Poptr

Santiago Suárez Aguilar, Nicolas Arevalo Rodriguez, Angel González Bejarano

No. de Equipo Trabajo: 3

I. INTRODUCCIÓN

En el presente documento se describe una alternativa de solución a un problema común en la cotidianidad, el cual está asociado al proceso que conllevan los trámites en Colombia. Debido a la burocracia y la baja eficiencia en términos de tiempo para el desarrollo de los mismos, se convierten en una tarea tediosa y demandante. Durante el documento, se describirán los principales requerimientos del software, la alternativa de solución propuesta con sus especificaciones técnicas incluyendo un análisis y prueba de la misma.

II. DESCRIPCIÓN DEL PROBLEMA A RESOLVER

Según el BID por sus siglas, Banco Interamericano de Desarrollo, en el año 2018 en Colombia se necesitaban 7.4 horas para completar un trámite, situando a nuestro país en el top tres de países latinoamericanos con más tiempo en realizar un trámite [1].

El porcentaje de trámites completamente digitales en nuestro país es del 4%, una cifra que está en el promedio de la región situándose esta en 3.7% [1]. Aun así, existe todavía mucho potencial para implementar la tecnología en la simplificación de trámites, asumiendo que ya el Estado mismo ha hecho una simplificación de manera legal, es decir, ha reducido la complejidad de los trámites y los requerimientos que se suponen para ellos.

Estamos hablando de que el potencial que tienen los trámites digitales en nuestro país es del 35%, es decir, trámites que se pueden empezar y completar en línea.

Mientras esto ocurre, y la brecha digital que en nuestro país existe, se reduce, lo que se plantea con esta idea es optimizar los trámites presenciales ya existentes y darles las herramientas para poder simplificarlos, conectando a los usuarios que realizan estos trámites de una manera fácil, sencilla y eficaz.

III. USUARIOS DEL PRODUCTO DE SOFTWARE

El software contará con dos tipos de usuarios, los cuales se caracterizan dependiendo de los privilegios de acceso a la información y a las diferentes funcionalidades del software. Por un lado estarán las personas que quieren agendar un trámite o interactuar con otros usuarios a través de la red social, ellos tendrán acceso a la información publicada por los demás, además podrán agendar turnos, crear publicaciones y comentar las mismas. Por otro lado, estarán los administradores, los cuales podrán acceder a las colas de preguntas, podrán eliminar contenido que consideren erróneo o inoportuno, además de manipular las listas de usuarios.

IV. REQUERIMIENTOS FUNCIONALES DEL SOFTWARE

- *Registro de usuarios:* Recopilación de datos personales y de contacto que faciliten la interacción con el usuario. El usuario debe proporcionar datos verídicos con el fin de facilitar el contacto con el mismo. En el caso de que el usuario incorpore datos con un formato erróneo, se indicará que la entrada no es válida.

- *Reporte de usuarios:* Listado de todos los usuarios registrados, esta funcionalidad le permite al administrador conocer los datos de sus usuarios.

- *Búsqueda de usuario*: Búsqueda parcial de un usuario a partir de su correo, retornará el nodo con la información del usuario en cuestión.

- *Actualización de usuario:* El usuario puede consultar y actualizar su nombre, descripción y foto desde su perfil.

- *Eliminación de usuario*: A partir del correo del usuario, se puede eliminar dicho nodo de la lista..

- *Validación de usuarios:* Consulta y verificación en la base de datos de que el usuario exista. Es necesario conozca sus credenciales para acceder al sistema.

- *Creación de publicación:* Al ser una red social, es necesario establecer una funcionalidad que le permita a los usuarios publicar noticias, preguntas o información pertinente que corresponda a la temática deseada.

- *Eliminación de publicación:* Esta funcionalidad responde a la necesidad de quitar publicaciones que contengan información errónea, desactualizada o contenido inoportuno por parte del usuario. Esta funcionalidad estará disponible para los administradores, que bajo sus criterios y normas de uso del sitio, establecerá la validez de la publicación.

- *Agendamiento de trámite:* Los usuarios podrán agendar los trámites disponibles dentro del sitio, en esta sección a

partir del tiempo de duración del trámite, se le asignará una prioridad dinámica a medida que pase el tiempo.

- *Extracción de trámites en la cola:* Para depurar el agendamiento de trámites, una vez pase el tiempo de duración del trámite, el trámite se desencola puesto que posee la menor prioridad.

- *Obtención de los hashtags en tendencia:* Con el fin de proporcionarle información útil al usuario, el usuario podrá acceder a los hashtags que son tendencia, para mantenerse informado de la actualidad de los trámites.

- *Creación de notificaciones:* A traves del software se podrán crear notificaciones al usuario con el fin de mantenerlo informado.

- *Desapilamiento de notificaciones:* El usuario verá las notificaciones al iniciar sesión, y después de un tiempo las notificaciones se desapilan.

- *Formulación de preguntas:* El usuario tendrá la posibilidad de preguntarle directamente al perfil de administrador sobre los trámites

- *Desencolamiento de preguntas:* El administrador una vez haya respondido la pregunta formulada por el usuario, podrá desencolar la pregunta, para de esta forma depurar la cola de preguntas.

