Bocu, Bogotá Cultural

Deivid Farid Ardila Herrera, Diego Alejandro Arevalo Arias, Angel David Beltran García, Cristofer Damian Camilo Ordonez Osa.

No. de equipo de trabajo: *B*

I. Introducción

El presente documento define las características del proyecto a desarrollar, describiendo la problemática, los tipos de usuario, los requerimientos funcionales, su interfaz de usuario, los entornos de operación y desarrollo, las pruebas de complejidad para las estructuras de datos lineales y no lineales, y el prototipo final. Así mismo, se establecen los roles y tareas que tendrá cada integrante del equipo durante la elaboración del producto de software.

El problema adjunto al proyecto reside en la dificultad de los pequeños y jóvenes artistas para hacerse visibles en el ciberespacio y -principalmente- en Bogotá, lo cual propicia la decadencia cultural, artística y social de la ciudad capitalina, que ha tenido su génesis en diversos dilemas políticos y sociales.

Por otro lado, se tomarán en cuenta, esencialmente, tres tipos de usuario: invitado, que solo podrá navegar en la página principal donde encontrará diversas actividades organizadas; usuario registrado, que tendrá una cuenta propia personalizable en la que podrá guardar sus preferencias; y expositor, que será capaz de crear eventos o actividades, especificando la información correspondiente.

El producto de software tendrá las siguientes funcionalidades: inicio y cierre de sesión; creación de una cuenta; selección de eventos favoritos; filtrado de eventos; y acceso a la página principal, tendencias, perfil y configuración. Las funcionalidades dependen del tipo de usuario y se muestran en la interfaz definida en el punto V.

El proyecto será elaborado principalmente en Android Studio, un entorno de desarrollo para aplicaciones móviles; de la misma forma, se usará como lenguaje base Java, dada la comodidad que le aporta a los integrantes del equipo.

Esta versión contempla estructuras de datos lineales y no lineales, a saber: colas, pilas, listas ordenadas, listas desordenadas, árboles BST, árboles AVL, montículos, conjuntos disjuntos, tablas hash y grafos.

El prototipo en todas sus versiones se encontrará en un repositorio de GitHub, con el fin de tener un óptimo control de las versiones del proyecto.

II. descripción del problema a resolver

La evidente y creciente interconexión digital ha facilitado profundamente la difusión casi irrestricta e instantánea de la información; actualmente, es posible enterarse de lo que sucede en cualquier parte del mundo, los datos viajan de persona a persona, aparentando no tener ningún tipo de traba. Sin embargo, esto no implica que los pequeños y medianos artistas de toda índole puedan tener visibilidad en el amplio y saturado universo del internet, solo aquellos con el suficiente poder, influencia y capital tienen una alta probabilidad de conseguir un lugar privilegiado en el ciberespacio, sin importar la utilidad y la idoneidad del contenido que aportan.

Una cantidad significativamente grande de información cultural provechosa se pierde constantemente. La publicidad solo es para aquellos con el dinero suficiente para pagarla, es un negocio que degrada la libre circulación del arte.

El presente proyecto busca dar a conocer a los artistas emergentes que no tienen una proyección alentadora en el ciberespacio, mediante la presentación de los eventos de los cuales son partícipes y/o organizadores.

Por otro lado, Bogotá -ciudad en la que se enmarca la problemática- fue catalogada en los 80's por José Maria Vergara [1] como "la Atenas de Sudamérica", dada su extensa oferta cultural y su destacado grupo de intelectuales, entre los que se encontraba Gabriel García Márquez, galardonado con el premio Nobel de Literatura. No obstante, con el pasar de los años se presentaron diversos problemas políticos y otras circunstancias sociales que deterioraron el producto cultural de la zona, haciendo que el arte pasará a un tercer plano.

De esta forma, el actual proyecto intenta también mejorar el decreciente panorama cultural de Bogotá mediante la publicitación de sus eventos artísticos y sociales, así como de sus jóvenes artistas.

III. usuarios del producto de software

A continuación se encuentran los diversos roles que puede tener un usuario en la aplicación.

Invitado: será capaz de ingresar a la app y navegar en la página principal donde encontrará diversos eventos y distintas funcionalidades, como filtrar y consultar las diferentes actividades de la ciudad. También podrá crear una cuenta de tipo individual o de tipo expositor en el apartado *cuenta*.

Usuario registrado: será capaz de ingresar a la app y en el apartado de *cuenta* editar su información; podrá añadir sus temas preferidos, tendrá acceso a una página principal personalizada de acuerdo a sus intereses, y le será posible buscar, filtrar y consultar las diversas actividades de la ciudad. Además podrá guardar sus actividades favoritas en una sección específica.

Expositor: podrá ingresar a la aplicación, editar su cuenta, navegar en la página principal, y tanto filtrar como consultar las diversas actividades de la ciudad. Además, será capaz de crear actividades en las cuales tendrá que especificar el nombre, tipo, costo, ubicación, fecha, horario, entre otra información; con el fin de que los demás usuarios de la app puedan conocer sobre estas.

IV. requerimientos funcionales del software

1. Inicio de sesión

Descripción: al entrar al apartado *cuenta*, se mostrará la opción de iniciar sesión. El usuario podrá acceder si ya posee una cuenta en la aplicación.

Acción iniciadora: oprimir el botón *Iniciar sesión*.

Comportamiento esperado: el sistema deberá validar la información de inicio de sesión; si la información coincide con la registrada, se abrirá una ventana que muestra la información del usuario, en caso contrario, se le indicará al usuario que los datos proporcionados no son correctos.

Requerimientos funcionales:

- *Consulta de datos:* se debe buscar el nombre de usuario y validar la contraseña en la estructura de datos correspondiente.
- *Validación de información:* si el usuario existe, se valida la contraseña. Si la contraseña coincide con la almacenada, se ingresa a la página de inicio; de lo contrario, se muestra una ventana para informar que el usuario o la contraseña son incorrectos.

2. Crear cuenta

Descripción: si el usuario desea crear una cuenta, deberá ingresar la información pertinente para acceder a la aplicación.

Acción iniciadora: presionar el botón *Crear cuenta*.

Comportamiento esperado: el sistema mostrará la opción de registrarse como usuario o como artista, después, abrirá los campos necesarios para ingresar la información de registro; una vez el usuario ingrese sus datos de registro, el sistema los validará y almacenará, para finalmente dar ingreso a la aplicación.

Requerimientos funcionales:

- *Validación de datos:*
    - *Usuario:* para crear una cuenta como usuario, se deberá ingresar: nombre, fecha de nacimiento, correo, localidad de residencia, contraseña, confirmación de contraseña e intereses.
    - *Artista:* para crear una cuenta como artista, se deberá ingresar: nombre del usuario u organización, correo, contraseña, confirmación de contraseña y tipo de eventos.

  Al ingresar la información anterior y dar clic en el botón *Aceptar*, el sistema debe corroborar los datos con base en los siguientes criterios:

    - Se deben llenar todos los campos.
    - La contraseña debe tener como mínimo ocho caracteres
    - La dirección de correo electrónico debe ser válida.
    - Los campos *Contraseña* y *Confirmar contraseña* deben contener la misma información.
- *Creación de datos:* al crear una cuenta, se crea un objeto de la clase *Usuario Registrado* para manipular la información del nuevo usuario.
- *Almacenamiento:* se guarda la información del nuevo usuario en una base de datos creada localmente en el dispositivo.

3. Ingreso como invitado

Descripción: si el usuario no desea crear una cuenta, deberá ingresar como invitado en la aplicación.

Acción iniciadora: oprimir el botón *Ingresar como invitado*.

Comportamiento esperado: el sistema deberá cambiar a la ventana de inicio. Sin embargo, el usuario invitado tendrá funcionalidades acortadas en comparación con un usuario registrado.

Requerimientos funcionales:

- *Validación de datos:* el sistema deberá mostrar, únicamente, las páginas *inicio* y *descubrir*,

4. Página de Inicio

Descripción: al momento de acceder a la página de inicio, si el usuario lo prefiere, el sistema mostrará la información relevante de los eventos más próximos.

Acción iniciadora: entrar a la aplicación o presionar el boton *Inicio*.

Comportamiento esperado: el sistema deberá cargar una lista con los eventos más próximos (si el usuario lo prefiere). El usuario podrá acceder a más información presionando cualquier evento.

Requerimientos funcionales:

- *Creación:* en Android Studio se necesitará de un adaptador para crear una lista con los datos de los eventos.
- *Consulta de datos:* se debe recuperar la información de los eventos en la base de datos para mostrarlos en pantalla.
- *Ordenamiento:* los eventos que estén más cerca a la fecha del dispositivo deben aparecer al principio de la lista solo si el usuario lo prefiere, en otro caso, la lista aparecerá desordenada.

5. Acceso a categorías de eventos

Descripción: para buscar algún evento en específico, se puede simplificar el proceso entrando a alguna categoría de eventos.

Acción iniciadora: entrar a la sección *Descubrir* en la aplicación.

Comportamiento esperado: se va a mostrar una lista de categorías. Si el usuario presiona alguna de las categorías, se va a efectuar el mismo proceso que realiza la página de inicio para mostrar los eventos, sin embargo, en este caso sólo van a aparecer los eventos que concuerden con la categoría elegida.

Requerimientos funcionales:

- *Creación:* en Android Studio se necesitará de un adaptador para crear una lista con los datos de los eventos.
- *Consulta de datos:* se debe recuperar la información de los eventos en la base de datos para mostrarlos en pantalla. Solo se deben obtener los eventos que concuerden con la categoría de evento seleccionada por el usuario.
- *Ordenamiento:* si el usuario lo decide, los eventos que estén más cerca a la fecha del dispositivo deben aparecer al principio de la lista. En caso contrario, se van a mostrar de forma desordenada.

