibles():** Muestra todos los activos disponibles en la matriz ejecutando cada unas de las funciones que hacen esto que se encuentran dentro de cada uno de lo arboloes de cada uno de lo usuario dentro de la matriz.

```
568     //Función para mostrar todos los activos disponibles
569 >   void Matriz::mostrarActivosDisponibles(){...}
```

- **encontrarActivo(std::string id):** Busca un activo por su ID en toda la matriz, recorriendo todos lo usuario y buscandolo en cada uno hasta encontrarlo y poder devolverlo.

```
599 >   AVL* Matriz::encontrarActivo(std::string id){...}
```

- **rentarActivo(std::string id, int dias, std::string usuarioRentador, std::string contrasenaRentador):** Marca un activo como rentado, actualizando su información y añadiéndolo a la lista de activos rentados del usuario que lo renta.
-

```
636  //Función para rentar un activo
637  void Matriz::rentarActivo(std::string id,int dias,std::string usuarioRentador,std::string contrasenaRentador){
638      Activo *activoParaRentar=encontrarActivo( id)->getActivo();
639      activoParaRentar->setTiempoRenta(dias);
640      activoParaRentar->setUsuarioRentador( usuarioRentador);
641
642      //Aquí ahora voy a pasar el activo a la lista de activos rentados por el usuario
643      NodoMatriz* usuarioRentadorEncontrado=encontrarUsuario( contrasenaRentador, user: usuarioRentador);
644      usuarioRentadorEncontrado->getUsuario()->getActivosRentados()->agregarNodoActivo(activoParaRentar->getId()
645  }
```

Funciones para el manejo de cabeceras en la matriz

- cabeceraV(std::string elian)
- cabeceraH(std::string elian)
- insertarCabeceraHorizontal(std::string elian)
- insertarCabeceraVertical(std::string elian)
- presenteEnCabeceraHorizontal(NodoMatriz *nodo)
- presenteEnCabeceraVertical(NodoMatriz *nodo)

Estas funciones se encargan unicamente de gestionar las cabeceras de la matriz (buscan, insertan y verifican su presencia)

```
191  //Funciones para encontrar las cabeceras
192  NodoMatriz *Matriz::cabeceraV(std::string elian){...}
209  NodoMatriz *Matriz::cabeceraH(std::string elian){...}
227
228  //Funciones para saber si están presentes en sus respectivas cabeceras
229  NodoMatriz *Matriz::presenteEnCabeceraHorizontal(NodoMatriz *nodo){...}
239  NodoMatriz *Matriz::presenteEnCabeceraVertical(NodoMatriz *nodo){...}
250  //-----------------------------------------------------------------------
251  //Función para insertar una cabecera horizontal y/o vertical
252  NodoMatriz *Matriz::insertarCabeceraHorizontal(std::string elian){...}
271  NodoMatriz *Matriz::insertarCabeceraVertical(std::string elian){...}
```

Funciones para la insersión de nodos

- insertarValor(std::string elian, std::string cabeH, std::string cabeV,std::string contra,std::string nombreCompleto)

- insertarAlFinalHorizontal(NodoMatriz *elian, NodoMatriz *cabeH)
- insertarAlFinalVertical(NodoMatriz *elian, NodoMatriz *cabeV)
- insertarAlMediaHorizontal(NodoMatriz* valor, NodoMatriz* horizontal)
- insertarAlMediaVertical(NodoMatriz* valor, NodoMatriz* vertical)
- insertarAlFinal(NodoMatriz *elian, NodoMatriz* cabeH, NodoMatriz* cabeV

Estas funciones se centran en la lógica de inserción de usuarios en la matriz, incluyendo el manejo de las diferentes posiciones y la inserción al final o en medio de las listas enlazadas.

```
289     //Funciones para insertar al usuario al final
290   > void Matriz::insertarAlFinalHorizontal(NodoMatriz *elian, NodoMatriz *cabeH){...}
299   > void Matriz::insertarAlFinalVertical(NodoMatriz *elian, NodoMatriz *cabeV){...}
308
309     //Funciones para insertar al la mitad de la final y la horizontal
310   > void Matriz::insertarAlMediaHorizontal(NodoMatriz* valor, NodoMatriz* horizontal){...}
321   > void Matriz::insertarAlMediaVertical(NodoMatriz* valor, NodoMatriz* vertical){...}
331
332     //Función para insertar al usuario al final
333   > void Matriz::insertarAlFinal(NodoMatriz *elian, NodoMatriz* cabeH, NodoMatriz* cabeV){...}
```