V. DESCRIPCIÓN DE LA INTERFAZ DE USUARIO PRELIMINAR

Se hará uso de una interfaz gráfica que facilitará la interacción del usuario con el software y los demás usuarios. La pantalla inicial permitirá que el usuario se registre o en el caso de que esté registrado, valide su sesión. En la pantalla de registro, el usuario podrá introducir sus datos, los cuales serán usados para la validación de la sesión y además para contactarlo en caso de que agende un trámite.

Una vez el usuario valide su sesión podrá acceder a la sección de publicaciones y trámites. En la sección de trámites el usuario podrá crear publicaciones, y también podrá comentar las publicaciones de los demás. Además tendrá las notificaciones del sitio y podrá observar los hashtags en tendencia.

En la sección de trámites el usuario podrá ver un listado de trámites que puede realizar con la información y documentación requerida, para de esta forma elegir el trámite, el cual desea agendar. Por otro lado, el usuario también podrá observar los trámites que están en curso (agendados).

En la sección de perfil, el usuario podrá preguntar al administrador, y además actualizar sus datos personales.

Finalmente, en el caso de la persona que tiene el rol de administrador, esta tendrá habilitada una sección adicional, la cual responde a el manejo de la cola de preguntas, en esta se podrá responder y eliminar preguntas en la cola. Adicionalmente, tendrá habilitada la opción de eliminar publicaciones, buscar usuarios y tendrá un reporte de los usuarios existentes.

El mockup de la interfaz gráfica descrita se puede ver en el siguiente link: https://xd.adobe.com/view/5e4308ae-9b6b-4553-647c-fae6f52ea6a8-2872/

VI. ENTORNOS DE DESARROLLO Y DE OPERACIÓN

El software se desarrolló en Typescript y la interfaz gráfica se realizó a partir de una librería de Javascript llamada React. El IDE elegido fue Visual Studio Code y además como entorno de ejecución se usó Node.js. El servidor creado como *local host* se aloja en el navegador Google Chrome durante el tiempo de ejecución.

El sistema operativo en el que se desarrolla la operación es Windows 10 y las especificaciones del hardware son:

- Procesador: Intel Core i7-9750H 2.6GHz
- RAM: 16 GB DDR4

VII. DESCRIPCIÓN DEL PROTOTIPO DE SOFTWARE

El prototipo de software se compone en gran parte del desarrollo de la interfaz gráfica de usuario, donde se puede ver la aplicación e implementación de las estructuras de datos utilizadas como parte de esta. Para esta entrega se realizó la implementación de una tabla hash, la cual se destinó a realizar la operación de búsqueda de usuarios, que para la primera entrega se realizó a partir de una lista enlazada. También se añadieron otras operaciones como la eliminación y actualización de usuarios para su respectiva comparación con los métodos de la lista implementada inicialmente. Teniendo en cuenta lo anterior, se recibe cada usuario de un arreglo con el conjunto de datos, y se posiciona en el nuevo arreglo a partir de la funcion de hashing del metodo **getPolyHash**. Se debe tener en cuenta que para el control de las colisiones se realizó la implementación con listas enlazadas en las posiciones llenas del arreglo, es decir, al insertar un dato por primera vez en una determinada posición del arreglo, se crea una lista enlazada a la que se agrega el elemento insertado, luego, en el caso de una colisión, el nuevo elemento se agrega a la cola de la lista enlazada existente; basados en esto, para realizar la búsqueda de un usuario específico, se obtiene su código hash o su posición (en este caso a partir del correo) y luego se recorre la lista correspondiente a esa posición del arreglo para encontrarlo. Es necesario recalcar que debido a que la búsqueda se realiza directamente en la lista encadenada, los métodos implementados para la búsqueda, eliminación y actualización son parte de la clase LinkedList, por lo tanto se

encuentran en el archivo **LinkedList.ts**, ubicado en la carpeta *classes*. Por otro lado, la implementación de la tabla hash se encuentra en la misma carpeta como **HashTable.ts** y los cambios en la implementación para la búsqueda de los usuarios se pueden evidenciar en el archivo **Admin.tsx**, perteneciente a *components*.

A continuación, se describen brevemente las operaciones principales que soporta la tabla hash implementada y las operaciones ya mencionadas pertenecientes a la lista enlazada:

- **getPolyHash = (s: string, p: number):** Permite generar el hash de un string teniendo en cuenta el tamaño del arreglo.
- **add = (element: UserProps):** Permite añadir un usuario a la tabla, creando una lista enlazada con este o agregando el elemento a la lista en caso de que ya exista una en esta posición.
- **searchUser= (user :String):** Permite buscar un usuario en la lista a partir de su correo.
- **updateN = (user: String, prevdata : String, newdata: String):** Permite actualizar los datos de un usuario en la lista; para esto, busca al usuario mediante el método searchUser y cambia un dato existente por uno nuevo.
- **removeht (user : String):** Permite remover un usuario de la lista. Se utiliza el método searchUser para buscar el usuario y se cambian los debidos apuntadores como corresponda para su eliminación.

Así, las estructuras que conforman el prototipo final de software son: lista enlazada, utilizada para el manejo de comentarios en las publicaciones, además de la construcción de otras estructuras y funcionalidades de estas; tabla hash, que permite la búsqueda de usuarios en la plataforma; cola de prioridad de tipo min-heap, que permite encolar los diferentes trámites y remover el elemento en la primera posición cuando termina su tiempo; árbol AVL, para la construcción de la sección de trending, mostrando los dos hashtags más utilizados entre todas las publicaciones; cola estándar, que permite encolar y desencolar preguntas de los usuarios hacia el administrador; pila, que permite mostrar las notificaciones y por último, arreglos para el manejo de publicaciones, algunos datos provenientes de la base de datos, entre otros.

