MANUAL TÉCNICO
Fredy José Gabriel Herrera Funes
No. Registro Académico: 202130478.

-Descripción: programa dirigido para la gestión de una biblioteca mágica, en la que puedes realizar inserción masiva de libros, inserción manual, búsquedas, eliminaciones, comparación entre algoritmos, visualizar las estructuras utilizadas.

-Herramientas utilizadas:

- Desarrollado en sistema operativo Ubunto linux, versión 24.04.
- IDE utilizado: Visual Studio Code.
- Para la gráfica de los arboles se usó la herramienta Graphviz.
- Lenguaje de programación utilizado: C++.

-Estructuras utilizadas:

- Árbol AVL ordenado por ISBN y ordenado por Título(orden alfabético).
- Árbol B, ordenado según género de cada libro.
- Árbol B +, ordenado según el animo de cada libro.
- Lista enlazada no ordenada.

-Complejidades de cada método utilizado:

- **Clase GestorBiblioteca:**

`bienvenidaInicial()` — **O(1)**

Justificación: Solo imprime la bienvenida inicial y muestra el menú principal.

`menuBiblioteca()` — **O(1)**

Justificación: Solo muestra el menú principal.

`menuBiblioteca()` — **O(1)**

Justificación: Solo muestra el menú principal.

`menuFinAccion()` — **O(1)**

Justificación: Solo muestra el menú al finalizar cada acción realizada.

`realizarAccion(int& opcion)` — **O(1)**

Justificación: Solo llama a las función elegida en el menú principal.

`convertirISBN(string isbn)` — **O(1)**

Justificación: recorre el string para quitar guiones y convertirlo a un long long int.

`crearLibro(string linea)` — **O(n²)**

Justificación: Por las acciones realizadas en el codigo Linea.fin() y linea.erase().

`validarISBN(string isbn)` — **O(n)**

Justificación: recorre los caracteres para validar que tenga el numero de digitos correspondiente.

`cargarLibros(Libro* libro)` — **O(n)**

Justificación: recorre los caracteres para validar que tenga el numero de digitos correspondiente.

`buscarLibro()` — **O(n)**

Justificación: esto se debe a las posibles búsquedas secuenciales que se realizan en este método.

`leerArchivo()` — **O(n*C) Teniendo en cuenta que C es el coste que tiene al entrar en los métodos crearLibro() y cargarLibros().**

Justificación: se utiliza para cargar el archivo .csv y para guardar de forma correcta los datos del libro.

`validarOpcion(int minimo, int maximo)` — **O(n)**

Justificación: esto se debe a que si se equivoca el usuario metiendo una opción no válida se repetirá el bucle hasta que ingrese una opcion válida.

`mostrarLibros()` — **O(n)**

Justificación: esto se debe a los algoritmos que se implementaron al mostrar cada estructura.

`eliminarLibros()` — **O(n)**

Justificación: esto se debe a los algoritmos que se implementaron al eliminar cada estructura.

`generarGraphvizArboles()` — **O(n)**

Justificación: esto se debe a los algoritmos que se implementaron al generar cada graphviz cada estructura.

`buscarArbolB()` — **O(n)**

Justificación: esto se debe a los algoritmo utilizado para encontrar lo que se busca en el árbol b se debe a que este llena una lista con los datos de la búsqueda.

`buscarPorRangoAnio()` — **O(n)**

Justificación: esto se debe a los algoritmo utilizado para encontrar lo que se busca en el árbol b+ se debe a que este llena una lista con los datos de la búsqueda.

`compararTiempos()` — **O(1)**

Justificación: esto se debe a que solo se implementa el imprimir cada tiempo de búsqueda ya sea binaria o secuencial.

`cargarLibroManual()` — **O(n)**

Justificación: esto se debe a las validaciones que se realizan para verificar que los datos sean correctos.

- **Clase ListaLibro:**

`agregarLibro(Libro* libro)` — **O(n)**

Justificación: se debe a que se recorre en un ciclo toda la lista para insertar al final de la misma.

