SIApp, una forma eficiente de planear horarios académicos

Cristian Fabian Martinez Bohórquez, Edward Jeisen Jair Arévalo Peña, Juan David Cruz Giraldo

No. de equipo de trabajo: F

I. INTRODUCCIÓN

En la universidad Nacional de Colombia, planear horarios de forma eficiente es un desafío recurrente de los estudiantes en el momento de iniciar un nuevo semestre, principalmente por las fallas que tiene la plataforma del portal de servicios académicos. Para abordar este problema surge SIApp, una aplicación diseñada para planificar horarios académicos de forma eficiente, mediante el uso de estructuras de datos. En este trabajo se presentará una descripción detallada del diseño y desarrollo de la aplicación.

II. DESCRIPCIÓN DEL PROBLEMA A RESOLVER

La plataforma SIA de la universidad, encargada de proporcionar información vital como historial académico, datos generales, disponibilidad de materias y trámites educativos, enfrenta frecuentes caídas y fallos, especialmente en fechas críticas para los estudiantes. Esta situación dificulta enormemente la planificación adecuada de horarios y puede provocar estrés adicional al impedir el acceso oportuno a la información necesaria. La falta de disponibilidad para acceder a la plataforma puede resultar en la incapacidad de realizar ajustes necesarios en los horarios académicos, lo que afecta negativamente el progreso de los estudiantes.

III. USUARIOS DEL PRODUCTO DE SOFTWARE

SIApp está diseñada específicamente para estudiantes de la Universidad Nacional de Colombia.

IV. REQUERIMIENTOS FUNCIONALES DEL SOFTWARE

Funcionalidad: Búsqueda de Materias

Descripción: Permitir a los usuarios buscar y visualizar información detallada sobre las materias disponibles en la universidad.

Acciones iniciales y comportamiento esperado:

1. El usuario ingresa el nombre de la materia o parte del nombre en el campo de búsqueda.
2. El sistema muestra una lista de resultados coincidentes con el término de búsqueda.
3. El usuario puede seleccionar una materia de la lista para ver información detallada como cupos disponibles, profesor asignado, horarios de clases, entre otros.
4. Si la materia es de interés para el estudiante, puede guardarla en materias favoritas.

Funcionalidad: Creación de Horario

Descripción: Permitir a los usuarios crear y personalizar su horario académico agregando materias de su interés.

Acciones iniciales y comportamiento esperado:

1. El usuario accede a la función de creación de horario desde el menú principal de la aplicación.
2. El sistema muestra una interfaz donde el usuario puede ver un horario semanal vacío con los días de la semana y franjas horarias.
3. El usuario puede seleccionar una materia de su lista de materias favoritas o buscar una nueva y agregarla al horario.
4. El sistema valida que no existan conflictos de horario con otras materias ya agregadas y muestra un mensaje de error si se detecta algún conflicto.
5. El usuario puede ajustar el horario según sus preferencias, agregar más materias o eliminar materias ya agregadas.
6. El sistema guarda automáticamente los cambios realizados en el horario y permite al usuario acceder y editar el horario en cualquier momento.

Funcionalidad: Control de horas de estudio.

Descripción: Permitir a los usuarios registrar y gestionar el tiempo dedicado al estudio de cada materia en base a los créditos correspondientes.

Acciones iniciales y comportamiento esperado:

1. El usuario selecciona una materia de su lista de materias.
2. El usuario ingresa la cantidad de horas dedicadas al estudio de esa materia en un día específico o activa un contador que lleva la cantidad de horas que llega estudiando.
3. El sistema registra y guarda esta información para el análisis posterior del tiempo de estudio por materia y día.

Funcionalidad: Calendario Integrado

Descripción: Integrar un calendario dentro de la aplicación para que los usuarios puedan organizar sus horarios académicos y eventos importantes.

Acciones iniciales y comportamiento esperado:

1. El usuario puede visualizar un calendario mensual con eventos destacados como clases, exámenes, fechas límite de proyectos, etc.

2. El usuario puede agregar nuevos eventos al calendario, especificando la fecha, hora, título y descripción del evento.

3. El sistema muestra notificaciones de eventos próximos o recordatorios de actividades importantes.

Funcionalidad: Cálculo de promedio y seguimiento de notas.

Descripción: Permitir a los usuarios calcular su promedio académico, llevar un registro de las notas en cada materia, y conocer cuánto deben sacar en su próxima nota para pasar la materia.