6. Acceso a eventos

Descripción: para ver la información adicional de un evento, se puede presionar cualquier parte del mismo.

Acción iniciadora: presionar el recuadro del evento.

Comportamiento esperado: se muestra un *AlertDialog* con toda la información del evento.

Requerimientos funcionales:

- *Consulta de datos:* se debe recuperar la información del evento en la estructura de datos para mostrarla en pantalla.

7. Página de cuenta

Descripción: el usuario o artista puede revisar la información suministrada a la aplicación mediante la sección de cuenta.

Acción iniciadora: presionar el botón *Cuenta*.

Comportamiento esperado: se debe iniciar una *Activity* que muestre la información almacenada del usuario.

Requerimientos funcionales:

- *Actualización:* si el usuario desea cambiar algún dato de la cuenta, se deben habilitar los campos correspondientes; una vez el usuario ingresa la nueva información, esta se debe validar y reemplazar donde corresponde.
- *Consulta de datos:* para mostrar la información relevante de la cuenta, se debe acceder a la estructura de datos mediante el correo o nombre del usuario. Con la información obtenida se crea un objeto de la clase *Usuario Registrado*.
- *Almacenamiento:* para el almacenamiento y actualización de datos, se maneja la base de datos ubicada en el dispositivo.

8. Página de eventos - Artista

Descripción: el artista posee una sección especial para crear, modificar o eliminar eventos.

Acción iniciadora: presionar el botón *Eventos*.

Comportamiento esperado: se debe mostrar una lista con los eventos que ha creado el artista.

Requerimientos funcionales:

- *Creación:* en Android Studio se necesitará de un adaptador para crear una lista que contenga los datos de los eventos, con un filtro que indique el artista que lo creó. Para crear un nuevo evento, se oprime el botón *Crear*, que llevará a una nueva Activity, la cual pedirá: nombre del evento, descripción, categoría del evento, localidad dónde se va a realizar el evento, dirección específica, fecha, hora y costo.
- *Validación de datos:* para crear un evento, se deben considerar los siguientes criterios: se deben llenar todos los campos, el nombre del evento debe ser de máximo 50 caracteres, y la descripción debe ser de máximo 500 caracteres. Al corroborar lo anterior, el sistema almacena la información en la base de datos y en la estructura correspondiente.
- *Consulta de datos:* se debe recuperar la información de los eventos creados por el artista en la base de datos para crear un objeto de la clase *Evento* y así mostrarlo en pantalla.
- *Almacenamiento:* para el almacenamiento, actualización y eliminación de eventos, se maneja la base de datos ubicada en el dispositivo.
- *Actualización:* si el artista desea cambiar la información del evento, se debe consultar dicho evento en la base de datos para mostrarlo en pantalla. Luego de hacer los cambios, se deben corroborar las excepciones para acceder a la base de datos y modificar la información.
- *Ordenamiento:* los eventos creados por el artista se muestran del más reciente al más antiguo, sólo si el artista así lo prefiere; en caso contrario, los eventos creados no se muestran de forma organizada

9. Filtros.

Descripción: cualquier tipo de usuario puede ingresar filtros de búsqueda para encontrar eventos con características específicas.

Acción iniciadora: presionar el botón *Aplicar filtros*.

Comportamiento esperado: se debe inicializar un *Fragment* que muestre todos los eventos que coincidan con la descripción suministrada por el usuario (usuario registrado, invitado o artista).

Requerimientos funcionales:

- *Actualización:* si el usuario desea aplicar nuevos filtros, se debe permitir regresar al *Fragment* inicial al presionar el botón *Filtrar*. Después de ingresar los nuevos filtros el usuario podrá volver a presionar el botón *Aplicar filtros* y verá todos los eventos que coincidan con la nueva descripción suministrada.
- *Consulta de datos:* para mostrar los eventos filtrados, primero es necesario obtener todos los eventos y luego aplicar métodos especializados utilizando estructuras de datos para realizar el filtrado.

- *Almacenamiento:* todos los eventos marcados como "favoritos" son guardados en una *DynamicUnsortedList* independiente, la cual es actualizada cada vez que el usuario decide guardar o eliminar un evento.

10. Ordenamiento:

Descripción: el usuario o artista puede ordenar los eventos filtrados de acuerdo a su preferencia:

- Orden alfabético: A_Z o Z_A.
- Costo: Mayor a Menor o Menor a Mayor.
- Fecha: Más recientes o Más antiguos.

Acción iniciadora: presionar el botón *menú kebab* y seleccionar la opción de ordenamiento que se desea.

Comportamiento esperado: se debe actualizar el *Fragment* para que muestre todos los eventos ordenados, según la opción seleccionada por el usuario.

Requerimientos funcionales:

- *Actualización:* el *Fragment* debe actualizarse cargando los eventos ordenados, con base en la elección del usuario.

11. Favoritos:

Descripción: el usuario común puede guardar eventos que le hayan interesado-gustado, ya sea para poder acceder a ellos de manera más rápida en futuras ocasiones o por cualquier motivo personal.

Acción iniciadora: presionar el botón en forma de corazón al momento de revisar un evento.

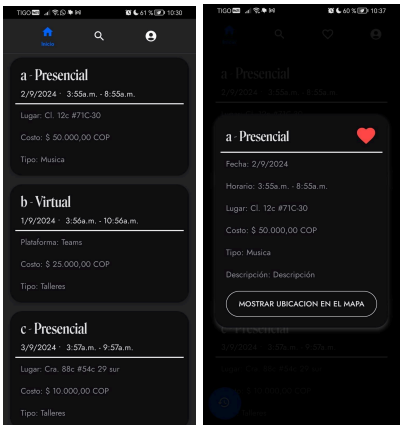
Comportamiento esperado: el evento debe ser guardado y asociado al usuario en la base de datos. Estos eventos asociados son mostrados en la página *Favoritos*. El usuario puede marcar o desmarcar eventos como favoritos a voluntad. Los eventos que han sido desmarcados, deben dejar de aparecer en la sección *Favoritos*.

Requerimientos funcionales:

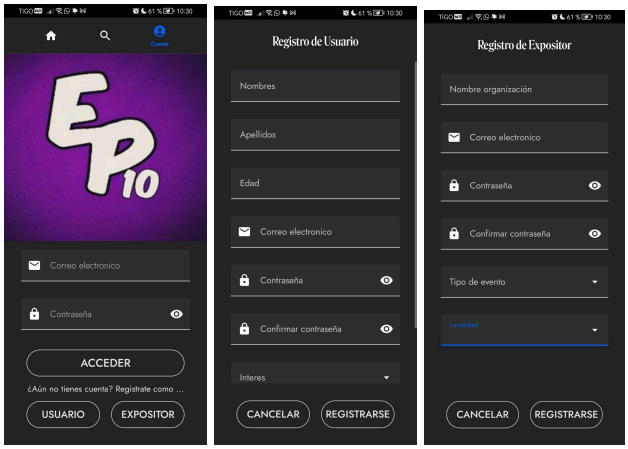
- *Actualización:* si el usuario desea guardar o eliminar un evento favorito, se debe permitir recargar, en caso de estar en el *Fragment Favoritos*, los eventos a mostrar. Si el usuario no se encuentra en el *Fragment Favoritos*, debe actualizarse la información a mostrar la siguiente vez que el usuario decida ingresar al *Fragment*.
- *Consulta de datos:* para mostrar los eventos marcados como favoritos es necesario consultar todos los eventos que hayan sido almacenados con esta condición.
- *Almacenamiento:* todos los eventos favoritos son guardados en una *DynamicUnsortedList* independiente.

V. DESCRIPCIÓN DE LA INTERFAZ DE USUARIO

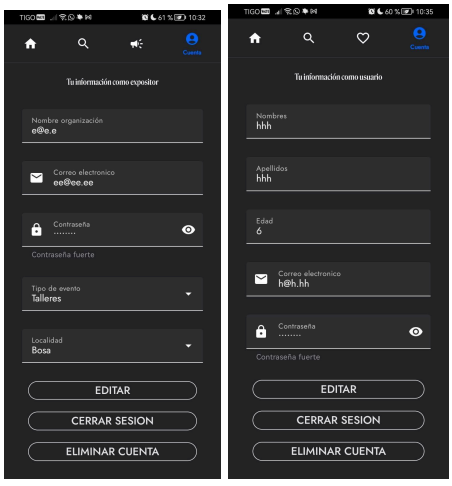
Página principal y vista eventos:



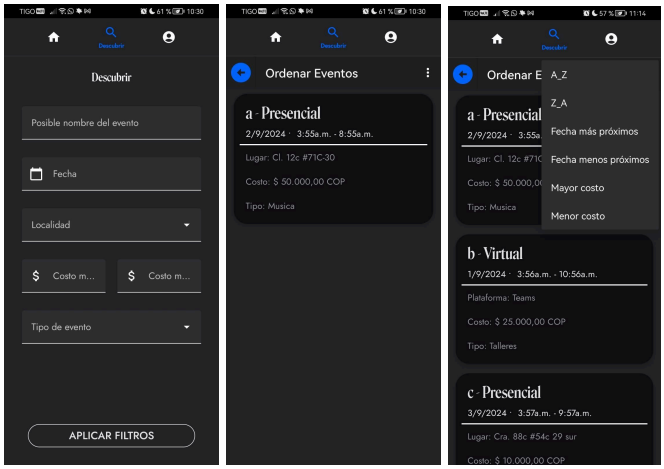
Inicio de sesión y creación de cuenta:



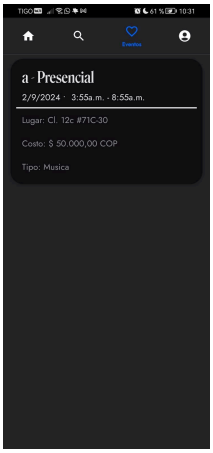
Cuenta:



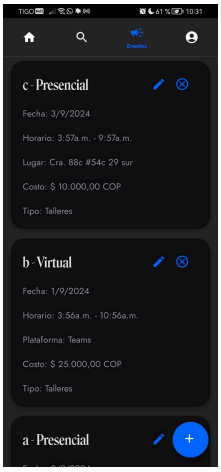
Descubrir, filtrar y ordenar eventos:



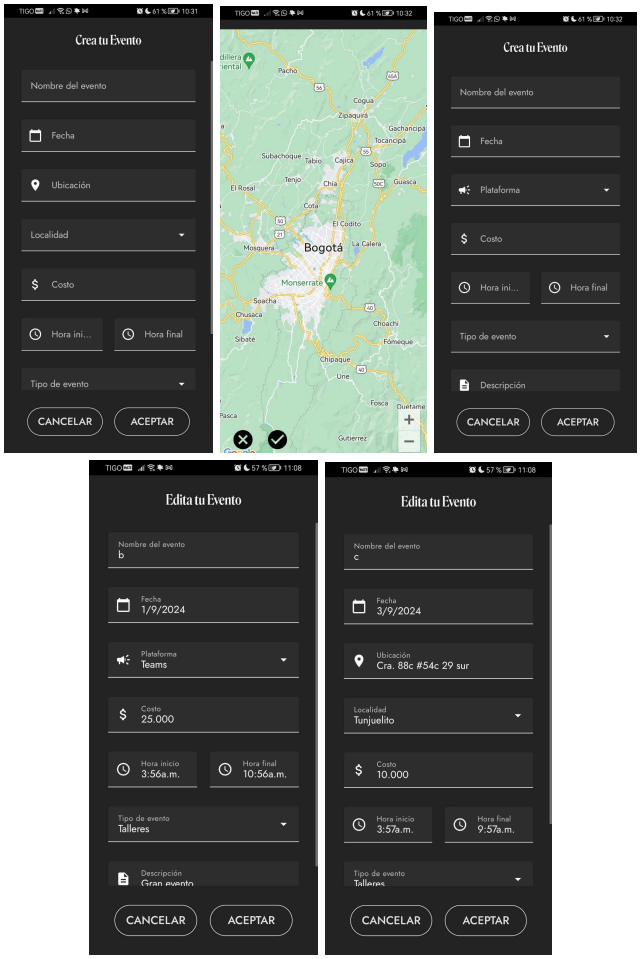
Espacio para eventos favoritos:



Espacios para eventos del expositor:



Espacio para crear y editar un evento:



VI. ENTORNOS DE DESARROLLO Y DE OPERACIÓN

Para la creación del software se va a utilizar el entorno de desarrollo integrado Android Studio, ya que provee las herramientas necesarias para crear aplicaciones. Los lenguajes de programación en los que se basa Android Studio son Java y Kotlin, sin embargo, se va a utilizar el primero por la comodidad que ofrece al grupo de trabajo.

El entorno seleccionado para poner en operación el proyecto es el sistema operativo Android. Los dispositivos a utilizar para emular la aplicación son: Huawei Y9 Prime 2019, Samsung Galaxy A52 y Samsung Galaxy A51.

Para colaborar de forma simultánea a lo largo del proyecto, se va a utilizar Github, que facilita la creación y manipulación de repositorios en la nube. La web utiliza el sistema de control de versiones Git, para manejar el código con mayor facilidad entre dispositivos.

VII. PROTOTIPO DE SOFTWARE FINAL

El link del repositorio es el siguiente:

https://github.com/faridardila/DataStructuresProjectGroupB_Entrega3.git

VIII. DISEÑO, IMPLEMENTACIÓN Y APLICACIÓN DE LAS ESTRUCTURAS DE DATOS

Clases:

1. *Bocu*

- Descripción: *Bocu* es la clase madre. Contiene la información esencial para el funcionamiento de la aplicación.
- Atributos:
  - Estáticos
    - *SIN_REGISTRAR, USUARIO_COMUN, ARTISTA*: se utilizan para identificar el tipo de usuario.
  - Final
    - *Localidades*: lista que almacena las dieciséis localidades de la ciudad de Bogotá.
    - *Intereses*: una lista que almacena tipos de eventos.
    - *Plataformas*: una lista que contiene plataformas virtuales para realizar los eventos.
  - Otros
    - *eventos*, *eventosExpositor*, *eventosFavoritos*: listas de eventos.
    - *posicionesEventosExpositor*: lista que muestra, para cada expositor, las posiciones de los eventos de su autoría dentro de la lista principal *eventos*.
    - *expositores*: lista que contiene todos los expositores.
    - *usuariosComunes*: lista que contiene todos los usuarios comunes.
    - *usuario*: almacena al usuario cuya sesión está abierta.
- Funcionalidades: almacenar la información esencial para los procesos de la aplicación.

2. *Usuario Registrado*:

- Descripción: representa un usuario dentro de la app. Es una clase abstracta que, además, es superclase de *UsuarioComun* y *Artista*.
- Atributos definidos: id, correo electrónico, contraseña, localidad.
- *Usuario Común*
  - Atributos: nombre, apellido, edad, favoritos e intereses.
  - Funcionalidades: puede guardar favoritos y ver eventos.
- Artista:
  - Atributos: nombre del usuario u organización y tipo de evento.
  - Funcionalidades: puede añadir, editar, eliminar y ver eventos.

3. Evento

- Descripción: contiene la información relacionada con el evento en cuestión.
- Atributos: ubicación, costo, categoría, id, nombre, plataforma, horario, descripción, correo del autor, fecha
- Funcionalidades: almacenar información relevante para los usuarios.

Estructuras de datos:

1. Listas enlazadas (administración de usuarios):

- Implementación: búsqueda y comparación de datos.
- Contribución: evitar la creación de usuarios repetidos y permitir las funcionalidades de *Usuario Registrado o Expositor.*

2. Listas desordenadas con arreglos dinámico (página de inicio y página de eventos):

- Implementación: mostrar en el *recyclerView* los eventos generados por los *expositores*.
- Contribución: visibilizar los eventos a todos los usuarios.

3. Pilas con listas enlazadas (historial de eventos):

- Implementación: crea una lista que muestra los eventos desde el último visto hasta el primero.
- Contribución: permite al usuario acceder a los eventos vistos anteriormente.

4. *MaxHeap* y *MinHeap*:

- Implementación: filtros de ordenamiento para los eventos.
- Contribución: permite al usuario encontrar con facilidad eventos de acuerdo a su petición.

5. HashSet y HashMap
    - Implementación: autenticación de inicio de sesión y administración de eventos favoritos de los usuarios registrados.
    - Contribución: permite al usuario tener una mejor experiencia al iniciar sesión por la eficacia con respecto al tiempo de carga.

IX. PRUEBAS DEL PROTOTIPO Y ANÁLISIS COMPARATIVO

Para todas las implementaciones de estructuras lineales y no lineales se hicieron pruebas de rendimiento que tenían el objetivo de medir el tiempo promedio de procesamiento por dato que requería ejecutar una funcionalidad específica, tomando en cuenta diferentes magnitudes de datos: 10 mil, 100 mil, 1 millón, 10 millones y 100 millones. Dicho tiempo fue medido en nanosegundos.

Como se verá más adelante, algunas pruebas carecen de resultados para ciertas cantidades de datos; esto se debe a que se presentaron errores de falta de memoria, principalmente *Java heap space* y *GC overhead limit exceeded*, y -en otros casos- el tiempo de espera para conseguir resultados superó el día.

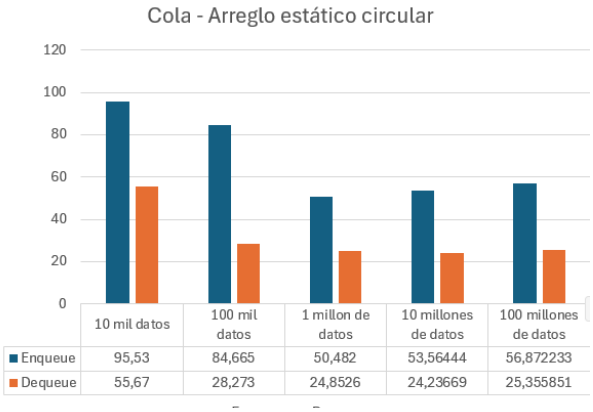
1. Arreglos Estáticos

Para el uso de arreglos estáticos, en cada caso se creó una clase con un *array* de longitud equivalente a la cantidad de datos especificada.

Colas

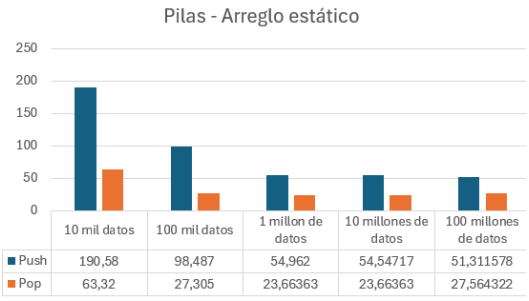
En el caso de la estructura de cola se realizaron dos enfoques para encolar y desencolar: El primero, utilizando un arreglo estático; y el segundo, agregando la metodología de arreglo circular.