Teniendo en cuenta lo anterior, en el archivo **Admin.tsx** se hace uso de las estructuras de tabla hash y cola estándar, este corresponde a la sección del usuario de tipo administrador. Por otra parte, en el archivo **New.tsx** se utiliza la estructura de lista enlazada, correspondiente a los comentarios; el árbol AVL es utilizado en el archivo **NewsSection.tsx** para las sección de trending; la pila es utilizada en los archivos **NewsSection.tsx y Notifications.tsx** para la sección de notificaciones y en el archivo **Turns.tsx** se utiliza la cola de prioridad para el manejo de la cola de trámites. Basados en esto, se tienen las ventanas para usuarios finales y los administradores.

Por último, se debe resaltar el archivo **Landing.tsx**, correspondiente a la ventana de inicio de la aplicación, donde se realiza el registro y la autenticación de usuarios.

La realización de las pruebas se da a partir de arreglos de datos importados desde archivos .json, de la misma manera que se realizaron en entregas anteriores. La estructura general del prototipo se compone de las diferentes implementaciones de las estructuras de datos, que se encuentran en la carpeta *classes,* ubicada en *src,* además en esta carpeta también se encuentran los diferentes archivos para la realización de las pruebas, que poseen extensión .js. Por otro lado, los archivos para la realización de la aplicación y su componente gráfica se encuentran en la carpeta *components*, ubicada en *src* y los archivos de datos con extensión .json se encuentran en la carpeta *data* de la misma ubicación. Por último, las hojas de estilo para la interfaz gráfica (archivos css), están ubicadas en la carpeta *styles* en *src.*

Así, para la realización de las pruebas, se importan las estructuras implementadas al igual que los arreglos de datos, en cada archivo .js correspondiente para aplicar sus métodos sobre diferentes cantidades de objetos dependiendo de la prueba. Luego se exportan los métodos de las pruebas para su posterior llamado en el landing, lo cual muestra el contenido de la prueba en consola. Así, para la prueba de 1 millón de datos de la tabla hash, por ejemplo, en el landing se ve importada una función que crea una instancia de la tabla, toma los objetos del archivo usersData04, los inserta a esta y luego realiza las diferentes operaciones.

A continuación se encuentran los enlaces del repositorio y de la aplicación web respectivamente:

- **https://github.com/angegonzalez/Poptr**
- **https://modest-chandrasekhar-9dd1a6.netlify.app/**

VIII. PRUEBAS DEL PROTOTIPO

Con el fin de cuantificar los tiempos que toma correr cada funcionalidad, se usaron los métodos console.time() y console.timeEnd(), estos permiten calcular el tiempo que toma correr el código en la consola del navegador.

Las funcionalidades a probar se implementaron en las diferentes estructuras aprendidas durante el curso. Para dichas funcionalidades se realizaron pruebas con:

- 10000 datos
- 100000 datos
- 500000 datos
- 1000000 datos
- 10000000 datos
- 50000000 datos

Se omitió la prueba de orden 10^7 datos en la estructuras de lista y tabla hash, por las razones que serán descritas en la sección de dificultades. Para las demás estructuras se omitió la prueba de 500000 datos.

A continuación se presentan las tablas comparativas y los gráficos para cada una de las funcionalidades elegidas para tal fin, clasificadas por tipo de estructura:

1) Lista enlazada y tabla de usuarios

Para esta entrega se sustituyó la lista enlazada por una tabla hash de usuarios. A continuación se hará una comparación de las dos funcionalidades para cada estructura.

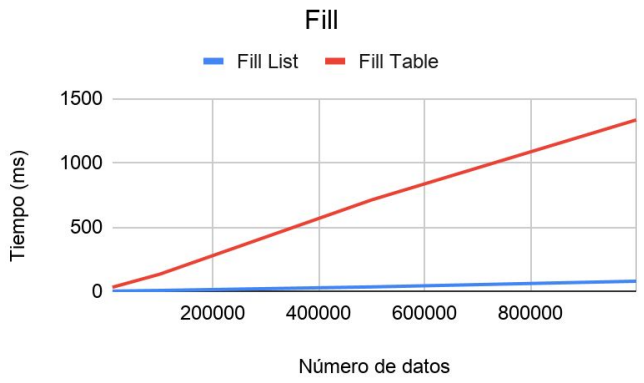
- Funcionalidad: Registro de usuarios (Fill List y Fill Table)

Tiempos para la lista

Número de datos	10000	100000	500000	1000000
Tiempo (ms)	1,89	7,39	34,67	79,03

Tiempos para la tabla hash.

Número de datos	10000	100000	500000	1000000
Tiempo (ms)	32,02	134,25	711,12	1332,82



Se observa que el tiempo que toma llenar la lista de usuarios es lineal ya que se debe repetir la operación pushBack() n veces. Por lo anterior se puede concluir que el tiempo que toma la operación pushBack() es constante, es decir, pushBack() es de complejidad O(1).