Acciones iniciales y comportamiento esperado:

1. El usuario selecciona las materias cursadas y registra las calificaciones obtenidas en cada una.

2. El sistema calcula automáticamente el promedio general del estudiante, mostrando el resultado en pantalla.

3. Adicionalmente, el usuario puede seleccionar la opción “nota final para pasar” qué hará el cálculo de cuánto debe sacar en la próxima nota para pasar la asignatura.

Funcionalidad: Cola prioritaria de tareas

Descripción: Permite a los usuarios crear una cola prioritaria de tareas y parciales, lo que les permite organizar y gestionar eficientemente sus actividades académicas.

Acciones iniciales y comportamiento esperado:

1. El usuario accede a la función de cola prioritaria desde el menú principal de la aplicación.

2. El sistema muestra una interfaz donde el usuario puede ver una lista de todas las tareas y parciales pendientes.

3. El usuario puede agregar nuevas tareas o parciales especificando el nombre, la fecha de vencimiento y la prioridad de cada una.

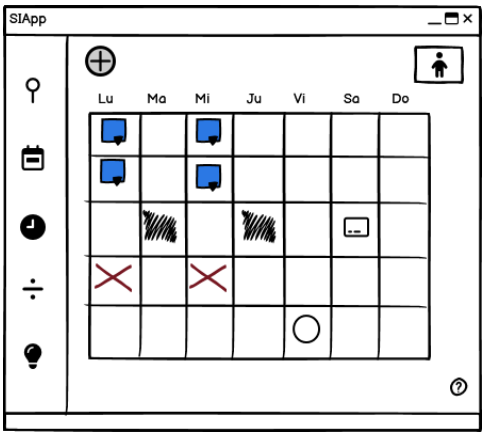
4. El sistema ordena automáticamente las tareas y parciales en la cola según su prioridad, con las tareas más urgentes en la parte superior de la lista.
5. El usuario puede editar o eliminar tareas y parciales de la cola según sea necesario.

6. El sistema notifica al usuario sobre las tareas o parciales próximos a vencerse, ayudándoles a gestionar su tiempo de manera efectiva.

7. El usuario puede marcar las tareas o parciales completadas, lo que los elimina de la cola o los mueve a una sección de tareas completadas.

V. DESCRIPCIÓN DE LA INTERFAZ DE USUARIO PRELIMINAR

8. La interfaz de usuario de SIApp está diseñada para ser intuitiva y fácil de usar. **En la parte izquierda de la pantalla**, se encontrará un panel lateral que contiene botones para acceder a cada funcionalidad de la aplicación. Cada botón representa una funcionalidad específica, como "Buscar Materias", "Calendario", etc. Al hacer clic en uno de estos botones, se abrirá la función respectiva en el área principal de la interfaz. **En la esquina superior derecha** de la pantalla, estará ubicado un menú de configuración del usuario. para acceder a opciones como editar su nombre, correo institucional u otras preferencias de la cuenta. El resto de la interfaz está dedicada a mostrar la funcionalidad seleccionada. Por ejemplo, si estás utilizando la función de "Horario", verás una representación visual del horario de tus clases.



VI. ENTORNOS DE DESARROLLO Y DE OPERACIÓN

Lenguaje: Java

Entorno de desarrollo: Para el desarrollo del software haremos uso de la IDE intelliJ, además de algunas herramientas adicionales como git.

Sistemas operativos compatibles: La aplicación debe ser compatible con windows desde una versión igual o superior a 7, linux y macOs.

Configuración específica: La aplicación será fácil de instalar y ejecutar siempre y cuando cumpla con requisitos mínimos de hardware y con tener instalado el entorno de ejecución de ja**va.**

VII. PROTOTIPO DE SOFTWARE INICIAL

El prototipo inicial de software cuenta con las siguientes funcionalidades incorporadas:

Adición, búsqueda y eliminación de asignaturas de la base de datos:
Nuestro programa permite agregar, buscar o eliminar una asignatura específica de la base de datos al proporcionar los atributos respectivos.

Creación de horario para usuario:
Se creará un horario como una lista de estructura de datos, diseñada para almacenar asignaturas con los atributos que mejor se adapten a las preferencias del usuario.

Adición y eliminación de asignaturas en el horario:
Los usuarios podrán añadir o eliminar asignaturas de su horario según su conveniencia, lo que les proporcionará un mejor panorama para la selección de materias.