- Cola con arreglo circular:

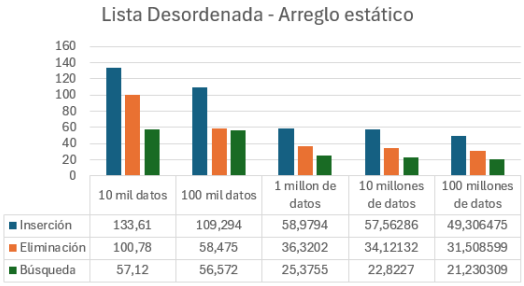


- Cola con arreglo normal:

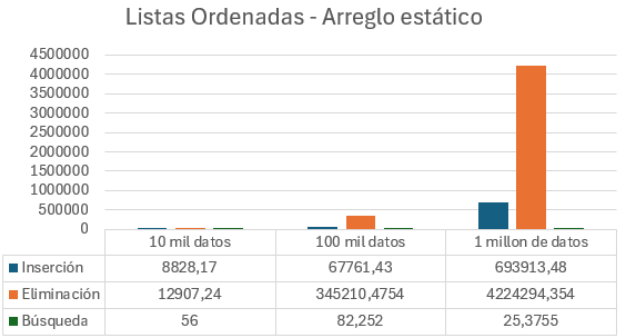
Pilas



Listas desordenadas



Listas Ordenadas



Al analizar los gráficos de las distintas estructuras se concluye que, en ciertos casos, el procesamiento de la estructura en sus distintos métodos no está fuertemente relacionado con la cantidad de datos, mientras que en otros si. Por lo tanto, para los siguientes métodos, la complejidad es O(1): *enqueue* y *dequeue* en colas con arreglo circular, *enqueue* en colas con arreglo estático, *push* y *pop* en pilas, búsqueda en listas

ordenadas, y tanto búsqueda como eliminación e inserción en listas desordenadas.

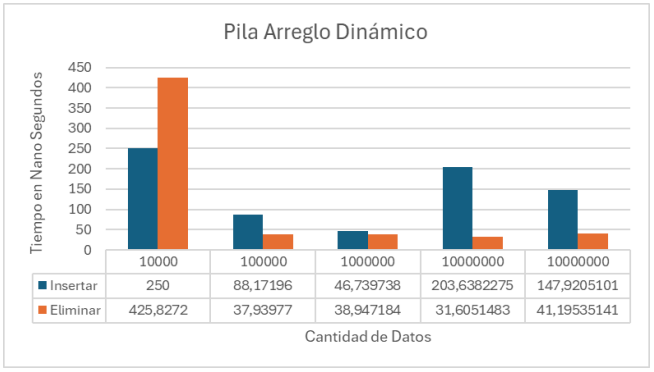
La complejidad es O(n) para los siguientes métodos: *dequeue* en colas con arreglo normal, y tanto inserción como eliminación en listas ordenadas.

2. Arreglos Dinámicos

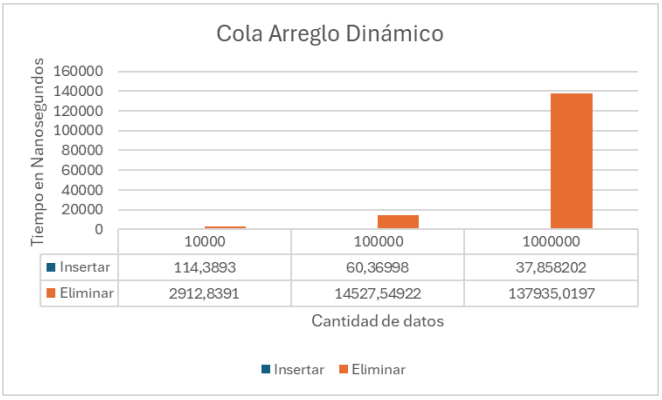
Al implementar arreglos dinámicos se inicializa un arreglo con diez espacios, el cual, una vez está lleno, se duplica de tamaño; para las magnitudes de datos con las cuales se hicieron las pruebas fueron necesarias la siguientes cantidades de redimensionamientos:

Cantidad de Datos	Veces que fue redimensionado
10000	10
100000	14
1000000	17
10000000	20
100000000	24

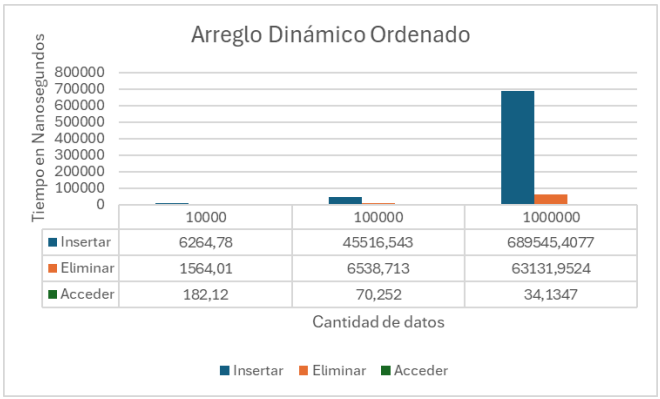
Pila



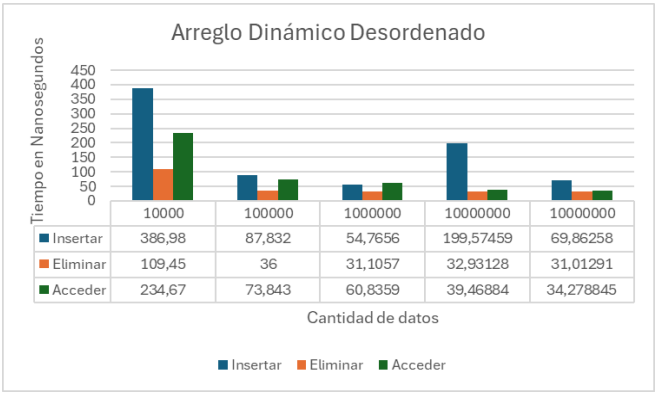
Cola



Arreglo Ordenado



Arreglo Desordenado



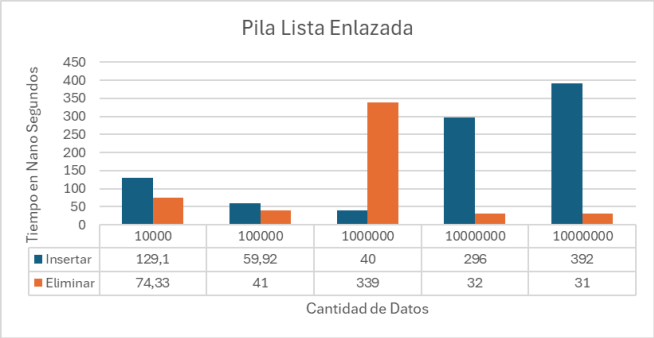
Al analizar los resultados es evidente que en ciertos casos los tiempos no dependen de la cantidad de datos ingresados, mientras que en otros el tiempo se ve fuertemente afectado; por lo tanto, los métodos que tienen complejidad constante O(1) son: inserción y eliminación en pilas, inserción en colas, acceso por índice en listas ordenadas, y acceso, inserción y eliminación en arreglos desordenados.

Además, los métodos que tienen complejidad lineal (O(n)) son: eliminación en colas y tanto inserción como eliminación en listas ordenadas

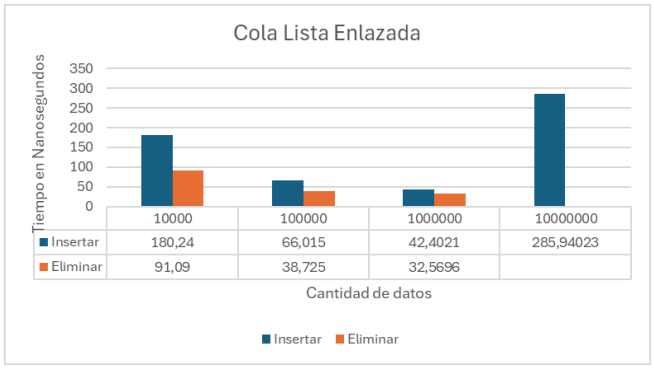
3. Linked List

Para desarrollar las listas enlazadas se utilizaron nodos con dos atributos: *data* de tipo genérico, que contiene el dato almacenado en el nodo, y *next* de tipo *Node*, que guarda la referencia del nodo posterior.

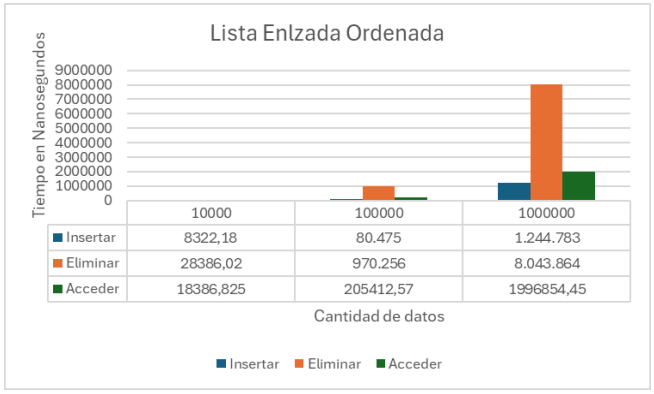
Pilas



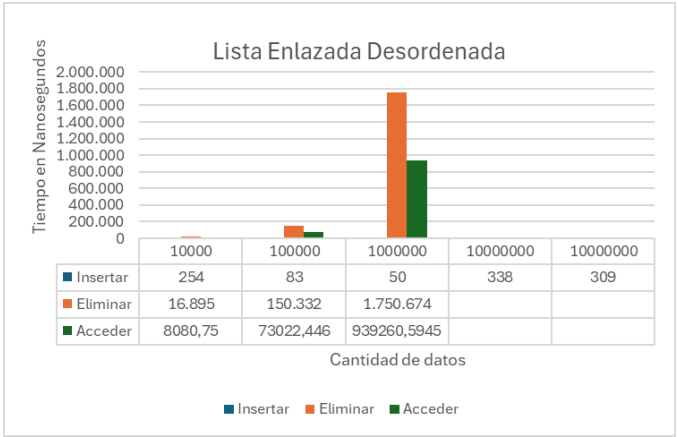
Cola



Listas Ordenadas



Listas Desordenadas



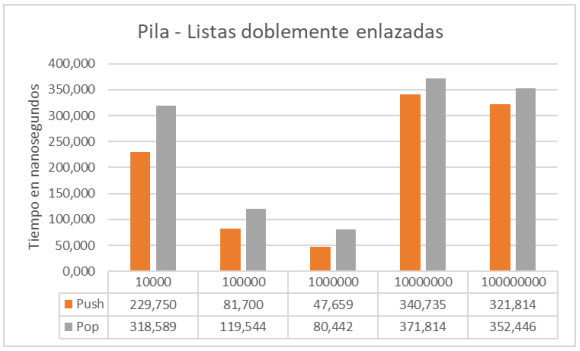
Después de revisar las gráficas obtenidas y llevar a cabo el respectivo análisis asintótico, se determinó que los métodos *push*, *pop*, *enqueue*, *dequeue*, e *insert* en listas desordenadas poseen una complejidad O(1); así mismo, las funcionalidades que tienen complejidad O(n) son inserción para listas ordenadas y tanto eliminación como acceso para listas ordenadas y no ordenadas.