Por otro lado se observa que el tiempo que toma llenar la tabla hash es lineal, ya que se debe acceder al arreglo n veces. Sin embargo, dado que el arreglo es dinámico y depende del factor de carga, la constante que acompaña al comportamiento lineal es grande en comparación a la de Fill List. Por lo anterior se puede concluir que a pesar de que las dos son lineales, la inserción de un usuario es más costosa en la tabla hash.

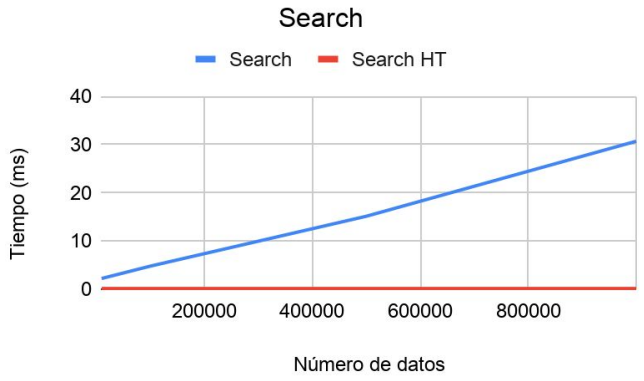
- Funcionalidad: Búsqueda de usuario (Search y Search HT)

Tiempos para la lista

Número de datos	10000	100000	500000	1000000
Tiempo (ms)	2,25	4,81	15,15	30,71

Tiempos para la tabla hash.

Número de datos	10000	100000	500000	1000000
Tiempo (ms)	0,17	0,18	0,18	0,17



Se observa que en esta funcionalidad al ser necesario recorrer la lista para acceder al nodo que se busca, el tiempo que toma es lineal. Esta prueba se realizó para el peor de los casos, por lo tanto, el acceso a elementos que estén más cercanos a la cola o a la cabeza tomarán un tiempo menor.

Por otro lado, se observa que para la tabla hash el tiempo de acceso es constante, ya que se accede al arreglo a partir del índice con el hash calculado.

- Funcionalidad: Actualización de usuario (Update y Update HT)

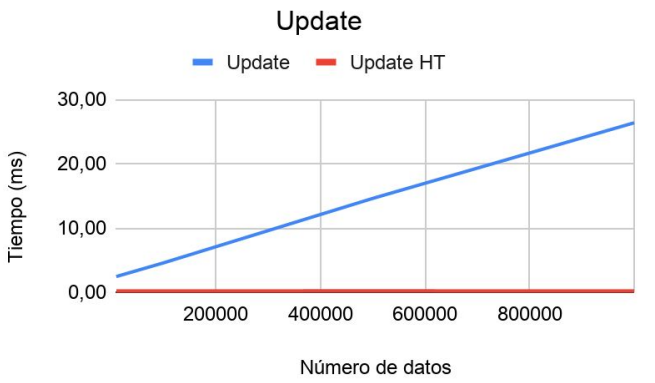
Tiempos para la lista

Número de datos	10000	100000	500000	1000000
Tiempo (ms)	2,46	4,58	14,60	26,37

Tiempos para la tabla hash

Número de datos	10000	100000	500000	1000000

Tiempo (ms)	0,24	0,23	0,24	0,23



Se observa que en esta funcionalidad al ser necesario recorrer la lista para acceder al nodo que se busca, el tiempo que toma es lineal. En comparación a la funcionalidad anterior, se puede resaltar que toma un poco más de tiempo debido a que además de realizar la operación de búsqueda, tiene que realizar la operación de actualizar.

Por otro lado, se observa que para la tabla hash el tiempo de actualización es constante, ya que se accede al arreglo a partir del índice con el hash calculado y se actualiza la información requerida. Al igual que en la lista, se observa que toma un poco más de tiempo que el search debido a que además de realizar la operación de search, tiene que realizar la operación de actualizar.

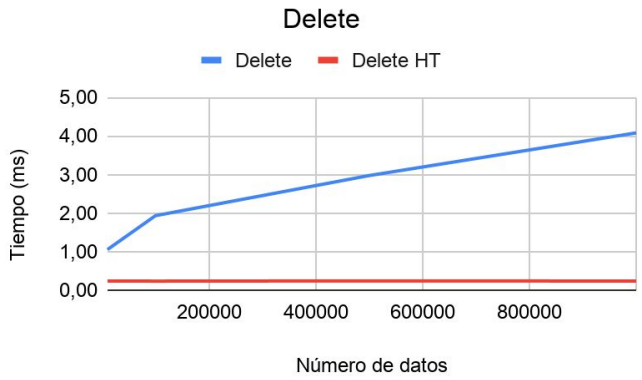
- ❖ Funcionalidad: Eliminación de usuario usando índice (Delete y Delete HT)

Tiempos para la lista

Número de datos	10000	100000	500000	1000000
Tiempo (ms)	1,06	1,94	2,98	4,08

Tiempos para la tabla hash

Número de datos	10000	100000	500000	1000000
Tiempo (ms)	0,24	0,24	0,25	0,24



Se observa que en esta funcionalidad al ser necesario recorrer la lista para acceder al nodo que se busca a partir del índice, el tiempo que toma es lineal. La prueba se realizó para el peor de los casos. Por otro lado, se observa que para la tabla hash el tiempo de eliminación es constante, ya que se accede a partir del índice con el hash calculado para el usuario a eliminar.