`eliminarLibro(string titulo)` — **O(n)**

Justificación: se debe a que recorre la lista buscando el libro que coincida con el título para luego eliminar ese nodo.

`buscarPorTitulo(string titulo)` — **O(n)**

Justificación: se debe a que recorre la lista buscando el libro que coincida con el título para luego retorna ese nodo.

`buscarPorIsbn(string titulo)` — **O(n)**

Justificación: se debe a que recorre la lista buscando el libro que coincida con el isbn para luego retorna ese nodo.

`imprimirLista()` — **O(n)**

Justificación: se debe a que recorre la lista imprimiendo los datos que contiene cada nodo.

`estaVacia()` — **O(1)**

Justificación: se debe a que solo compara que el tamaño no sea cero.

- **Clase ArbolAVL:**

`rotacionDD(),rotacionII(),rotacionID(),rotacionDI()` — **O(1)**

Justificación: se debe a que solo se realiza la un enlace de punteros y se actualiza la altura y factor de equilibrio.

`obtenerAltura(NodoAVL*)` — **O(1)**

Justificación: solo devuelve un valor almacenado en el nodo.

`obtenerFactorDeEquilibrio(NodoAVL*)` — **O(1)**

Justificación: solo llama a obtenerAltura() para realizar la actualizacion del factor de equilibrio.

`insertarRecursivoPorTitulo,insertarRecursivoPorIsbn` — **O(n)**

Justificación: se obtiene del recorrido que puede llegar a hacer desde la raíz hasta la hoja para la inserción.

`insertarPorTitulo,insertarPorIsbn` — **O(n)**

Justificación: ya que llama una función recursiva y asigna una raíz.

`buscarRecursivoPorTitulo,buscarRecursivoPorIsbn` — **O(n)**

Justificación: se realiza una busqueda binaria por propiedades del BST.

`buscarPorTitulo,buscarPorIsbn` — **O(n)**

Justificación: ya que llama a la función recursiva empezando por la raíz.

`eliminarRecursivoPorTitulo,eliminarRecursivoPorIsbn` — **O(n)**

Justificación: se debe a que se realiza una búsqueda primero para encontrar el nodo que se debe eliminar, reasigna y realiza rotaciones.

`eliminarPorTitulo,eliminarPorIsbn` — **O(n)**

Justificación: asigna raíz al resultado del recursivo.

`inOrdenRecursivo,inOrden` — **O(n)**

Justificación: recorre todos los nodos.

- **Clase arbol B:**

`buscar(string genero, ListaLibro resultados)` — **O(log(n))**

Justificación: ya que es un árbol balanceado la altura del árbol es proporcional a log (m) de n, donde m es el orden del árbol, entonces cómo se utiliza una búsqueda árbol binaria la complejidad es logarítmica.

`insertar(Libro* libro)` — **O(log(n))**

Justificación: debido a la búsqueda con complejidad O(log(n)) y división siendo esta última dependiendo de las circunstancias, la complejidad se eleva a logarítmica.

`eliminar(const string genero, Libro* libro especifico)` — **O(log(n))**

Justificación: debido a la búsqueda que se realiza primero y como ya vimos que la búsqueda es de complejidad O(log(n)).

`inOrden()` — **O(n)**

Justificación: debido a los ciclos que esta recorre al imprimir todos los nodos.

- **Clase nodoB:**

`inOrden()` — **O(n)**

Justificación: debido a los ciclos que esta recorre al imprimir todos los nodos.

`buscarPorGenero(const string&, ListaLibro&)` — **O(n)**

Justificación: recorre cada clave en el nodo y llama recursivamente a los hijos.

`insertarNoLleno(Libro* libro)` — **O(log(n))**

Justificación: al buscar la posición en la que se insertará, al chequear que su hijo esté lleno y en dado caso dividir y recurrir a un hijo, la complejidad se vuelve logarítmica.