4. Double Linked List

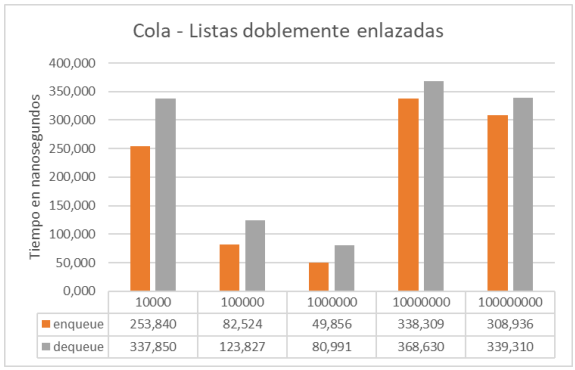
Al igual que las listas enlazadas, las doblemente enlazadas utilizan, por nodo, los atributos *data* y *next*, el primero de tipo genérico y el segundo de tipo *Node*, sin embargo, esta implementación utiliza un nodo adicional llamado *prev* de tipo *Node*, que almacena la referencia del nodo anterior.

En las estructuras de datos lineales generales se encuentra la siguiente información:

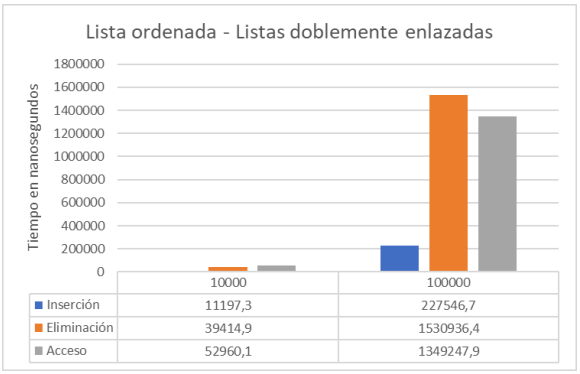
Pilas



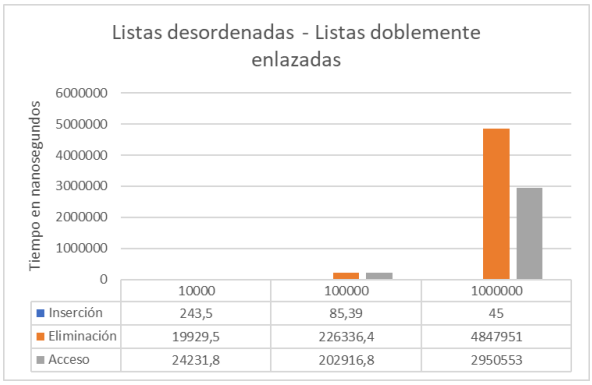
Colas.



Listas ordenadas.

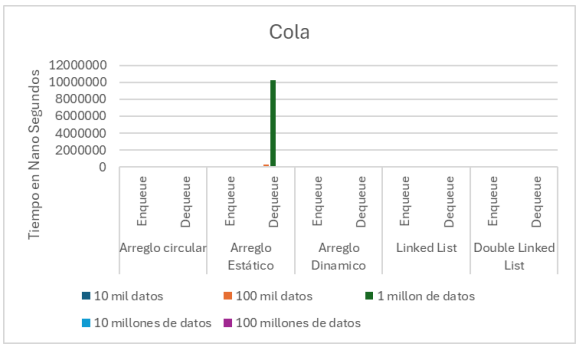


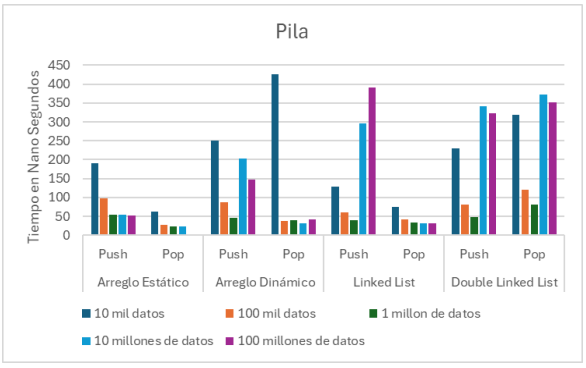
Listas desordenadas.



Al examinar cada una de las gráficas anteriores y realizar el respectivo análisis asintótico, se encontró que los métodos *push*, *pop*, *enqueue*, *dequeue*, e *insert* en listas desordenadas poseen una complejidad O(1); de la misma forma, las funcionalidades que presentan complejidad O(n) son inserción para listas ordenadas y tanto eliminación como acceso para listas ordenadas y no ordenadas.

En las estructuras de datos lineales generales se encuentra la siguiente información.





En definitiva, las mejores implementaciones para colas fueron con arreglos estáticos circulares, y para las demás estructuras lineales fueron con arreglos estáticos, que en general fueron las más eficientes; sin embargo, es necesario tener en cuenta el contexto de uso, ya que puede que se requiera almacenar una longitud indeterminada de datos. También depende del problema o de lo que se quiera hacer, pues puede ser más sencillo implementar unas estructuras en específico incluso si no son las más eficientes.

Para analizar el rendimiento de las estructuras de datos no lineales se plantean hipótesis, se realizan las pruebas de rendimiento, se analizan los resultados obtenidos y, finalmente, se verifica la validez de las hipótesis planteadas.

5. BST y AVL

Para el análisis y posterior comparación de BST y AVL se tomaron en cuenta los métodos de inserción, búsqueda *y* eliminación.

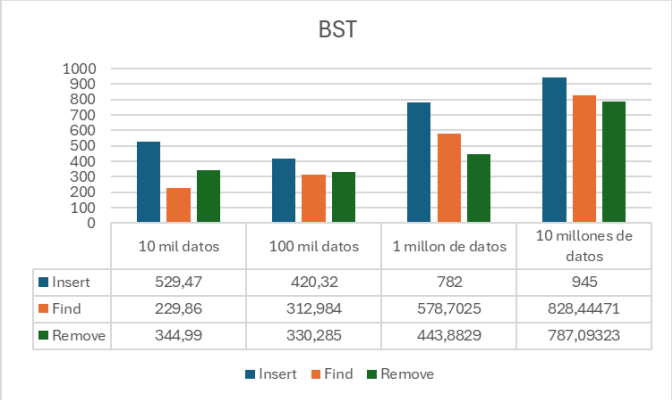
Hipótesis

Se espera que el AVL con inserción y eliminación utilizando recursión tenga mejor rendimiento con respecto a la

implementación con iteración; sin embargo, se prevé que no llegue a funcionar con los 100 millones de datos por la cantidad de memoria que se debe utilizar para estos métodos.

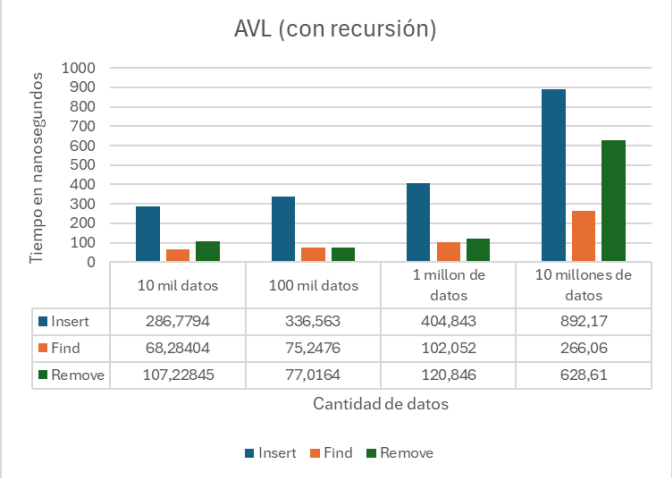
Con respecto a la comparación entre AVL y BST, se espera que el primero consiga un mejor rendimiento general teniendo en cuenta que a mayor número de datos utilizados, el AVL en teoría debe manejar mejor el espacio con el balanceo.