Funciones para la búsqueda de Nodos, Activos y Usuarios

- encontrarUsuario(std::string contrasena, std::string user)
- obtenerNodo(std::string cabeH, std::string cabeV)
- encontrarActivo(std::string id) Estas funciones se encargan de buscar nodos, ya sean usuarios o activos, dentro de la estructura de la matriz.

```
139     //Funcion para encontrar usuario a partir de nombre y contraseña
140   > NodoMatriz* Matriz::encontrarUsuario(std::string contrasena, std::string user){...}
175     //Funciones para insertar adelante y atras ----------------------------------------
176     //Función para obtener el nodo en cierta posición
177   > NodoMatriz* Matriz::obtenerNodo(std::string cabeH, std::string cabeV){...}

599   > AVL* Matriz::encontrarActivo(std::string id){...}
```

Funciones para la generación de repotes

- generarReporteMatriz()
- generarDotGrafica()
- generarReporteActivosEmpresa(std::string empresa)
- concatenarStringPorFila(NodoMatriz* cabV)

- generarReporteActivosDepartamento(std::string departamento)
- concatenarStringPorColumna(NodoMatriz* cabH)
- generarReporteActivosRentados(...) (para usuarios)

Este grupo se dedica a generar los reportes gráficos, separando la lógica de construcción del string DOT de la gestión de archivos y la llamada a Graphviz.

```
367     //Funciones para generar el reporte de la matriz
368     void Matriz::generarReporteMatriz(){...}
379     std::string Matriz::generarDotGrafica(){...}

433     //Función para obtener el cuerpo de la cosa esa para la graficación
434     std::string Matriz::concatenarStringPorFila(NodoMatriz* cabV){...}
462     //Función para generar el reporte de todos los activos de una empresa
463     void Matriz::generarReporteActivosEmpresa(std::string empresa){...}
481     //*************************************************************************
482     //Lo mismo de arriba pero para la columna :)
483     std::string Matriz::concatenarStringPorColumna(NodoMatriz* cabH){...}
511
512     void Matriz::generarReporteActivosDepartamento(std::string departamento){...}

557     //Función para crear informe de los activos rentados por un usuario :)
558     void Matriz::generarReporteActivosRentados(std::string user,std::string contrasena){...}
```

Funciones para la obtención de información de activos por consola

- mostrarActivosUsuario(std::string user,std::string contrasena)
- mostrarActivosRentados(std::string user,std::string contrasena)
- mostrarActivosDisponibles()
- activosEnRentaDEUsuario()

Estas funciones se encargan de mostrar información relacionada con los activos, ya sea de un usuario específico, los rentados o los disponibles.

```
533     //Función para mostrar los activos del usuario
534     void Matriz::mostrarActivosUsuario(std::string user,std::string contrasena){...}
545     //Función para mostrar los activos rentados por un usuario
546     void Matriz::mostrarActivosRentados(std::string user,std::string contrasena){...}

568     //Función para mostrar todos los activos disponibles
569     void Matriz::mostrarActivosDisponibles(){...}

647     //Función para imprimir los activos en renta de un usuario
648     void Matriz::activosEnRentaDEUsuario(std::string user,std::string contrasena){...}
```

Funciones para la renta de activos

- rentarActivo()

- hayActivosEnRenta()

```
636     //Función para rentar un activo
637 >   void Matriz::rentarActivo(std::string id,int dias,std::string usuarioRentador,std::string contrasenaRentador){...}
```

```
654     //Función para retornar si hay activos en renta
655 >   bool Matriz::hayActivosEnRenta(){...}
688     //*********************************************
```

ArbolAVL

Un árbol AVL es un tipo de árbol binario de búsqueda que se mantiene balanceado para que las operaciones de inserción, eliminación y búsqueda sean más eficientes. Para este proyecto se utilizó este tipo de árbol para almacenar la información de los activos de cada uno de los usarios.