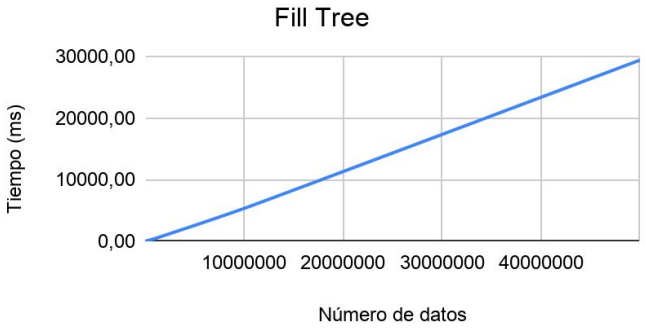
El resumen de la complejidad operacional en la notación Big O de cada una de las funcionalidades se encuentra a continuación:

Funcionalidad	Tipo de estructura	Complejidad
Fill	Lista enlazada	O(n)
	Tabla Hash	O(1)
Search	Lista enlazada	O(n)
	Tabla Hash	O(1)
Update	Lista enlazada	O(n)
	Tabla Hash	O(1)
Delete	Lista enlazada	O(n)
	Tabla Hash	O(1)

2) Árbol AVL para tendencias.

- ❖ Funcionalidad: Registro de hashtags (Fill Tree)

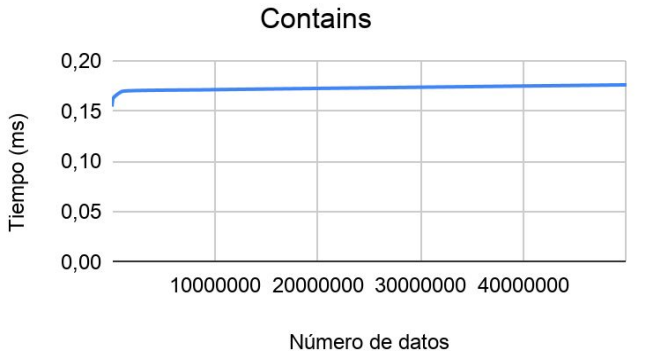
Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	10,42	50,01	452,13	5333,00	29395,56



Se observa que el tiempo que toma llenar el árbol de hashtags es de complejidad O(nlogn), ya que se debe repetir la operación insert() n veces. Por lo anterior se puede concluir que el tiempo que toma la operación insert() es de tiempo logarítmico, es decir, insert() es de complejidad O(logn).

- Funcionalidad: Verificación de existencia (Contains)

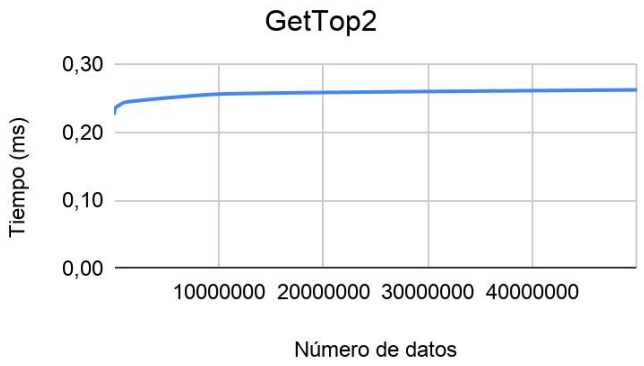
Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	0,15	0,16	0,17	0,17	0,18



Se observa que el tiempo que toma saber si un nodo está contenido en el árbol es logarítmico debido a que al cumplir la propiedad AVL, buscar un nodo tiene complejidad O(logn), a diferencia de los árboles BST sin balancear, en los que esta operación tiene un costo operacional de O(n).

- Funcionalidad: Obtención de los 2 hashtags en tendencia (GetTop2)

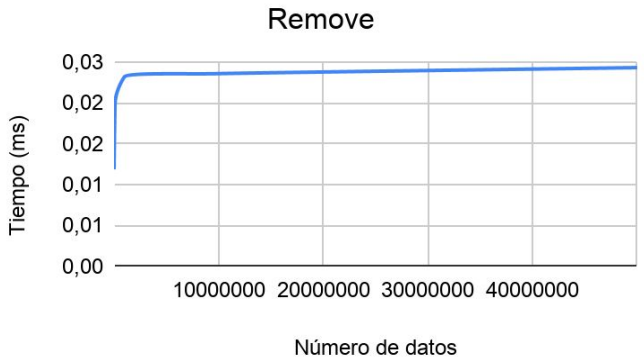
Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	0,23	0,23	0,24	0,26	0,26



Se observa que el tiempo que toma obtener los hijos del nodo más a la derecha (que no es una hoja) es logarítmico debido a que al cumplir la propiedad AVL y al ser un BST, buscar un nodo tiene complejidad O(logn).

- Funcionalidad: Eliminación de un hashtag del árbol (Remove)

Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	0,01	0,02	0,02	0,02	0,02



Se observa que el tiempo que toma eliminar un nodo está contenido en el árbol es logarítmico debido a que al cumplir la propiedad AVL, buscar un nodo tiene complejidad O(logn). A diferencia de las otras funcionalidades, esta operación posee constantes menores que marcan la diferencia en los tiempos medidos.