Pruebas y resultados



	10 mil datos	100 mil datos	1 millon de datos	10 millones de datos
Insert	529,47	420,32	782	945
Find	229,86	312,984	578,7025	828,44471
Remove	344,99	330,285	443,8829	787,09323

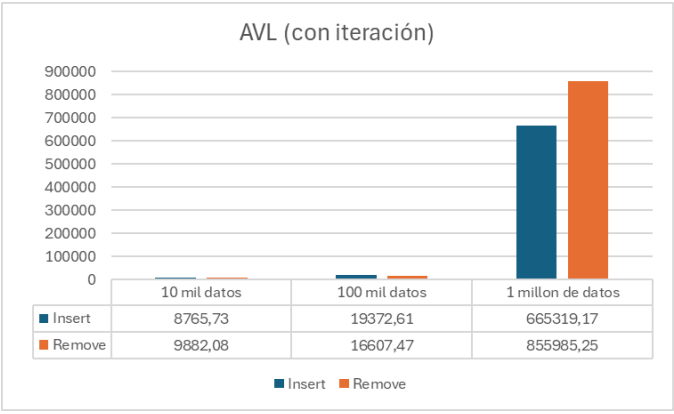
Para el caso de BST, no fue posible obtener los resultados para los 100 millones de datos, debido a que el tiempo de espera de los resultados superó las 24 horas. En los tres métodos se observa un aumento en el tiempo promedio con respecto a la cantidad de datos ingresados, sin embargo no es desproporcionado teniendo en cuenta la magnitud de los datos.



	10 mil datos	100 mil datos	1 millon de datos	10 millones de datos
Insert	286,7794	336,563	404,843	892,17
Find	68,28404	75,2476	102,052	266,06
Remove	107,22845	77,0164	120,846	628,61

En el caso de AVL utilizando recursión, no se registraron los resultados para la muestra de 100 millones de datos, debido a que el computador no pudo procesarlos por la alta cantidad de memoria que requería la implementación.

De acuerdo con los resultados obtenidos, se puede observar que los métodos son eficientes en el AVL. En general, la tendencia de los resultados parece llegar a un valor estable con respecto a la cantidad de datos, por lo que se deduce que los tres métodos son O(log(n)).



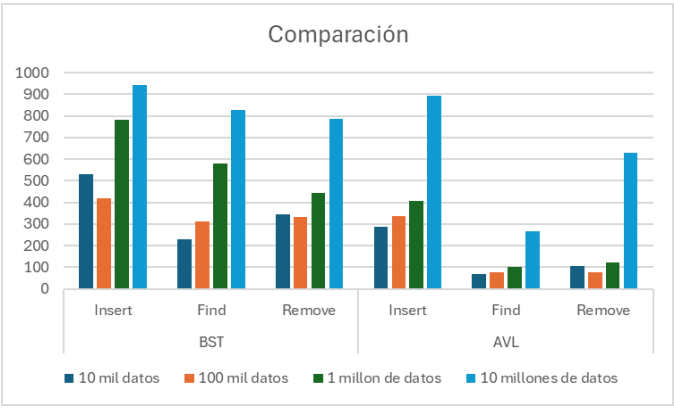
	10 mil datos	100 mil datos	1 millon de datos
Insert	8765,73	19372,61	665319,17
Remove	9882,08	16607,47	855985,25

En el caso del AVL con los métodos de inserción y eliminación utilizando iteración, se muestra que la implementación tiene un tiempo promedio mucho más alto en comparación con la implementación que utiliza recursión.

Por lo tanto, es posible concluir que no es factible utilizar esta forma en ningún caso, ya que el AVL mediante recursión lo supera -en términos de eficiencia- con cualquier cantidad de datos.

Análisis

Para la comparación, se utilizaron los resultados del BST junto con los del AVL con mejor rendimiento. Los métodos analizados son inserción, eliminación y búsqueda de datos.



Al revisar el caso de los 10 millones de datos, el AVL presenta un rendimiento notable mejor frente al BST en los tres métodos.

Teniendo en cuenta los análisis realizados, se plantean las complejidades para las dos implementaciones:

Método	BST		AVL
	Promedio	Peor caso	
Insert	O(log(n))	O(n)	O(log(n))
Find	O(log(n))	O(n)	O(log(n))

Remove	O(log(n))	O(n)	O(log(n))

En conclusión, se comprobó que el AVL con recursión es la mejor implementación con respecto a las comparaciones realizadas.

6. Conjuntos disjuntos

En el caso de los conjuntos disjuntos se analiza el rendimiento de los métodos *unión*, *find* y *connected* con dos implementaciones diferentes: la primera posee compresión de ruta y la segunda cuenta además, con unión por rango.

Hipótesis

Compresión de ruta

Se espera que todas las operaciones tengan un tiempo de ejecución menor que O(log(n)), el rendimiento puede ser aún mejor para los métodos *find* y *connected*, debido a la compresión ruta, no obstante, la operación *union*, al no estar optimizada, puede llegar a tener -en el peor de los casos- complejidad lineal.

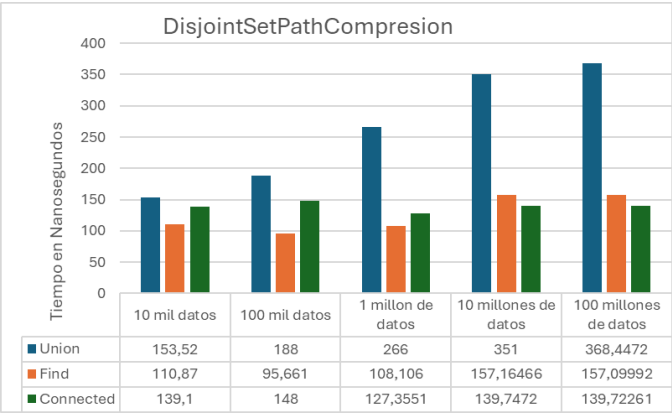
Compresión de ruta y unión por rango

Se estima que todos los métodos tendrán una complejidad computacional casi constante, debido a que la compresión de ruta reduce la altura del árbol en cada operación, y que la unión por rango siempre procura mantener la menor altura posible al fusionar dos conjuntos.

En general, se espera que todos los métodos presenten mejores tiempos de ejecución al ser implementados con unión por rango y compresión de ruta que al ser implementados únicamente con compresión de ruta.

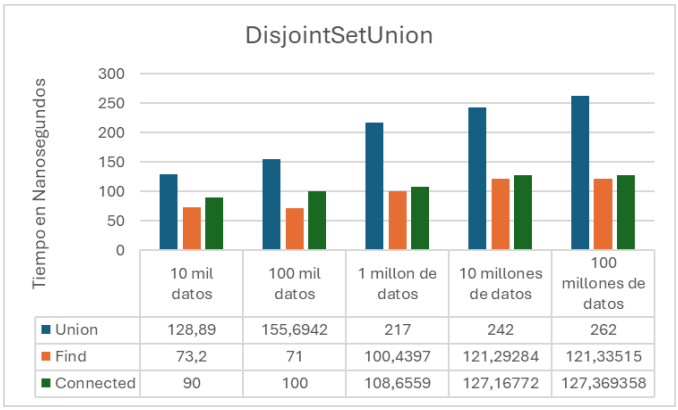
Pruebas y resultados

Compresión de ruta

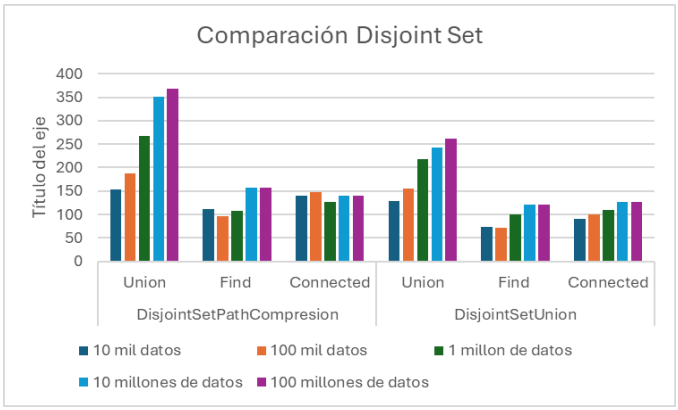


Se evidencia que el método *union* es el que tiene un mayor crecimiento, mientras que las funcionalidades *find* y *connected* crecen muy poco, se mantienen casi constantes.

Compresión de ruta y unión por rango



Una situación similar a la anterior sucede en este caso, sin embargo, esta vez la diferencia entre los tiempos de ejecución de la funcionalidad *union* y los tiempos de ejecución de las otras funcionalidades es menor.



Se observa que los métodos implementados con unión por rango y compresión de ruta presentan los mejores tiempos de ejecución; la operación *union* es la que mejora más su rendimiento al utilizar esta implementación.

Análisis

La mejor implementación, en general, es aquella que usa tanto compresión de ruta como unión por rango, así mismo, en todos los casos se obtienen resultados que confirman la validez de las hipótesis planteadas, exceptuando la unión en el caso de la implementación que solo cuenta con compresión de ruta. Consultando la información disponible en la red, se encuentran las complejidades para cada método en cada implementación.

Método	Unión por rango y compresión de ruta	Compresión de ruta
Union	O(α(n))	O(log*(n))

Find	O(α(n))	O(log*(n))
Connected	O(α(n))	O(log*(n))

O(α(n)) representa la función de Ackermann inversa, la cual, es casi constante y no supera a cuatro para valores razonables de n, por otro lado, O(log*(n)) es el logaritmo iterado que se comporta de manera casi constante y no supera a cinco para valores prácticos de n [2]. En definitiva, O(α(n)) crece más lentamente que O(log*(n)), esto reafirma el hecho de que la implementación con compresión de ruta y unión por rango haya sido la más eficiente.

7. Montículo mínimo y montículo máximo

Para los montículos (montículos máximos y mínimos) se analizaron las funcionalidades *insert*, *remove*, *extractMax-extractMin* y *heapSort* con implementaciones binarias, ternarias y cuaternarias.