Constructor:

- **ArbolAVL():** Inicializa el árbol creando una raíz nula

```
10  ArbolAVL::ArbolAVL(){
11      this->raiz=nullptr;
12  }
```

Funciones para la inserción:

- insertar(std::string id, std::string nombre, std::string descripcion): Función pública para insertar un nuevo nodo en el árbol. Crea un nuevo nodo AVL con la información proporcionada y luego llama a la función de inserción recursiva para insertar verdaderamente el nuevo nodo.
- insertar(AVL *valor, AVL *&raiz): Función recursiva que realiza la inserción del nodo en la posición correcta del árbol que es llamada por la función pública. Después de insertar, calcula el factor de equilibrio y realiza las rotaciones necesarias para balancear el árbol.

```
14 > void ArbolAVL::insertar(std::string id,std::string nombre,std::string
19 > void ArbolAVL::insertar(AVL *valor,AVL *&raiz){...}
```

Funciones para las rotaciones

- rotacionDerechaIzquierda(AVL*& nodo): Realiza una rotación doble. Primero a la derecha sobre el hijo izquierdo del nodo, y luego a la izquierda sobre el nodo original.
- rotacionIzquierdaDerecha(AVL*& nodo): Realiza una rotación doble. Primero a la izquierda sobre el hijo derecho del nodo, y luego a la derecha sobre el nodo original.

```
49    //Rotación para la izquierda y luego para la izquierda
50    void ArbolAVL::rotacionDerechaIzquierda(AVL*& nodo){
51        rotacionDerecha( nodo: [&]nodo->getIzquierda());
52        rotacionIzquierda( [&]nodo);
53    }
54    //Rotación para la derecha y luego para la derecha
55    void ArbolAVL::rotacionIzquierdaDerecha(AVL*& nodo){
56        rotacionIzquierda( nodo: [&]nodo->getDerecha());
57        rotacionDerecha( [&]nodo);
58    }
```

- rotacionDerecha(AVL*& nodo): Realiza una rotación simple a la derecha.
- rotacionIzquierda(AVL *&nodo): Realiza una rotación simple a la izquierda.

```
60  > void ArbolAVL::rotacionDerecha(AVL*& nodo){...}
81    //Rotación cuando está acumulado a la izquierda
82  > void ArbolAVL::rotacionIzquierda(AVL *&nodo){...}
```

Funciones para el cálculo de la altura y del factor de equilibrio

- alturaMaxima(AVL* nodo): Calcula la altura del subárbol con raíz en el nodo que se pasa.

- factorEquilibrio(AVL *nodo): Calcula el factor de equilibrio de un nodo.

```
104  int ArbolAVL::alturaMaxima(AVL* nodo){//Función para obtener la altura maxima
105      if (nodo==nullptr){//Si no tiene nada
106          return 0;//La altura es 0
107      }
108      //Busco la altura del lado derecho
109      int hIzquierda=alturaMaxima( nodo: nodo->getIzquierda());
110      //Busco la altura del lado izquierdo
111      int hDerecha=alturaMaxima( nodo: nodo->getDerecha());
112      //Retorno alguno de los dos
113      return hIzquierda>hDerecha ? hIzquierda +1 : hDerecha + 1;
114  }
115  int ArbolAVL::factorEquilibrio(AVL *nodo){//Función para obtener el factor de
116      if (nodo==nullptr) return 0;
117      int alturaDerecha = alturaMaxima( nodo: nodo->getDerecha());
118      int alturaIzquierda = alturaMaxima( nodo: nodo->getIzquierda());
119      return alturaDerecha - alturaIzquierda;
120  }
```

Funciones para la generación de graficas con graphviz

- generarDotGrafica(): Función principal que crea el archivo .dot con la estructura del árbol y luego ejecuta la función para generar el PDF.
- generarContenidoDot(AVL* nodo): Función recursiva que genera el código DOT para representar el arbol.

```
129  //*************************************************************************
130  //Funciones para generar la gráfica de un solo usuario
131  void ArbolAVL::generarDotGrafica(){...}//Función para generar el string .dot
144  // Nueva función para generar el contenido del dot
145  std::string ArbolAVL::generarContenidoDot(AVL* nodo){...}
168
169  //*************************************************************************
```

- generarDotContenidoActivos(): Similar a generarContenidoDot, genera el contenido DOT pero sin la estructura principal de la gráfica.

```
169  //*************************************************************************
170  //Función para generar el string .dot para la grafica de todos
171  std::string ArbolAVL::generarDotContenidoActivos(){
172      std::string graficaAVL="";
173      graficaAVL += generarContenidoDot(this->raiz);//mando la raiz
174      return graficaAVL;
175  }
176       Ctrl⇧L to Chat, Ctrl+I to Command
177  //*************************************************************************
```

Funciones para la eliminación de nodos

- eliminar(std::string valor): Función pública para eliminar un nodo con el ID especificado que llama a la función de eliminación recursiva.