El resumen de la complejidad operacional en la notación Big O de cada una de las funcionalidades se encuentra a continuación:

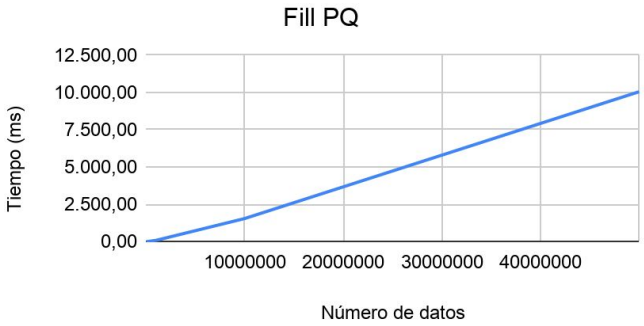
Funcionalidad	Complejidad
Fill	O(nlogn)
Contains	O(logn)

GetTop2	O(logn)
Remove	O(logn)

3) Cola de prioridad para trámites

❖ Funcionalidad: Registro de turnos (Fill PQ)

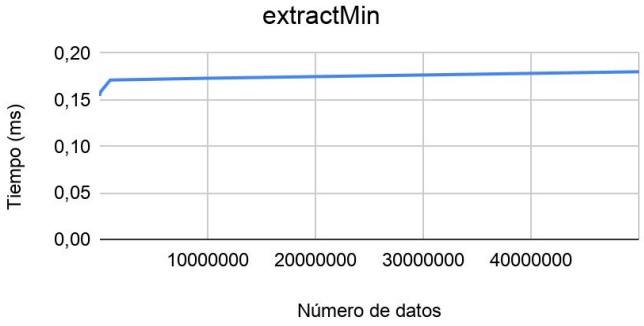
Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	3,26	12,53	95,57	1.557,53	10.044,07



Se observa que el tiempo que toma llenar el heap de trámites es de complejidad O(nlogn), ya que se debe repetir la operación insert() n veces. Por lo anterior se puede concluir que el tiempo que toma la operación insert() es de tiempo logarítmico, es decir, insert() es de complejidad O(logn).

❖ Funcionalidad: Extracción del elemento con menor prioridad (extractMin)

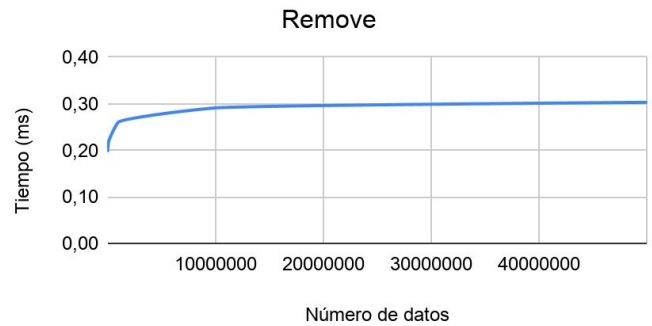
Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	0,15	0,16	0,17	0,17	0,18



Se observa que el tiempo que toma extraer el elemento de menor prioridad el heap de trámites es de tiempo logarítmico, ya que a pesar de que obtener el elemento de menor prioridad es de tiempo constante, al extraerlo, se hace necesario reemplazarlo por la última hoja y aplicar la operación siftDown() que es de tiempo logarítmico. Por lo anterior, se puede concluir que la complejidad de la operación extractMin() es O(logn).

❖ Funcionalidad: Eliminación un elemento de la cola (Remove)

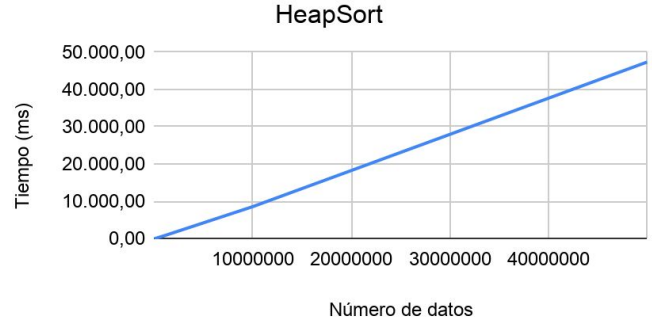
Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	0,20	0,22	0,26	0,29	0,30



En el caso de remove(), se observa que a pesar de que es de tiempo logarítmico, toma más tiempo que la funcionalidad extractMin(). Esto se debe a que para eliminar un elemento del heap es necesario realizar la operación de extractMin() y además la operación de siftUp(), por lo tanto las constantes de esta funcionalidad son mayores en comparación a la anterior.

❖ Funcionalidad: Ordenamiento de la cola de prioridad (HeapSort)

Número de datos	10000	100000	1000000	10000000	50000000
HeapSort	14,27	69,79	800,02	8.606,74	47.318,91



Se observa que el tiempo que toma ordenar el heap de trámites es de complejidad O(nlogn), ya que se debe repetir la operación siftDown() n veces dado que se realiza en el mismo heap, es decir es *in place.*

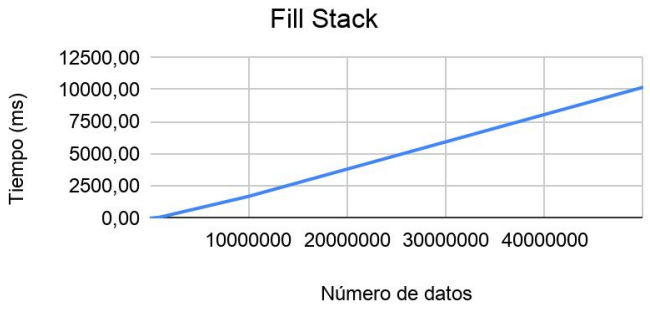
El resumen de la complejidad operacional en la notación Big O de cada una de las funcionalidades se encuentra a continuación:

Funcionalidad	Complejidad
Fill PQ	O(nlogn)
extractMin	O(logn)
Remove	O(logn)
HeapSort	O(nlogn)

4) Pila de notificaciones

- Funcionalidad: Inserción elemento a la pila (Fill Stack)

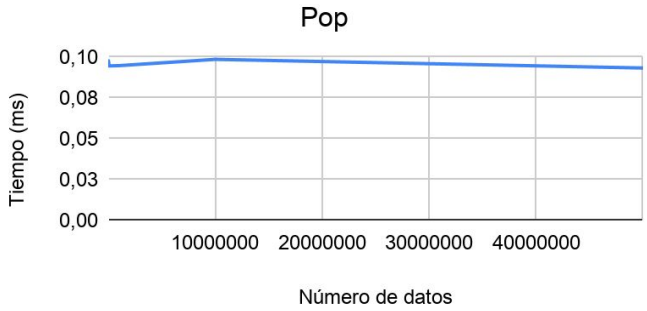
Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	2,68	12,85	81,28	1702,59	10189,42



Se observa que el tiempo que toma llenar la pila de notificaciones es de complejidad O(n), ya que se debe repetir la operación push() n veces. Dado que la implementación de la pila se hace sobre una lista enlazada, la operación push() es de tiempo constante ya que se solo se debe cambiar los apuntadores de la cabeza de la lista.

- Funcionalidad: Eliminación elemento de la pila (Pop)

Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	0,10	0,09	0,09	0,10	0,09



Se observa que el tiempo que toma desapilar el último elemento de la pila de notificaciones es de complejidad O(1), debido que la implementación de la pila se hace sobre una lista enlazada y solo se debe cambiar los apuntadores de la cabeza de la lista.

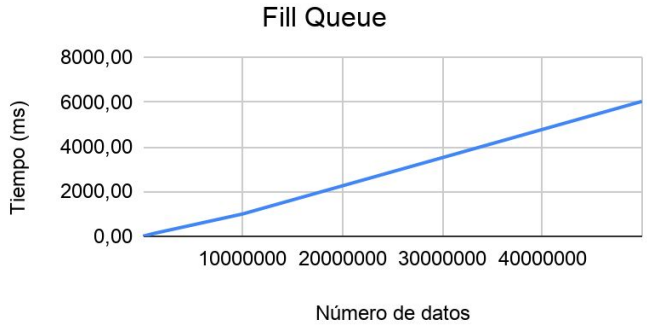
El resumen de la complejidad operacional en la notación Big O de cada una de las funcionalidades se encuentra a continuación:

Funcionalidad	Complejidad
Fill Stack	O(n)
¨Pop	O(1)

5) Cola de preguntas

- Registro de las preguntas (Fill Queue)

Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	2,45	11,76	117,87	999,84	6034,72



Se observa que el tiempo que toma llenar la cola de preguntas es de complejidad O(n), ya que se debe repetir la operación enqueue() n veces. Dado que la implementación de la pila se hace sobre una lista enlazada, la operación enqueue() es de tiempo constante ya que se solo se debe cambiar los apuntadores de la cola de la lista.

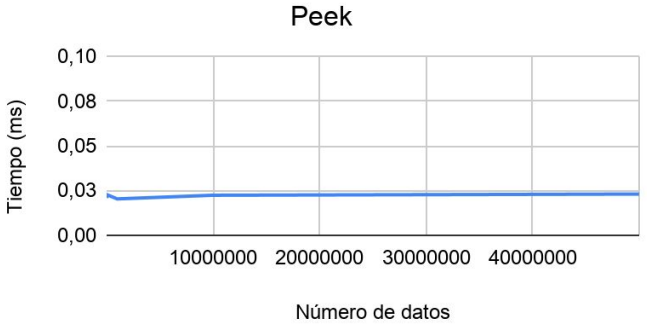
- Eliminación de la pregunta más antigua de la cola (Dequeue)

Número de datos	10000	100000	1000000	10000000	50000000
Tiempos (ms)	0,09	0,10	0,10	0,10	0,10

Se observa que el tiempo que toma desencolar un elemento de la cola es de complejidad O(1), debido que la implementación de la cola se hace sobre una lista enlazada y solo se debe cambiar los apuntadores de la cabeza de la lista.

- Visualización pregunta más antigua (Peek)

Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	0,02	0,02	0,02	0,02	0,02



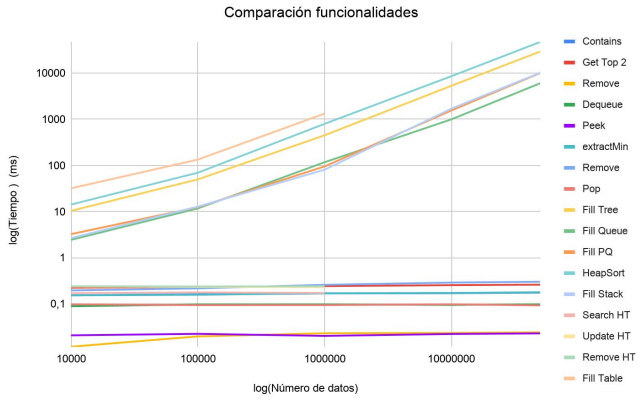
Se observa que el tiempo que toma visualizar el primer elemento de la cola es de complejidad O(1), debido que la implementación de la cola se hace sobre una lista enlazada y solo se debe retornar la información de la cabeza de la lista. En comparación con la operación de dequeue, al no tener que actualizar los apuntadores, las constantes son menores y por lo tanto los tiempos también.