`dividir(int indice, nodoB* hijo)` — **O(n)**

Justificación: ya que se copian n cantidad de claves y de hijos al nuevo nodo, se redimensiona vectores y se hace la inserción en hijos, se puede decir que se realizan operaciones lineales.

`encontrarLlave(const string &genero)` — **O(n)**

Justificación: realiza una búsqueda secuencial dentro del nodo comparando cada clave hasta encontrar coincidencia; el número de comparaciones crece linealmente con la cantidad de claves en el nodo.

`fusionar(int idx)` — **O(n)**

Justificación: combina todas las claves y los hijos de dos nodos en uno solo, moviendo elementos y eliminando referencias, lo cual requiere un recorrido completo de las listas de claves e hijos.

`prestarDeAnterior(int idx)` — **O(n)**

Justificación: inserta un elemento al inicio del vector llaves, lo que provoca un desplazamiento lineal de los elementos; además mueve un hijo si no es hoja, resultando en tiempo lineal.

`prestarDeSiguiente(int idx)` — **O(n)**

Justificación: borra el primer elemento del vector del hermano derecho (erase(begin())), operación que desplaza todos los elementos siguientes, resultando en complejidad lineal.

`llenar(int idx)` — **O(n)**

Justificación: llama a prestarDeAnterior, prestarDeSiguiente o fusionar, las cuales tienen complejidad lineal en el tamaño del nodo.

`eliminarDeHoja(int idx)` — **O(n)**

Justificación: elimina un elemento del vector llaves usando erase(begin() + idx), lo que desplaza los elementos posteriores una posición, generando coste lineal.

`obtenerPredecesor(int idx)` — **O(log(n))**

Justificación: desciende recursivamente por el hijo derecho hasta llegar a una hoja, recorriendo la altura del árbol, que es logarítmica respecto al número total de claves.

`obtenerSucesor(int idx)` — **O(log(n))**

Justificación: similar al predecesor, recorre la altura del árbol descendiendo por el hijo izquierdo hasta la hoja, siendo el número de niveles proporcional a log(n).

`eliminarDeNoHoja(int idx)` — **O(log(n))**

Justificación: puede reemplazar la clave por su predecesor o sucesor (cada uno de los cuales requiere recorrer la altura del árbol) o fusionar nodos, lo cual implica un coste logarítmico en el peor caso.

`eliminar(const string &genero, Libro libroEspecifico)*` — **O(log(n))**

Justificación: recorre el árbol descendiendo por un solo camino de longitud igual a la altura del árbol (log n) y realiza operaciones locales (buscar, eliminar o fusionar) en cada nivel, cuyo coste es constante o lineal en el grado del nodo, manteniendo la complejidad logarítmica global.

- **Clase ArbolB+:**

`encontrarHoja(Nodo nodo, int key)*`— **O(log(n))**

Justificación: desciende desde la raíz hasta una hoja (altura h≈log n). En cada nodo hace un while lineal sobre nodo->keys (O(m)). Si se considera m constante, queda O(log n).

`insertarEnHoja(Nodo hoja, Libro libro)**`— **O(m)**

Justificación: usa lower_bound (O(log m)) para encontrar posición, pero luego realiza vector::insert y/o vector::push_back y records.insert que desplazan/copian elementos en el vector (operaciones lineales en el número de claves del nodo → O(m)). Si m constante, se trata como O(1) amortizado por nodo.

`dividirHoja(Nodo hoja)*`— **O(m)**

Justificación: crea un nuevo nodo y copia la mitad superior de keys y records (≈m/2 elementos), borra el rango en la hoja original y ajusta punteros next. Las operaciones de asign/erase sobre vectores son lineales en la cantidad movida.

`dividirInterno(Nodo node)*`— **O(m)**

Justificación: copia las claves y children del lado derecho al nuevo nodo y reduce los vectores del nodo original; las operaciones assign/erase mueven O(m) elementos (m = #claves por nodo).