Hipótesis

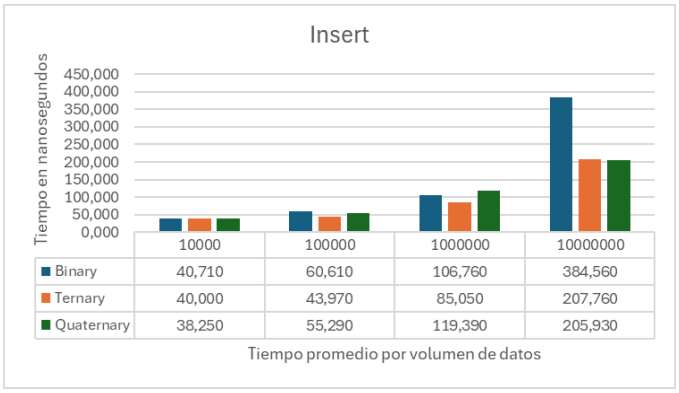
Las implementaciones con más hijos por nodo (ternaria y cuaternaria) deberían, en teoría, ser más rápidas para la mayoría de las operaciones, debido a la reducción en la profundidad del árbol. No obstante, estas ventajas pueden venir a costa de una mayor cantidad de comparaciones por nodo en operaciones específicas. Por lo anterior, se cree que las complejidades serán aproximadamente:

	Insert	Remove	ExtractMax - ExtractMin	HeapSort
Binary	$O(log_2 n)$	$O(2\ log_2 n)$	$O(2\ log_2 n)$	$O(2n\ log_2 n)$
Ternary	$O(log_3 n)$	$O(3\ log_3 n)$	$O(3\ log_3 n)$	$O(3n\ log_3 n)$
Quaternary	$O(log_4 n)$	$O(4\ log_4 n)$	$O(4\ log_4 n)$	$O(4n\ log_4 n)$

Pruebas y Resultados

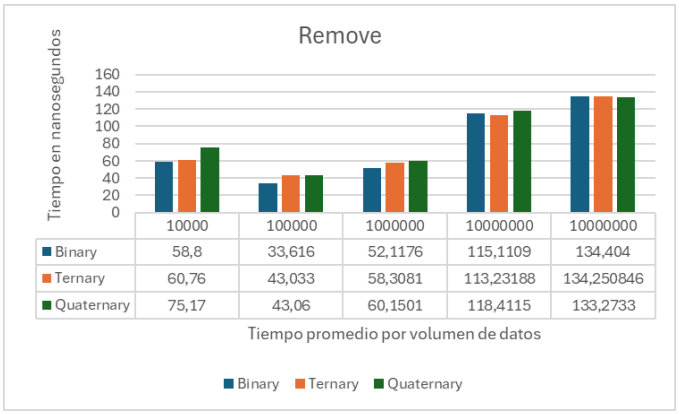
Para realizar las siguientes gráficas se tomaron en cuenta las funcionalidades tanto en *MaxHeap* como en *MinHeap*, al tener un rendimiento similar en todas las funcionalidades se realiza el promedio entre los resultados obtenidos para cada una.

Insert



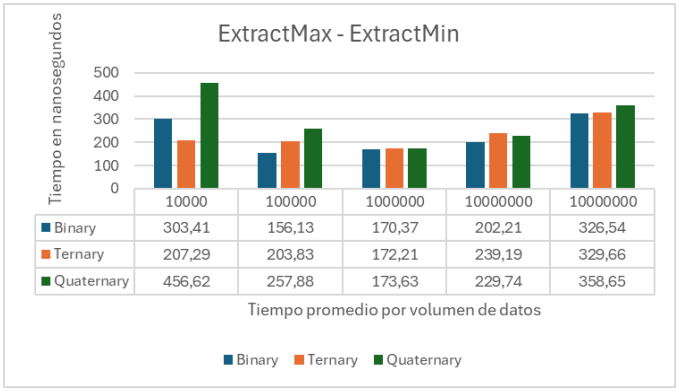
Se evidencia que la implementación binaria es la que posee un mayor tiempo de ejecución y, por ende, una complejidad más alta que la obtenida para las implementaciones ternarias y cuaternarias.

Remove



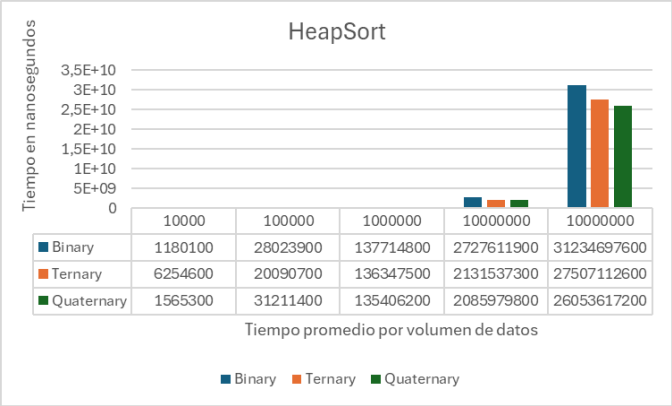
Se vuelve a evidenciar los expresado en el método *insert*, diferenciándose en que en esta ocasión los tiempos de ejecución son más cercanos entre sí.

ExtractMax - ExtractMin



Se obtienen resultados similares entre las diferentes implementaciones para esta prueba rendimiento.

HeapSort



	10000	100000	1000000	10000000	10000000
Binary	1180100	28023900	137714800	2727611900	31234697600
Ternary	6254600	20090700	136347500	2131537300	27507112600
Quaternary	1565300	31211400	135406200	2085979800	26053617200

Los tiempos de ejecución resultaron muy similares para la funcionalidad de ordenamiento.

Análisis

Teniendo en cuenta los resultados anteriores se puede concluir que la hipótesis inicial es correcta mayoritariamente. En métodos donde la complejidad depende tanto de la cantidad de hijos como de la profundidad del árbol se observan tiempos de ejecución muy similares, en cambio, para métodos donde la complejidad depende únicamente de la profundidad del montículo, se aprecia que las implementaciones con mayor cantidad de hijos tienen tiempos de ejecución más pequeños.

Recordando la siguiente propiedad para el cambio de base entre logaritmos

$$log_b a = \frac{log_c a}{log_c b} \; / \; a, \; b, \; c \in R$$

Es posible concluir que la complejidad para los métodos en las tres implementaciones es:

Funcionalidad	Big(O)
Insert	$O(\log n)$
Remove	$O(\log n)$
ExtractMax - ExtractMin	$O(\log n)$
HeapSort	$O(n \log n)$

Comentarios adicionales: Aunque las implementaciones pueden tener una complejidad esperada, en la práctica, el rendimiento puede depender de otros factores como la eficiencia del acceso a memoria y la gestión del caché.

8. Tablas hash

En el caso del *HashTables* se analiza el rendimiento de los métodos *add*, *contains, remove* y el número de colisiones, en 6 implementaciones donde se modifica el factor de carga en 0.5, 0.75 y 1, y usando la operación módulo o el operador *bitwise*.

Hipótesis

Add: se espera una complejidad mayor a O (1) que aumente a medida que los datos y los redimensionamientos aumentan y que sea inversamente proporcional al factor de carga.

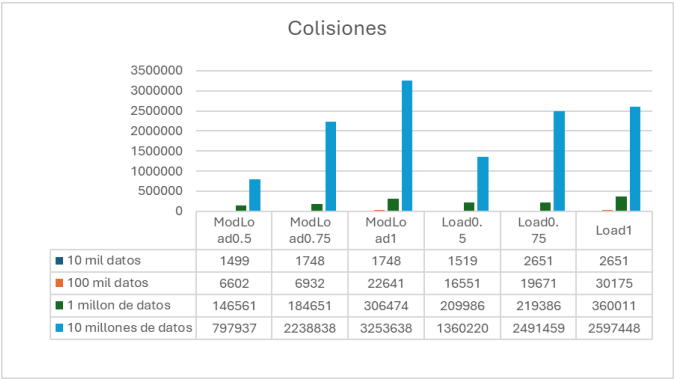
Contains y Remove: se esperan tiempos casi constantes que no se vean afectados por la cantidad de datos, pues únicamente se trata de accesos a un arreglo y luego a una lista enlazada que se estima sea de un tamaño reducido

Colisiones

Se espera que el factor de carga de 0.5 tenga menores colisiones, pues la cantidad de espacios usados se va a mantener a la mitad, luego le seguiría el factor de carga de 0.75 y por último 1.

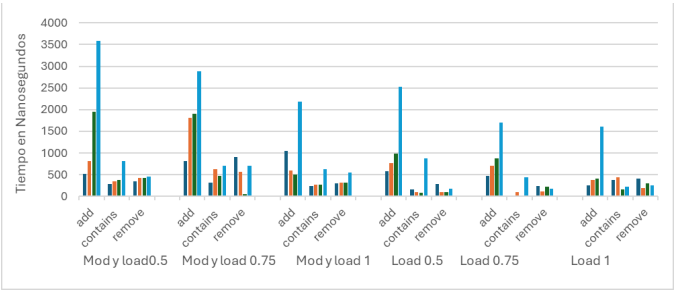
Comparativamente se espera que el factor de carga de 0.5 de mejores resultados en cuestión de colisiones, pero de el peor resultado en tiempo pues el arreglo va a ser redimensionado más veces. Además, que ambas implementaciones respecto al uso del módulo den resultados similares.

Resultados



	ModLoad0.5	ModLoad0.75	ModLoad1	Load0.5	Load0.75	Load1
10 mil datos	1499	1748	1748	1519	2651	2651
100 mil datos	6602	6932	22641	16551	19671	30175
1 millon de datos	146561	184651	306474	209986	219386	360011
10 millones de datos	797937	2238838	3253638	1360220	2491459	2597448

Observamos que el número de colisiones aumenta a medida que lo hace el factor de carga, y notamos que aquella implementación que no usa el módulo genera menos colisiones.



