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

[ÍNDICE 1](#_Toc165147344)

[INTRODUCCIÓN 2](#_Toc165147345)

[OBJETIVOS 2](#_Toc165147346)

[1. GENERAL 2](#_Toc165147347)

[2. ESPECÍFICOS 2](#_Toc165147348)

[ALCANCES DEL SISTEMA 2](#_Toc165147349)

[ESPECIFICACIÓN TÉCNICA 3](#_Toc165147350)

[● REQUISITOS DE HARDWARE 3](#_Toc165147351)

[● REQUISITOS DE SOFTWARE 3](#_Toc165147352)

[DESCRIPCIÓN DE LA SOLUCIÓN 4](#_Toc165147353)

[LÓGICA DEL PROGRAMA 7](#_Toc165147354)

# 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

## GENERAL

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

## ESPECÍFICOS

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

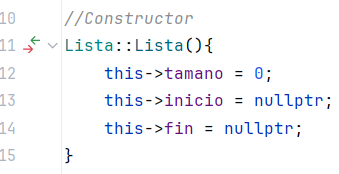
Para la creación del programa Renta Activos, se utilizaron varias estructuras utilizando las clases, usando convenciones para tener una mejor calidad de código creando *headers*, las cuales son:

## Lista Circular Doblemente enlazada

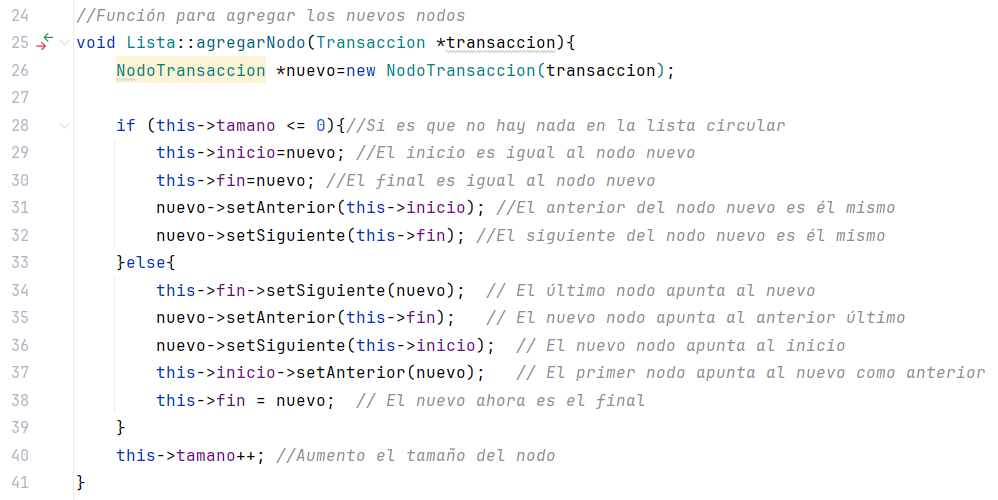
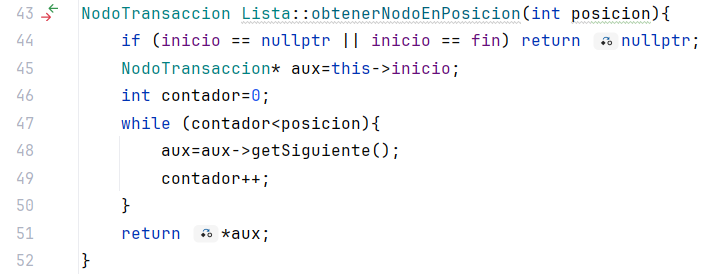
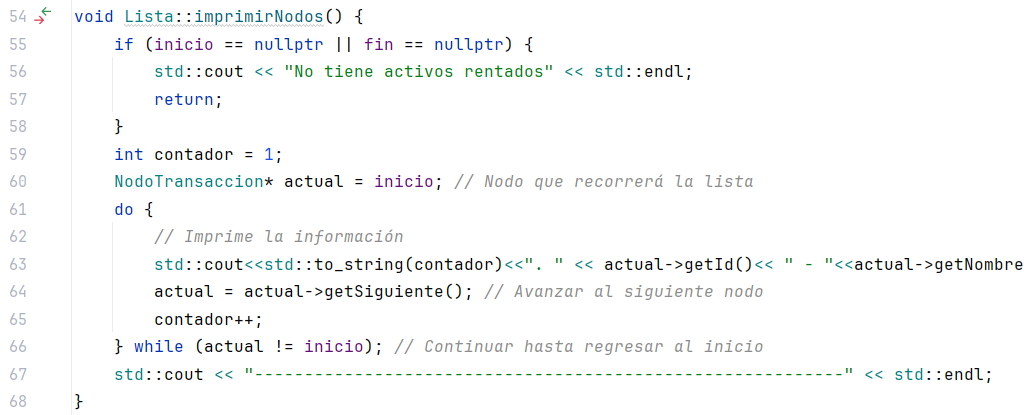
Una lista circular doblemente enlazada es una estructura de datos lineal en la que cada elemento, que se llama nodo, contiene, además de los datos, dos punteros: uno que referencia al nodo siguiente y otro que apunta al nodo anterior. En esta estructura, a diferencia de las listas normales, el último nodo apunta al primero, y el primero al último, formando así un círculo.