El resumen de la complejidad operacional en la notación Big O de cada una de las funcionalidades se encuentra a continuación:

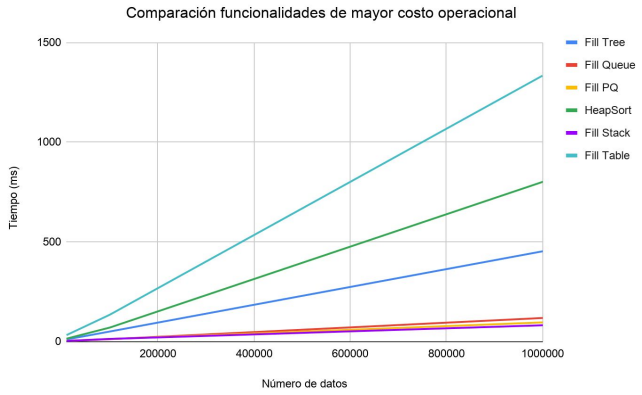
Funcionalidad	Complejidad
Fill Queue	O(n)
Dequeue	O(1)
Peek	O(1)

I. Análisis Comparativo

Para el análisis comparativo, inicialmente se realizó una gráfica en escala logarítmica en los dos ejes, de los tiempos de ejecución vs el número de datos para todas las estructuras y sus respectivas funcionalidades, con el fin de seleccionar las de mayor costo operacional y tener una descripción general del software. La gráfica descrita previamente se presenta a continuación:



De la gráfica anterior se observa que las funcionalidades que más tiempo toman son la operación Fill de todas las estructuras y el HeapSort de la cola de prioridad. A continuación se procede a realizar una gráfica de tan solo estas funcionalidades, la cual se observa a continuación:



De la gráfica se puede concluir, que la funcionalidad con mayor costo operacional es llenar la tabla hash de todos los usuarios (Fill Table), ya que al ser un arreglo dinámico que depende del factor de carga, por la cantidad de datos se debe realizar la duplicación del mismo múltiples veces, esto con el fin de hacer un uso eficiente de la memoria. Además también se observa, que estas funcionalidades al ser inevitable realizar la misma operación sobre todos los n datos, son las que poseen un mayor costo.

Para realizar una comparación más precisa, se recopiló la complejidad y el tipo de estructuras usadas en cada funcionalidad, en una tabla:

Nombre de la funcionalidad	Tipo(s) de estructura de datos	Análisis realizado (Notación Big O)
Fill PQ	Heap	O(nlogn)
Fill Stack	Cola	O(n)
Fill Table	Tabla Hash	O(n)
Fill Tree	Árbol binario	O(nlogn)
Fill Stack	Pila	O(n)
HeapSort	Heap	O(nlogn)

De la tabla se puede concluir que, a pesar de que intuitivamente y dado las características de las complejidades, la complejidad de O(nlogn) crece más rápido en comparación a la de O(n), el tiempo de ejecución de la funcionalidad Fill Table es mayor en comparación a las funcionalidades de Fill Tree y Fill PQ.

Esto se debe a que en el análisis asintótico con la notación Big O, se omiten las constantes que acompañan a las funciones que describen el tiempo en función de la cantidad de datos, por esta razón en este caso se puede concluir que la constante que está asociada a la funcionalidad Fill Table es lo suficientemente grande para que los tiempos crezcan más rápido que en las otras funcionalidades, para la muestra de datos tomada, dado que asintóticamente el crecimiento de nlogn va a ser mayor que n.

II. DIFICULTADES Y LECCIONES APRENDIDAS

Una de las principales dificultades durante el desarrollo fue las pruebas de carga, debido a que el software almacena su información en Firebase (base de datos) para la persistencia de los datos y dicha plataforma no permite almacenar dichos datos sin un costo adicional. Por ello, se deben realizar las pruebas en consola. Por otro lado, también fue necesario omitir las pruebas de 100 millones de datos para cada una de las estructuras, esto debido a que el heap de Node.js no permite hacer dicha prueba, ya que la cantidad de recursos necesarios para realizar la prueba de carga son muy altos; sin embargo, se realizaron pruebas de 50 millones de datos para el árbol AVL, la cola de prioridad, la cola y la pila. Por último, fue necesario omitir las pruebas de 10 y 100 millones de datos para la tabla hash, esto debido a la falta de recursos para generar archivos con tal cantidad de elementos diferentes, teniendo en cuenta que se deben utilizar los mismos tipos de datos utilizados para las pruebas de la lista enlazada.

Durante el desarrollo de esta entrega se evidenciaron las ventajas de la implementación de la tabla hash para optimizar los tiempos que toman ejecutar algunas funcionalidades del software.

IX. REFERENCIAS

[1]“BID- El fin del trámite eterno", BID, 2020. [Online]. Available: https://cloud.mail.iadb.org/fin_tramite_eterno?UTMM=Direct&UTMS=Website#el-problema-con-los-trámites. [Accessed: 13- Mar- 2020].