- eliminar(std::string valor, AVL *&nodo): Función recursiva que busca el nodo a eliminar y lo elimina. Es llamada por la función pública.

```
178     //Función para eliminar pública
179     void ArbolAVL::eliminar(std::string valor){
180         eliminar( valor, this->raiz);
181     }
182     //Función que se iría llamando a sí misma
183     void ArbolAVL::eliminar(std::string valor, AVL *&nodo){...}
```

- balancear(AVL *nodo): Revisa el factor de equilibrio de un nodo y realiza las rotaciones necesarias si está desbalanceado. Esta función se llama después de la eliminación.
- masALaDerecha(AVL* nodo): Encuentra el nodo más a la derecha en un subárbol.
- esHoja(AVL* nodo): Determina si un nodo es una hoja o no.

```
240 void ArbolAVL::balancear(AVL *nodo){...}
258
259 AVL* ArbolAVL::masALaDerecha(AVL* nodo) {
260     // Encuentra el nodo más a la derecha.
261     if (nodo->getDerecha() == nullptr) return nodo;
262     return masALaDerecha( nodo: nodo->getDerecha());
263 }
264 //Función para saber si el último nodo
265 bool ArbolAVL::esHoja(AVL* nodo) {
266     return nodo->getIzquierda() == nullptr && nodo->getDerecha() == nullptr;
267 }
```

Funciones para la búsqueda de funciones

- buscar(std::string id): Función pública para buscar un nodo por su ID. Que llama a la función de búsqueda recursiva.
- buscar(std::string id, AVL* nodo): Función recursiva que realiza la búsqueda del nodo en el árbol y que es llamada por la función pública.

```
269  AVL* ArbolAVL::buscar(std::string id) {
270      return buscar( id, this->raiz); // Llama a la func
271  }
272  //Función para buscar un nodo
273  AVL* ArbolAVL::buscar(std::string id, AVL* nodo) {
274      if (nodo == nullptr) return nullptr;//Si no hay nada
275
276      if (id == nodo->getActivo()->getId()) {// Si el nodo
277          return nodo;
278      } else if (id < nodo->getActivo()->getId()) {// Ahor
279          return buscar( id, nodo: nodo->getIzquierda());
280      } else {
281          return buscar( id, nodo: nodo->getDerecha());//
282      }
283  }
```

Funciones para la impresión de información en la consola

- imprimirActivos(): Imprime la información de todos los activos en orden inordinado.
- imprimirRevursivo(AVL* nodo): Función recursiva que realiza el recorrido inorden para imprimir los activos.

```
286  //Funciones para imprimir activos
287  void ArbolAVL::imprimirActivos(){...}
290  //La función recursiva
291  void ArbolAVL::imprimirRevursivo(AVL* nodo){...}
```

- imprimirActivosDisponibles(): Imprime solo los activos que no están rentados.
- imprimirActivosDisponiblesRecursivo(AVL* nodo): Función recursiva que es llamada por la anterior función y que imprime los activos disponibles.

```
324  void ArbolAVL::imprimirActivosDisponibles(){...}
327  void ArbolAVL::imprimirActivosDisponiblesRecursivo(AVL* nodo){...}
```

- imprimirActivosNoDisponibles(): Imprime solo los activos que si están rentados.

- imprimirActivosNoDisponiblesRecursivo(AVL* nodo): Función recursiva que imprime los activos no disponibles y que es llamada por la función anterior.

```
342  void ArbolAVL::imprimirActivosNoDisponibles(){
343      imprimirActivosNoDisponiblesRecursivo(this->raiz);
344  }
345  void ArbolAVL::imprimirActivosNoDisponiblesRecursivo(AVL* nodo){
346      if (nodo == nullptr) return;//Si no tiene nada no imprimo nada xd
347      //Primero me voy por el lado izquierdo
348      imprimirActivosNoDisponiblesRecursivo( nodo: nodo->getIzquierda());
349
350      // Imprimo la información del nodo
351      if (nodo->getActivo()->getTiempoRenta() >0){
352          std::cout << "ID: " << nodo->getActivo()->getId()<< ", Nombre: " << no
353                << ", Descripcion: " << nodo->getActivo()->getDescripcion() <<",
354
355          // Por último me voy al lado derecho
356      }
357      imprimirActivosNoDisponiblesRecursivo( nodo: nodo->getDerecha());
358  }
```