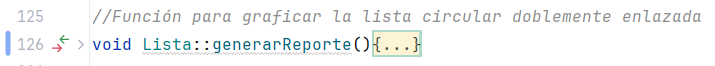
Para este proyecto, se usó este tipo de lista para guardar las transacciones que se harían durante la ejecución del programa Renta Activos y sus métodos principales son estos:

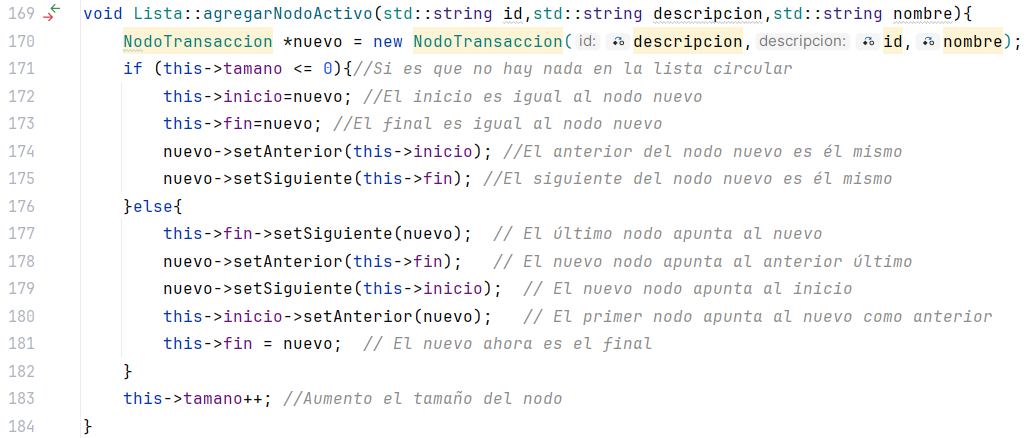
### Constructor

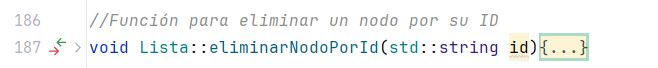
* **Lista()**: Inicializa una lista circular doblemente enlazada vacía, estableciendo los punteros inicio y fin en nullptr y el tamaño en 0. 

### 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. 
* **obtenerNodoEnPosicion(int posicion)**: Retorna un puntero al nodo que se encuentra en la posición especificada. Realiza una busqueda iterativa. 
* **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
* **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. 
* **ordenarDescendente()**: Igual a ordenarAscendente(), pero ordena la lista de forma descendente según el idTransaccion (básicamente solo cambio un símbolo) 
* **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.



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

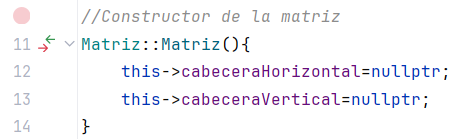


* g**enerarReporteActivosRentados(std::string user):** Genera un reporte gráfico DOT similar a generarReporte(), pero hecho para los activos rentados. 