Observamos que los métodos *contains y remove* se mantienen casi constantes y varían muy poco en todas las implementaciones. En el caso de *add* se observa que tiene un

comportamiento creciente a medida que la cantidad de datos aumenta y que entre menor sea el factor de carga más tiempo se demora en realizar la operación. Comparativamente son mejores las implementaciones que no usan la operación módulo y se observan mejores tiempos en general con el factor de carga de 1 sin uso de módulo

Análisis

En el caso de las colisiones nos encontramos con los resultados esperados, pues en el caso de tener un factor de carga de 0.5 implica que la mitad del arreglo tiene elementos, por lo tanto, hay una menor probabilidad de tener colisiones entre los resultados de la función hash; en cambio, con un factor de carga de 1 se tiene un arreglo lleno y, por ende, es casi seguro que exista alguna colisión.

Para las operaciones de *contains y remove* nos encontramos con tiempos casi constantes que no se ven afectados por la cantidad de datos.

En general, se tiene que la implementación que usa el factor de carga de 1 y el operador *bitwise* es la que mejores tiempos tiene en la operación *add*, esto se debe a que *bitwise* es menos complejo que el módulo, y el factor de carga de 1 hace que el arreglo no se redimensione muchas veces como en el caso de 0.5, no obstante, esto trae un gran número de colisiones. Por lo tanto, podríamos concluir que en términos de uso de memoria, tiempo y colisiones el factor de carga de 0.75 y uso del operador bitwise es la implementación más balanceada en todos aspectos.

Cabe recalcar que los tiempos de ejecución de estas operaciones dependen de la selección de una buena función hash que distribuya de manera equitativa los datos, previniendo las colisiones excesivas.

9. Graphs

Este análisis tiene como objetivo comparar tres diferentes representaciones de grafos: matriz de adyacencia, lista de adyacencia y lista de vértices, para determinar cuál es más eficiente a la hora de consultar si existe una conexión entre dos vértices en un grafo. Además, se evaluaron los tiempos de ejecución de los algoritmos de búsqueda en profundidad (DFS) y en amplitud (BFS) para comprobar su rendimiento en función de la densidad y el tamaño del grafo.

Hipótesis

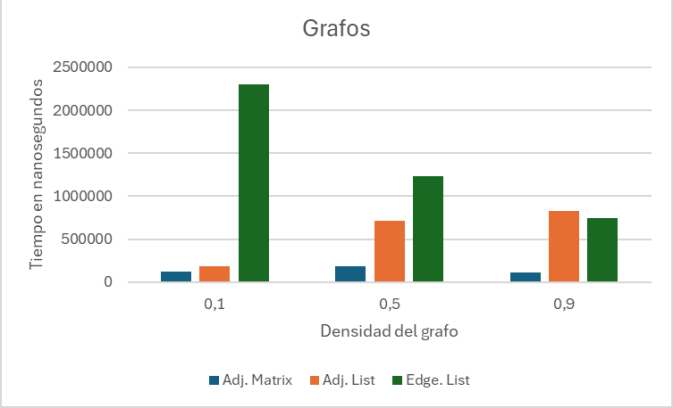
Las tres representaciones que se evaluarán tienen las siguientes características en términos de tiempo de consulta y uso de espacio:

Representación	Tiempo para encontrar un vértice	Espacio	Mejor para
Matriz de Adyacencia	O(1)	O(V^2)	Grafos densos, operaciones de álgebra de grafos.
Lista de Adyacencia	O(V) en el peor caso	O(V+E)	Grafos dispersos, búsqueda de vecinos
Lista de Vértices	O(E)	O(V)	Almacenar información básica de los vértices

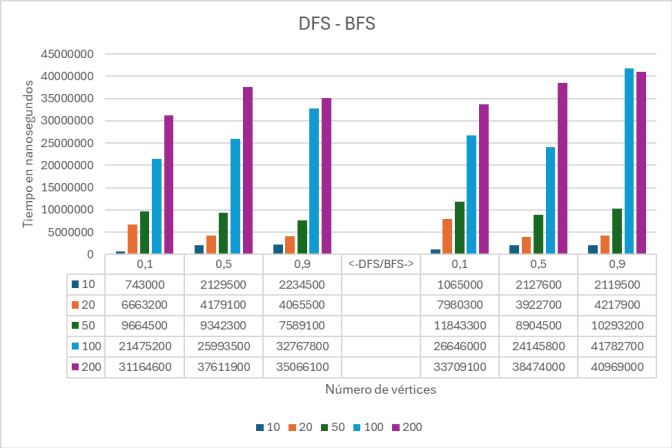
Con base en esta hipótesis, se espera que la lista de adyacencia sea la opción más eficiente para verificar la existencia de conexiones en grafos de baja densidad, mientras que la matriz de adyacencia será preferible para grafos de alta densidad. La matriz de adyacencia no será eficiente en grafos dispersos debido a que almacena todas las posibles conexiones, incluso las que no existen, lo que resulta en un uso innecesario de espacio. Por el contrario, en grafos poco densos, las listas de adyacencia y de vértices son más eficientes en términos de consumo de espacio.

Además, para los algoritmos DFS y BFS, se espera una complejidad temporal de O(V + E), es decir, los tiempos de ejecución aumentarán a medida que crezca el número de vértices y aristas en el grafo.

Pruebas y Resultados



Tal como se planteó inicialmente, la matriz de adyacencia mostró un comportamiento constante, sin importar la densidad del grafo. En contraste, la lista de adyacencia presentó tiempos de búsqueda crecientes a medida que aumentaba la densidad del grafo. Finalmente, la lista de vértices demostró ser la menos eficiente, ya que requiere recorrer todas las aristas para encontrar una conexión, lo que es especialmente ineficiente en grafos dispersos.



	0,1	0,5	0,9	<-DFS/BFS->	0,1	0,5	0,9
10	743000	2129500	2234500		1065000	2127600	2119500
20	6663200	4179100	4065500		7980300	3922700	4217900
50	9664500	9342300	7589100		11843300	8904500	10293200
100	21475200	25993500	32767800		26646000	24145800	41782700
200	31164600	37611900	35066100		33709100	38474000	40969000

El tiempo de ejecución del algoritmo DFS se comportó según lo esperado, con una complejidad de O(V + E). A medida que aumentaba el número de vértices y la densidad del grafo, el tiempo también creció de manera consistente. Mientras que BFS mostró un comportamiento similar al de DFS, con tiempos de ejecución también en O(V + E). Sin embargo, en algunos casos específicos, BFS fue ligeramente más rápido que DFS debido a la diferencia en la forma en que se exploran los vértices (por niveles en lugar de por profundidad).

Análisis.

En conclusión, la matriz de adyacencia es la opción más eficiente cuando el grafo es denso, ya que su comportamiento constante justifica el mayor consumo de espacio. Sin embargo, la lista de adyacencia pierde eficiencia a medida que la densidad del grafo aumenta. Por su parte, la lista de vértices resulta ser la peor opción para verificar conexiones entre vértices, especialmente en grafos dispersos, debido a su ineficiencia al tener que recorrer todas las aristas.

Los algoritmos DFS y BFS presentaron comportamientos similares, aumentando sus tiempos de ejecución a medida que el número de vértices y aristas en el grafo crecía, tal como se esperaba. Ambos algoritmos son apropiados para búsquedas en grafos, aunque en situaciones específicas, BFS puede ser ligeramente más eficiente dependiendo de la estructura del grafo.

X. Información de acceso al video demostrativo del prototipo de software

Entrega 1:

https://drive.google.com/drive/folders/15_-eG5SVKDk34VKrz7Br3tsFfCbIOmfQ?usp=sharing

Entrega 2:

https://drive.google.com/drive/folders/1YPkt88uwSq5_T8D8qechMXyEMlKmHK3V?usp=sharing

Entrega 3:

https://drive.google.com/file/d/1CXrcB_N0LZRrZZLpJhz1Lfy7FmmObFVN/view?usp=sharing

XI. Roles y actividades

INTEGRANTE	ROL(ES)	ACTIVIDADES REALIZADAS
Cristofer Ordoñez	Líder Experto	Descripción del problema a resolver. Descripción general del primer prototipo implementado.
Farid Ardila	Coordinador Observador	Entornos de desarrollo y de operación (ejecución y despliegue). Requerimientos funcionales del software. Tabla resumen de los integrantes del equipo, sus roles y las actividades realizadas.
Diego Arévalo	Técnico Investigador	Descripción preliminar de la interfaz (vista gráfica) de usuario. Usuarios del producto de software.
Ángel Beltran	Animador Secretario	Descripción del diseño y de la implementación.

XII. Dificultades y lecciones aprendidas

El análisis asintótico y las pruebas de rendimiento fueron esenciales para determinar la complejidad de las principales funcionalidades aplicadas en el proyecto.

Las pruebas de rendimiento demandaron una cantidad considerablemente grande de tiempo.

Las reuniones recurrentes fueron un aspecto clave para el correcto desarrollo del proyecto.

Los roles de equipo ayudaron a distribuir la carga de trabajo de forma equitativa.

Los requerimientos funcionales deben resolver las necesidades del usuario de forma eficiente.

XIII. Referencias bibliográficas

[1] A. Montenegro González, «La “Atenas suramericana”. Búsqueda de los orígenes de la denominación dada a Bogotá», Mem.soc, vol. 7, n.º 14, pp. 133–143, mar. 2014.W.-K. Chen, *Linear Networks and Systems* (Referencia de libro). Belmont, CA: Wadsworth, 1993, pp. 123–135.

[2] J. Kleinberg and E. Tardos, *Algorithm Design*, 1st ed. Pearson, 2006.