El software desarrollado registra en el siguiente repositorio de github: https://github.com/cmartinezbo/ProjectDataStructures

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

Estructuras de datos implementadas:
1. HashMap: Para implementar el hashmap usamos dos clases, hashNode y hashMap, la clase Node se encuentra dentro de la clase hashMap y contiene los siguientes atributos y métodos:

hashNode:

Atributos:
- **K key:** La clave del nodo, utilizada para ubicar el par clave-valor en el mapa.
- **V value**: El valor asociado a la clave en el nodo.
- **int hashCode:** El código hash de la clave, utilizado para determinar la posición en el array de buckets.
- **HashNode<K, V> next:** Referencia al siguiente nodo en caso de que haya colisiones (listas enlazadas dentro de un bucket).

Métodos:
- **getKey():** Retorna la clave almacenada en el nodo**.**
- **getValue():** Retorna el valor asociado a la clave en el nodo.

hashMap:

Atributos:
- **ArrayList<HashNode<K, V>> bucket:** ArrayList que contiene las referencias a los buckets donde se almacenan las listas enlazadas de nodos.
- **int capacity:** Capacidad del mapa, es decir, el tamaño del array de buckets.
- **int size:** Número de pares clave-valor actualmente almacenados en el mapa.
- **Double loadFactor:** Factor de carga que determina cuándo el mapa debe redimensionarse para mantener la eficiencia.

Métodos:
- **hash(K key):** Calcula la posición de un nodo basado en el código hash de la clave.
- **add(K key, V value):** Agrega una clave y su valor asociado al mapa. Si la clave ya existe, actualiza su valor. Redimensiona la tabla si el factor de carga supera el umbral.
- **resize():** Duplica la capacidad del mapa y reubica todos los nodos en sus nuevos buckets.
- size(): Retorna el número de pares clave-valor en el mapa.
- **isEmpty():** Verifica si el mapa está vacío.
- **remove(K key):** Elimina el nodo que contiene la clave dada, si existe, y devuelve el valor asociado.
- get(K key): Busca la clave en el mapa y devuelve su valor asociado, o null si la clave no está presente.
- **containsKey(K key):** Verifica si una clave específica está en el mapa.
- **values():** Retorna una lista con todos los valores almacenados en el mapa.

2. Matriz de adyacencia: Para implementar la matriz de adyacencia usamos dos clases, graphNode y graph.

GraphNode:

Esta clase representa un nodo en el grafo, que tendrá una clave y valor asociados.

Atributos:

- **K key**: La clave del nodo.
- **V value**: El valor asociado a la clave del nodo.
- **int index**: El índice del nodo en la matriz de adyacencia.

Métodos:

- **getKey()**: Retorna la clave del nodo.
- **getValue()**: Retorna el valor asociado a la clave del nodo.

Graph:

Atributos:

- **GraphNode<K, V>[] nodes:** Un array que contiene los nodos del grafo.
- **int[][] adjacencyMatrix:** Matriz de adyacencia donde adjacencyMatrix[i][j] es 1 si hay una arista entre los nodos i y j, y 0 si no la hay.
- **int size:** El número de nodos actuales en el grafo.

Métodos:

- **addNode(K key, V value):** Agrega un nodo al grafo.
- **addEdge(K key1, K key2):** Agrega una arista entre dos nodos.
- **isEdge(K key1, K key2):** Verifica si existe una arista entre dos nodos.
- **listEdges():** Lista todas las aristas del grafo.
- **getNeighbors(K key):** Lista los vecinos (nodos adyacentes) de un nodo específico.

3. Graph (Lista de adyacencia): Esta clase representa un grafo usando listas de adyacencia, mediante dos clases, graphNode y graph.

GraphNode:

Esta clase representa un nodo en el grafo, que tendrá una clave y valor asociados.

Atributos:

- **K key**: La clave del nodo.
- **V value**: El valor asociado a la clave del nodo.
- **int index**: El índice del nodo en la matriz de adyacencia.

Métodos:

- **getKey()**: Retorna la clave del nodo.
- **getValue()**: Retorna el valor asociado a la clave del nodo.

Graph:

Atributos:

- **GraphNode<K, V>[] nodes:** Un array que contiene los nodos del grafo.
- **List<List<GraphNode<K, V>>> adjacencyList:** Lista de adyacencia donde cada nodo tiene una lista de vecinos.
- **int size:** El número de nodos actuales en el grafo.
- **int capacity:** La capacidad máxima de nodos que el grafo puede manejar.