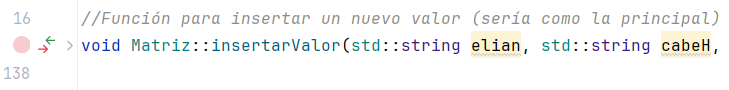
## Matriz Dispersa

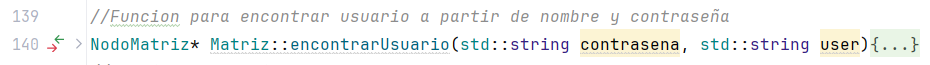
Una matriz dispersa es una matriz en la que la mayoría de sus elementos no ocupan espacio, lo que la distingue de un arreglo, en donde la mayoría de los elementos ya apartan especio de memoria. Para este proyecto se usó este tipo de matriz como base para guardar toda la información de los usuarios entre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

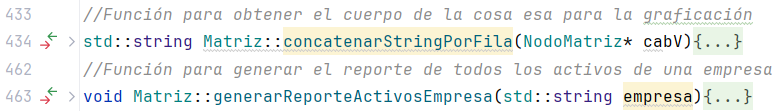
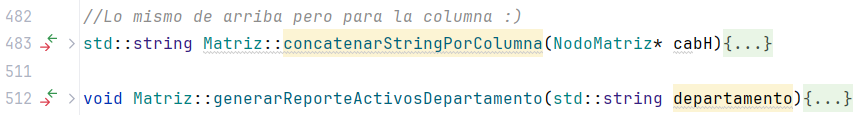
### Constructor

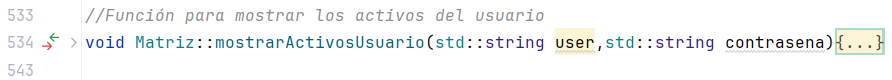
* **Matriz():** inicializa las cabeceras de la matriz con valor nulo. 

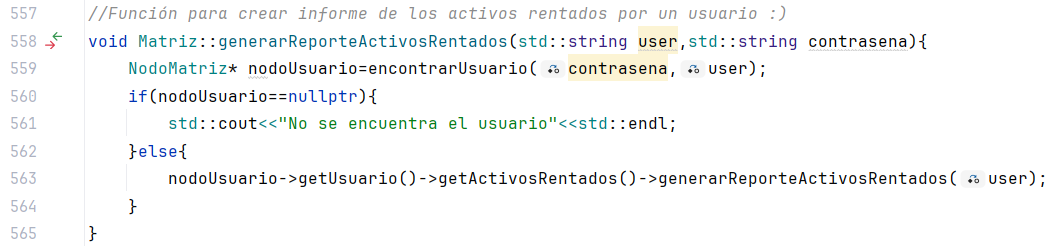
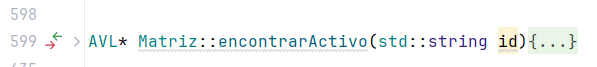
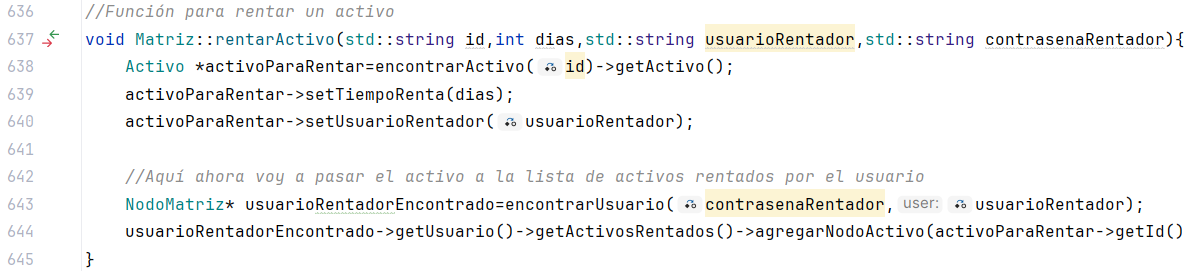
### Funciones principales

* **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). 
* **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.