- mostrarActivosRentados(): Lo mismo que la función imprimirActivosDisponibles() pero que se usa dentro de la matriz.
- imprimirActivosRentadosRecursivo(AVL* nodo): Lo mismo que la función imprimirActivosDisponiblesRecursivo pero para usarse dentro de la matriz dispersa.

```
362  void ArbolAVL::mostrarActivosRentados(){
363      imprimirActivosDisponiblesRecursivo(this->raiz);
364  }
365  void ArbolAVL::imprimirActivosRentadosRecursivo(AVL* nodo){
366      if (nodo == nullptr) return;//Si no tiene nada no imprimo nada xd
367      //Primero me voy por el lado izquierdo
368      imprimirActivosDisponiblesRecursivo( nodo: nodo->getIzquierda());
369
370      // Imprimo la información del nodo
371      if (nodo->getActivo()->getTiempoRenta() >0){
372          std::cout << "ID: " << nodo->getActivo()->getId()<< ", Nombre: " << nodo-
373                << ", Descripción: " << nodo->getActivo()->getDescripcion() << std:
374
375          // Por último me voy al lado derecho
376      }
377      imprimirActivosDisponiblesRecursivo( nodo: nodo->getDerecha());
378  }
```

Funciones para la verificación de activos rentados

- hayActivosRentados(): Función pública que verifica si existen activos rentados en el árbol llamando a la función recursiva.
- hayActivosRentadosRecursivo(AVL* nodo): Función recursiva que realiza la verificación que solicita la función anterior.

```
382  bool ArbolAVL::hayActivosRentados(){
383      return hayActivosRentadosRecursivo(this->raiz);
384  }
385  bool ArbolAVL::hayActivosRentadosRecursivo(AVL* nodo){
386      if (nodo==nullptr) return false;//Si no tiene nada no hago nada xd
387      if (nodo->getActivo()->getTiempoRenta() >0) return true;//Si no tiene nada no
388      //Primero me voy por el lado izquierdo
389      bool activoIzquierdo = hayActivosRentadosRecursivo( nodo: nodo->getIzquierda());
390      if (activoIzquierdo) return true;
391      // Por último me voy al lado derecho
392      bool activoDerecho =hayActivosRentadosRecursivo( nodo: nodo->getDerecha());
393      if (activoDerecho) return true;
394      return false;
395  }
```

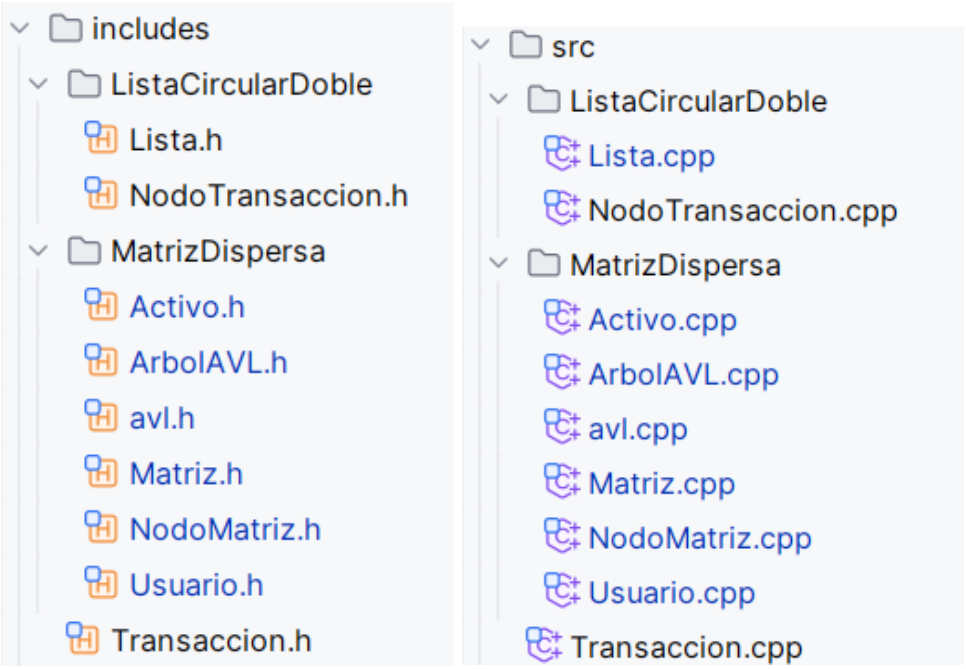
Funciones para la modificación de los valores de los nodos