Métodos:

- **addNode(K key, V value):** Agrega un nodo al grafo.
- **addEdge(K key1, K key2):** Agrega una arista entre dos nodos.
- **isEdge(K key1, K key2):** Verifica si existe una arista entre dos nodos.
- **listEdges():** Lista todas las aristas del grafo**.**
- **getNeighbors(K key):** Lista los vecinos (nodos adyacentes) de un nodo específico.

4. Graph (Lista de aristas): Esta clase representa un grafo usando listas de aristas, mediante tres clases, graphNode, edge y graph.

GraphNode:

Esta clase representa un nodo en el grafo, que tendrá una clave y valor asociados.

Atributos:

- **K key**: La clave del nodo.
- **V value**: El valor asociado a la clave del nodo.
- **int index**: El índice del nodo en la matriz de adyacencia.

Métodos:

- **getKey()**: Retorna la clave del nodo.
- **getValue()**: Retorna el valor asociado a la clave del nodo.

Edge:

Esta clase representa una arista entre dos nodos en el grafo.

Atributos:

- **GraphNode<K, V> from:** El nodo de origen de la arista.
- **GraphNode<K, V> to:** El nodo de destino de la arista.

Métodos:

- **getFrom():** Retorna el nodo de origen de la arista.
- **getTo():** Retorna el nodo de destino de la arista.

Graph:

Atributos:

- **GraphNode<K, V>[] nodes**: Un array que contiene los nodos del grafo.
- **List<Edge<K, V>> edges: U**na lista de todas las aristas del grafo.
- **int size:** El número actual de nodos en el grafo.

- **int capacity:** La capacidad máxima de nodos que el grafo puede manejar.

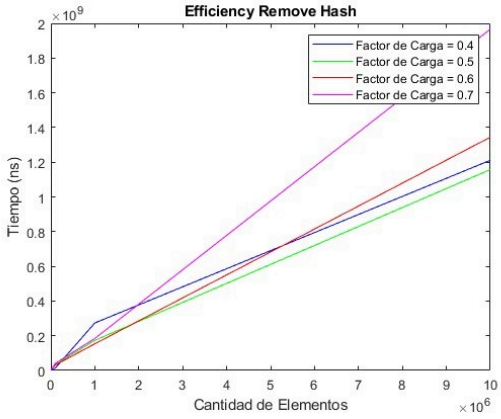
Métodos:

- **addNode(K key, V value):** Agrega un nodo al grafo.
- **addEdge(K key1, K key2):** Agrega una arista entre dos nodos.
- **isEdge(K key1, K key2):** Verifica si existe una arista entre dos nodos.
- **listEdges():** Lista todas las aristas del grafo.
- **getNeighbors(K key):** Lista los vecinos (nodos adyacentes) de un nodo específico.

IX. PRUEBAS DEL PROTOTIPO Y ANÁLISIS COMPARATIVO

hashMap(Modificando el factor de carga): Para las pruebas de esta entrega, en el caso de los hashmaps, modificamos las estructuras en un rango de 0.4 hasta 0.7 en los distintos métodos como add, remove y contains.

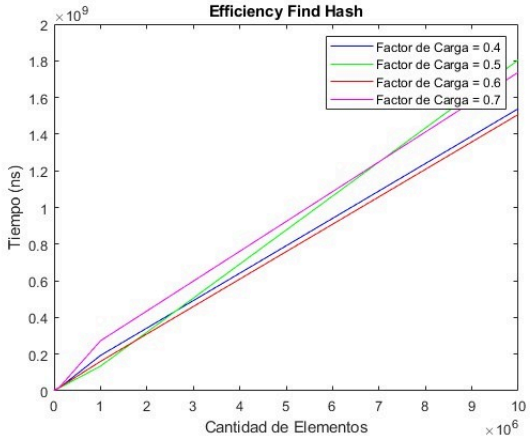
Remove:



En la gráfica, se muestra la eficiencia del tiempo de eliminación (en nanosegundos) en una tabla hash para diferentes factores de carga (0.4, 0.5, 0.6, y 0.7). A medida que se incrementa la cantidad de elementos, el tiempo de eliminación tiende a aumentar para todos los factores de carga, pero se observa que, para un factor de carga de 0.7, el tiempo crece de forma más pronunciada, especialmente a partir de los 9 millones de elementos. Esto puede deberse a que un factor de carga más alto implica más colisiones y mayor tiempo para realizar búsquedas y eliminaciones, lo que degrada el rendimiento. Los factores de carga menores, como 0.4 y 0.5, ofrecen mejor eficiencia, ya que las colisiones son menos frecuentes, reduciendo el tiempo promedio de operación.