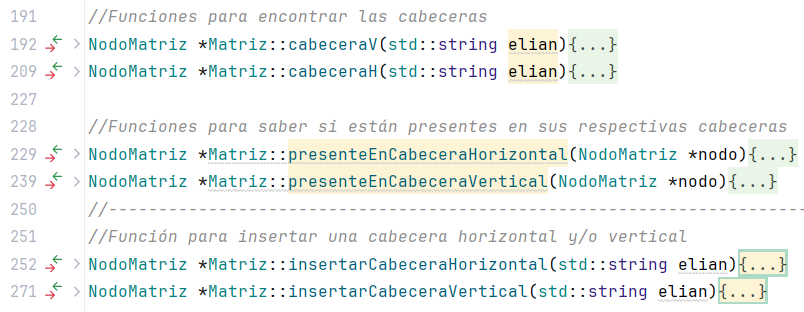
* **generarReporteMatriz() y generarDotGrafica():** Generan un reporte gráfico de la matriz en formato DOT, que luego se convierte a PDF con Graphviz. generarDotGrafica() es la función que crea la cadena de texto con el formato DOT, mientras que generarReporteMatriz() maneja la creación del archivo y la ejecución del comando dot. 
* **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. 
* **generarReporteActivosDepartamento(std::string departamento) y concatenarStringPorColumna(NodoMatriz\* cabH):** Similar a las anteriores, pero generan un reporte de los activos de un departamento específico. 
* **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.



* **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. 
* **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. 
* **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. 
* **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. 
* 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. 

### Funciones para el manejo de cabeceras en la matriz

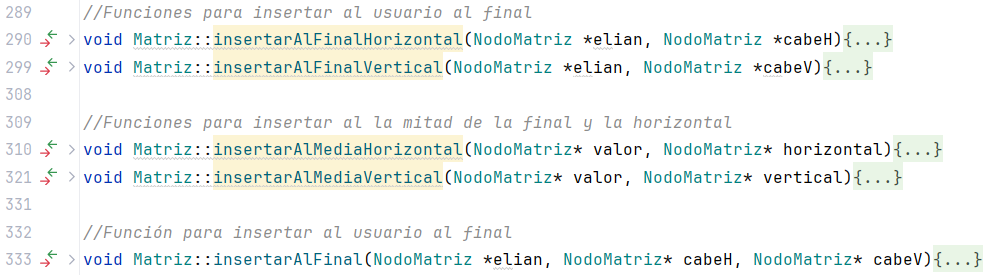
* **cabeceraV(std::string elian)**
* **cabeceraH(std::string elian)**
* **insertarCabeceraHorizontal(std::string elian)**
* **insertarCabeceraVertical(std::string elian)**
* **presenteEnCabeceraHorizontal(NodoMatriz \*nodo)**
* **presenteEnCabeceraVertical(NodoMatriz \*nodo)**

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

### Funciones para la insersión de nodos

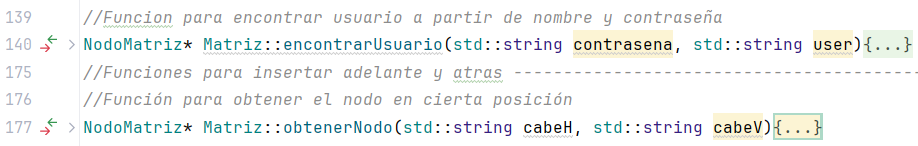
* insertarValor(std::string elian, std::string cabeH, std::string cabeV,std::string contra,std::string nombreCompleto)
* insertarAlFinalHorizontal(NodoMatriz \*elian, NodoMatriz \*cabeH)
* insertarAlFinalVertical(NodoMatriz \*elian, NodoMatriz \*cabeV)
* insertarAlMediaHorizontal(NodoMatriz\* valor, 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.