- modificar(std::string id, std::string nuevoNombre, std::string nuevaDescripcion): Modifica el nombre y la descripción de un activo existente, buscándolo por su ID (el que no se modifica).

```
309  //Función para modificar nodo
310  void ArbolAVL::modificar(std::string id, std::string nuevoNombre,std::string n
311      AVL* nodoModificado = buscar( id);
312      if (nodoModificado != nullptr) {
313          if (nodoModificado->getActivo()->getTiempoRenta() >0){
314              std::cout << "No se puede modificar un activo en renta\n";
315          }else{
316              nodoModificado->getActivo()->setNombre( nuevoNombre);
317              nodoModificado->getActivo()->setDescripcion( nuevaDescripcion);
318          }
319      } else {
320          std::cout << "Nodo con ID " << id << " no encontrado\n";
321      }
322  }
```

Distintas Clases utilizadas

Como se mencionó previamente, se utilizaron *headers* para permitir un código de mejor calidad, por tal motivo, todas las clases, por consiguiente, las estructuras abstractas ya descritas, lo usan:



Las clases que se utilizaron a lo largo del proyecto para almacenar y representar de mejor manera la información son las siguiente:

Transaccion

```
9     //Constructor :)
10    Transaccion::Transaccion(std::string IdTransaccion,
11        this->idTransaccion = IdTransaccion;
12        this->idActivo = idActivo;
13        this->usuarioRentador = usuarioRentador;
14        this->departamento = departamento;
15        this->empresa = empresa;
16        this->fecha = fecha;
17        this->tiempo = tiempo;
18    }
```

Esta clase se utilizó para almacenar la información de todas las transacciones dentro del nodo que iría dentro de la lista circular doblemente enlazada.

Usuario

```
7   Usuario::Usuario(std::string usuario){
8       this->usuario = usuario;
9       this->contrasena = "";
10      this->activos=new ArbolAVL();
11      this->activosRentados=new Lista();
12      this->departamento="";
13      this->empresa = "";
14      this->nombre = "";
15  }
```

Dentro del programa Renta Activos, es esencial el manejo de usuario para que estos manejen los activos de manera óptima, por tal razón, esta clase permite representar cada uno de los usuarios dentro de la plataforma.

Activo

```
7   //Constructor
8   Activo::Activo(std::string id,std::string nombre,std::string descripcion){
9       this->id = id;
10      this->descripcion = descripcion;
11      this->nombre = nombre;
12      this->tiempoRenta = 0;
13      this->usuarioRentador="";
14  }
```

La principal función de la aplicación es la renta de activos, por tal motivo se creó una clase que permitiría almacenar la información de cada activo que los usuarios publiquen o renten.

main.cpp

Esta es la función principal dentro del código, desde esta es donde todo comienza, por tal motivo, esta contiene la mayoría de las funciones y ciclos que dan el flujo del programa. Esta tiene varios ciclos que permiten que se presenten los distintos submenús que permiten el flujo correcto del programa.

```
42  //Ciclo para mostrar el menu del usuario :)
43  void usuarioMenu(Usuario *usuario){...}
229 //-------------------------------------------------------------------------------
230 //Ciclo para mostrar el menu del administrador :(
231 void adminMenu(){...}
408 //-------------------------------------------------------------------------------
409 //Ciclo para el ingreso de datos cuando se seleccionó la opción para esto
410 void ingreso(){...}
434 //-------------------------------------------------------------------------------
435 //Aquí se va a encontrar el ciclo principal desde se comenzará a correr el programa
436 int main(){...}
```

Adicionalmente, contiene la función destinada a la generación del id alfanumérico de 15 caracteres:

```
23  //Función para crear un id alfanumerico aleatorio a partir de la hora :)
24  string asignarIdAlfanumerico(){
25      std::srand( Seed: time(0));
26      string idAlfanumerico = "";
27      for (int i=0; i<15;i++){ //Lo itero 16 veces para que sea de longitud de 16
28          int opcion = std::rand() % 2 + 1;
29              if (opcion==1){ //Si es 1, será alfabetico
30                  int letra=char(std::rand()%26+97);
31                  idAlfanumerico+=char(letra);
32              }else{
33                  int valor=char(std::rand()%10+48);
34                  idAlfanumerico+=char(valor);
35              }
36      }
37      //Retorno el id alfanumérico
38      return idAlfanumerico;
39  }
```

Con todo esto, se finaliza el manual técnico que se espera que permita entender de mejor manera el funcionamiento los métodos que se utilizaron la esta solución.