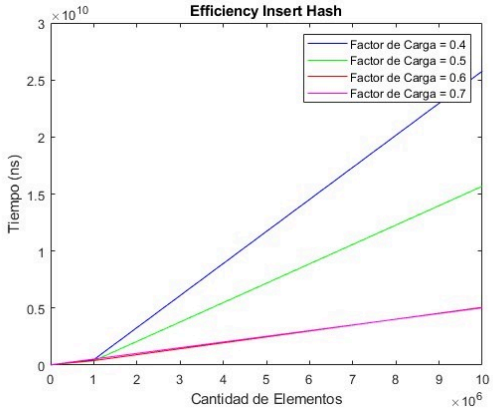
Find:

En la gráfica se analiza la eficiencia del método de búsqueda (find) en una tabla hash para diferentes factores de carga. Se observa un comportamiento similar al del gráfico anterior: a medida que aumenta la cantidad de elementos, el tiempo necesario para realizar una búsqueda también incrementa. Los factores de carga más altos, como 0.7, muestran una tendencia a tener mayores tiempos de búsqueda, especialmente en los últimos millones de elementos, debido al incremento en las colisiones. Los factores de carga más bajos, como 0.4 y 0.5, mantienen un mejor rendimiento ya que distribuyen mejor los elementos en los buckets, reduciendo las colisiones y los tiempos de búsqueda.



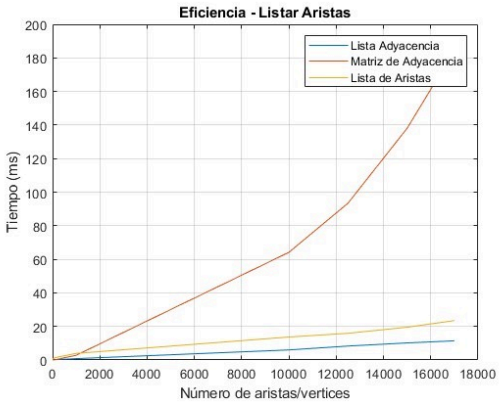
Insert:

Para factores de carga más bajos, como 0.4, el tiempo de inserción es mayor debido a que la tabla hash debe redimensionarse con más frecuencia. Esto sucede porque, al alcanzar el factor de carga límite, la tabla se redimensiona (duplicando su capacidad), lo que implica copiar todos los elementos existentes a un nuevo array de mayor tamaño. En cambio, para factores de carga más altos (0.7), el número de redimensionamientos es menor, ya que se permite una mayor densidad de elementos antes de que ocurra la redimensión, lo que hace que el tiempo de inserción sea más eficiente en general.



Grafos: Para las pruebas relacionadas a la estructura de datos *grafos*, evaluamos las tres implementaciones sugeridas: *lista de adyacencia, matriz de adyacencia* y *lista de aristas*. Para las anteriores implementaciones, evaluamos tres métodos: *listar aristas, verificar arista* y *listar vecinos.*

Listar Aristas

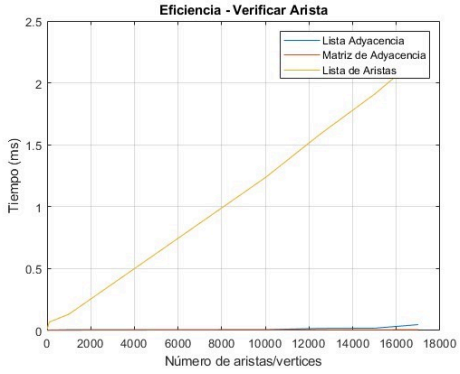


La lista de adyacencia se presenta como la más rápida para listar vecinos de un nodo en comparación con la lista de aristas. Esto se debe a que la lista de adyacencia permite acceder directamente a los nodos adyacentes, logrando un tiempo de operación proporcional al grado del nodo.

En cambio, la lista de aristas requiere recorrer toda la lista para identificar las conexiones, lo que implica un costo de O(E), donde E es el número total de aristas.