### Funciones para la búsqueda de Nodos, Activos y Usuarios

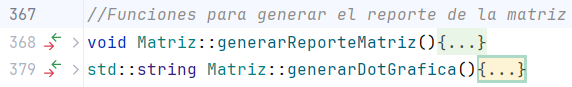
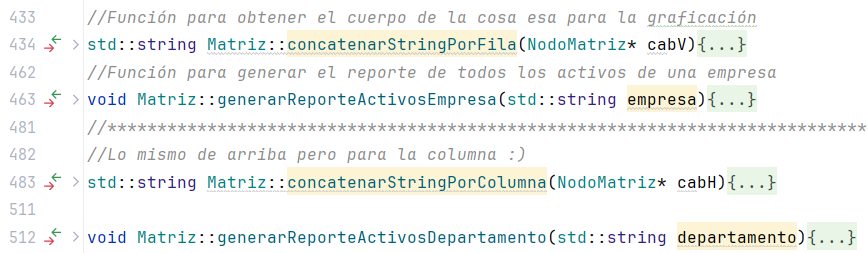
* encontrarUsuario(std::string contrasena, std::string user)
* obtenerNodo(std::string cabeH, std::string cabeV)
* encontrarActivo(std::string id) Estas funciones se encargan de buscar nodos, ya sean usuarios o activos, dentro de la estructura de la matriz.

### Funciones para la generación de repotes

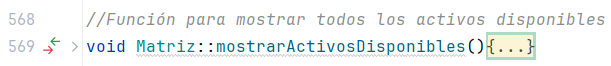
* generarReporteMatriz()
* generarDotGrafica()
* generarReporteActivosEmpresa(std::string empresa)
* concatenarStringPorFila(NodoMatriz\* cabV)
* generarReporteActivosDepartamento(std::string departamento)
* concatenarStringPorColumna(NodoMatriz\* cabH)
* generarReporteActivosRentados(...) (para usuarios)

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.

### 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.    

### Funciones para la renta de activos

* rentarActivo()
* hayActivosEnRenta()

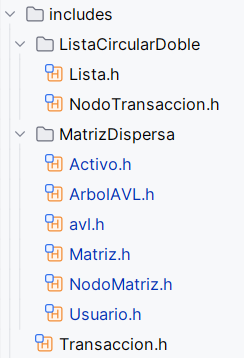
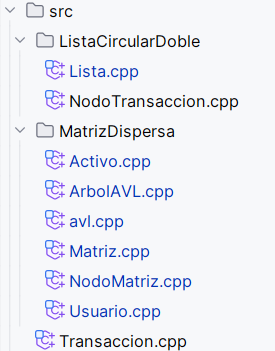




## ArbolAVL

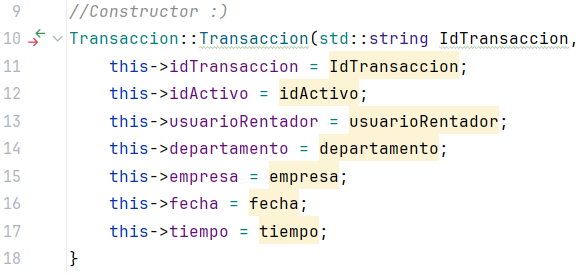
## Distintas Clases utilizadas

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:

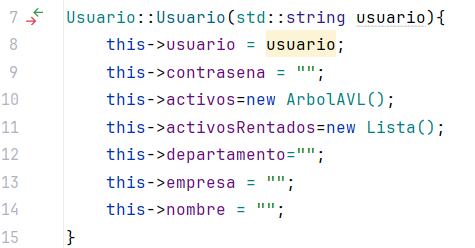
Las clases que se utilizaron a lo largo del proyecto para almacenar y representar de mejor manera la información son las siguiente:

### Transaccion



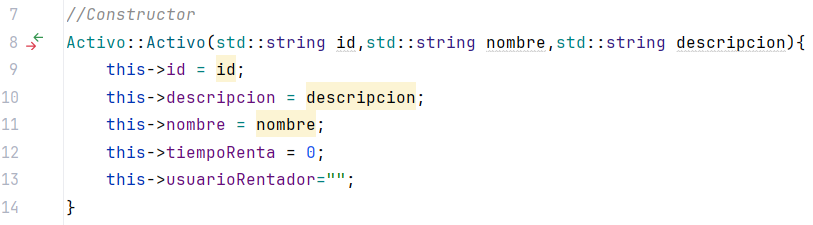
Esta clase se utilizó para almacenar la información de todas las transacciones dentro del nodo que iría dentro de la lista circular doblemente enlazada.

### Usuario



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



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 usuario 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.