`insertarRecursivo(Nodo node, int key, Libro libro, ...)**`— **O(log(n))**

Justificación: desciende recursivamente hasta hoja (altura h), en cada nivel realiza búsquedas lineales en keys (O(m)), posibles inserciones en vectores (O(m)) y, si hay split, llama a dividirHoja o dividirInterno (O(m)). Con m constante, el coste dominante es la altura: O(log n).

`insertarRecursivo(Nodo node, int key, Libro libro, ...)**`— **O(log(n))**

Justificación: desciende recursivamente hasta hoja (altura h), en cada nivel realiza búsquedas lineales en keys (O(m)), posibles inserciones en vectores (O(m)) y, si hay split, llama a dividirHoja o dividirInterno (O(m)). Con m constante, el coste dominante es la altura: O(log n).

`insertar(Libro libro)*`— **O(log(n))**

Justificación: llama a insertarRecursivo; en caso de split en la raíz crea una nueva raíz (O(1)). Por lo tanto la complejidad es la del recursivo: O(h·m) → O(log n) con grado fijo.

`recolectarRangoDesdeHoja(Nodo hoja, int anioInicial, int anioFinal, ListaLibro &resultados)*`— **O(n)**

Justificación: recorre hojas enlazadas desde hoja, para cada clave en cada hoja añade sus registros a resultados. Si devuelve k libros, el coste es proporcional a la cantidad de claves y registros visitados; si el rango cubre todo el árbol es O(n).

`buscar(int anioInicial, int anioFinal, ListaLibro &resultados)`— **O(log(n+k))**

Justificación: localiza la hoja inicial con encontrarHoja (O(h·m) ≈ O(log n)); puede ajustar/avanzar en hojas hasta la primera clave ≥ inicio (coste pequeño) y luego llama a recolectarRangoDesdeHoja (coste O(k) para recolectar k resultados). En conjunto: O(log n + k).

`eliminarPorTitulo(const string &titulo)`— **O(n)**

Justificación: recorre todas las hojas desde la más a la izquierda hasta el final (puede visitar cada hoja y cada registro) buscando el título; además los erase en vectores (tanto en records[i] como en keys/records) son operaciones que desplazan elementos, por lo que en el peor caso toca y modifica una cantidad lineal de datos.

`imprimirArbol()`— **O(n)**

Justificación: hace un recorrido por niveles (BFS) y emite las claves de cada nodo; cada nodo y cada clave se procesa una vez.

`imprimirHojas()`— **O(n)**

Justificación: baja hasta la primera hoja y sigue `next` a través de todas las hojas, imprimiendo cada clave; visita todas las claves hoja exactamente una vez.

-TADS:

TAD: ListaLibro:

- `crear()` → **ListaLibro**
- `agregarLibro(libro)`
- `eliminarLibro(titulo)`
- `imprimirLista()`
- `buscarPorIsbn(isbn)` → **Libro | NULL**
- `buscarPorTitulo(titulo)` → **Libro | NULL**
- `estaVacia()` → **bool**
- `getTamanio()` → **int**

TAD ArbolAVL:

- `crear()` → **ArbolAVL**
- `insertarPorIsbn(isbn, libro)`
- `insertarPorTitulo(titulo, libro)`
- `buscarPorIsbn(isbn)` → **NodoAVL* | NULL**
- `buscarPorTitulo(titulo)` → **NodoAVL* | NULL**
- `eliminarIsbn(isbn)`
- `eliminarTitulo(titulo)`
- `inorden()`
- `generarGraphviz(archivoDOT, imagen)`

TAD ArbolB:

- crear(grado) → ArbolB
- buscar(genero, resultados)
- insertar(libro)
- eliminar(genero, libroEspecifico)
- inOrden()
- generarGraphviz(dot, imagen)

TAD ArbolBPlus:

- crear(orden) → ArbolBPlus
- insertar(libro)
- buscar(anioInicial, anioFinal, resultados)
- eliminarPorTitulo(titulo) → bool
- imprimirArbol()
- imprimirHojas()
- generarGraphviz(dot, imagen)