La matriz de adyacencia, aunque ofrece un acceso constante O(1) para verificar la existencia de una arista entre dos nodos, consume más memoria y se vuelve menos eficiente cuando se trata de listar todos los vecinos de un nodo, ya que el tiempo requerido es O(V) (donde V es el número de vértices), lo que puede resultar en un rendimiento más lento en grafos dispersos.

Verificar Arista

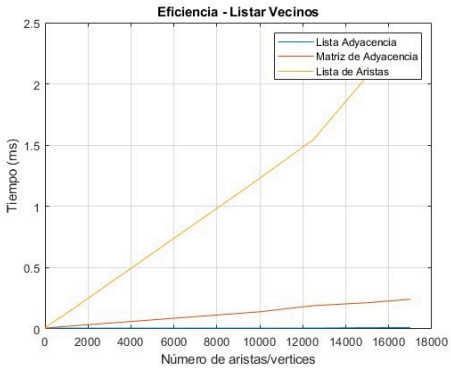


La matriz de adyacencia resulta la más eficiente para verificar aristas. Esto se debe a que permite acceder a la existencia de una arista en tiempo constante O(1), dado que el acceso a cualquier elemento de la matriz se realiza directamente mediante índices. Sin embargo, hay que tener en cuenta que este método requiere de más memoria.

En el caso de la lista de adyacencia, la verificación de la existencia de una arista implica recorrer la lista de vecinos del nodo, lo que resulta en un tiempo proporcional al grado del nodo O(grado).

En la lista de aristas, se requiere recorrer toda la lista para buscar la arista, lo que tiene un costo de O(E) (donde *E* es el número total de aristas).

Listar Vecinos



Listar los vecinos en una matriz de adyacencia muestra un tiempo de ejecución lineal respecto al número de vértices *n*, independientemente del número real de vecinos que un vértice tenga, lo que puede resultar costoso en grafos con pocos vecinos por vértice.

Por su parte, la eficiencia de listar vecinos en una lista de adyacencia depende directamente del número de conexiones de cada vértice, haciendo que este método sea óptimo para grafos dispersos, con un tiempo de ejecución lineal en relación al grado del vértice.

Por último, listar los vecinos en una lista de aristas es la menos eficiente de estas representaciones, especialmente en grafos grandes, ya que siempre se necesita recorrer todas las aristas, sin importar cuántos vecinos tenga el vértice.

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

Video adjunto a la entrega via moodle.

XI. Roles y actividades

INTEGRANTE	ROL(ES)	ACTIVIDADES REALIZADAS
Cristian Fabian Martinez Bohórquez	Técnico	Pruebas técnicas.
		Implementación de estructuras.
	Coordinador	Código al prototipo del proyecto.
Edward Jeisen Jair Arévalo Peña	Observador	Pruebas técnicas.
		Código al prototipo del proyecto.
	Animador	Contacto permanente

Juan David Cruz Giraldo	Investigador	Implementación de estructuras.
		Documento escrito.
	Experto	Diseñador

XII. DIFICULTADES Y LECCIONES APRENDIDAS

El mayor desafío fue hacer que diferentes estructuras, como listas y tablas hash, trabajasen juntas sin problemas. Esto puede ser complicado porque cada estructura tiene su propio funcionamiento, y asegurarse de que intercambien información correctamente requiere de atención especial. Adicionalmente, crear una serie de pruebas para asegurarse de que el sistema funcione bien, tanto en situaciones normales como en casos inusuales.

XIII. REFERENCIAS BIBLIOGRÁFICAS

[1] atlassian, Learn Git with Bitbucket Cloud, atlassian, disponible en https://www.atlassian.com/git/tutorials/learn-git-with-bitbucket-cloud, accedido el: 28 de marzo de 2024.

[2] reqtest, Why is the difference between functional and Non-functional requirements important?, reqtest, disponible en https://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/, accedido el: 29 de marzo de 2024.

[3] Y. D. Chong, Estructuras de datos secuenciales, espanol.libretexts, disponible en https://espanol.libretexts.org/Fisica/Física_Matemática_y_Pedagogía/Física_Computacional_(Chong)/02%3A_Tutorial_de_Scipy_(Parte_2)/2.01%3A_Estructuras_de_datos_secuenciales, accedido el: 29 de marzo de 2024.

[4] Streib, J. T., & Soma, T. (2017). Guide to Data Structures. Springer International Publishing.