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 alguno de sus atributos, retornará el nodo con la información del usuario en cuestión

- *Actualización de usuario:* Búsqueda del usuario a partir de uno de sus atributos, y además se asigna un nuevo valor al atributo que se desee cambiar.

- *Eliminación de usuario*: A partir del índice en el cual se encuentra alojado el nodo del usuario, se puede eliminar dicho nodo de la lista. Es necesario conocer la posición del usuario previamente.

- *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.

- *Búsqueda de palabra en publicación:* Búsqueda parcial de palabras en la publicación para obtener información

que el usuario quiere obtener sobre un trámite o alguna funcionalidad del sitio.

- *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 acceder a la aplicación web, en esta ocasión se presenta una solución más conveniente para el usuario, la cual se construyó a partir del servicio de hosting Netlify; el enlace a la aplicación se presenta al final de la sección, junto con el link al repositorio. Por otro lado, para esta entrega se implementaron cuatro nuevas estructuras de datos. Entre las principales estructuras se encuentra la cola de prioridad, implementada como un Min Heap, la cual se utiliza para gestionar los diferentes trámites que se pueden hacer. Para este caso cada trámite posee un tiempo de ejecución, de manera que el usuario agenda sus trámites, los cuales se adecuan en la cola basándose en el tiempo, así, el trámite que posee el menor tiempo quedará en la primera posición de la cola; luego el usuario puede empezar el trámite de sus solicitudes, produciendo que comience el tiempo de cada uno, y cuando el tiempo de un trámite termina, se extrae el elemento de la cola. Por otro lado, se tiene un árbol AVL, utilizado para administrar las tendencias de hashtags sobre las

publicaciones, esto permite mostrar los dos hashtags más comunes en la sección de "Trending" a partir del método **getTop2()**. Además, a partir de la lista encadenada implementada en la anterior entrega, se implementaron una pila y una cola; ambas estructuras poseen los métodos característicos propios, es decir, los que permiten remover y añadir elementos. La cola estándar se utilizó para almacenar y manejar preguntas que el usuario puede hacer al administrador, de esta manera, cuando el administrador responde la pregunta, esta se retira de la cola mediante la operación **dequeue().**

Teniendo en cuenta el entorno de desarrollo, el desplazamiento a través de la aplicación se da a partir de links de navegacion, por tanto, se utilizó la implementación de la pila para almacenar tareas realizadas por el usuario y mostrar recordatorios de las secciones de la aplicación mediante notificaciones.

A continuación, se describen brevemente las operaciones principales que soporta el árbol AVL implementado:

- Creación del árbol: La raíz del árbol inicialmente es nula y el contador de sus nodos equivalente a cero.
- **insert (node: Node<T> | null, value: T | any):** Permite la inserción de un dato (value) a partir de un nodo dado. En este caso se compara el valor count del objeto que contiene el hashtag para posicionarlo en el árbol.
- **delete(node: Node<T> | null, element: T | any)**: Permite remover un nodo del árbol partiendo de un nodo dado y comparando con el elemento a remover. Se debe tener en cuenta que al remover un nodo, se debe actualizar la altura y factor de balance del nodo que lo reemplaza, y hacer un rebalanceo. (esto también aplica al insertar un nodo nuevo)
- **contains (value: T | any, node: Node<T> | null)**: Permite realizar la búsqueda de un dato partiendo de un nodo. El método retorna verdadera en caso de que se haya encontrado el nodo al que corresponde la data.
- **getTop2 ( )**: Retorna un arreglo con los dos mayores valores en el árbol. Para este caso, el método permite obtener los dos hashtags más utilizados entre todas las publicaciones.

De igual manera, algunas de los principales métodos de la cola de prioridad son:

- Creación de la cola de prioridad: Define el arreglo vacío a utilizar para la cola de prioridad con size igual a cero.
- **insert = (data: T)**: Inserta un elemento en la última posición a llenar del arreglo, y lo posiciona en la cola de acuerdo a su prioridad.
- **extractMin():** Permite remover el elemento en la primera posición de la cola, teniendo en cuenta que este tiene el valor de prioridad más bajo.
- **changePriority = (i: number, p: number):** Permite cambiar la prioridad de un elemento en la posición i de la cola por una prioridad p.
- **remove = (i: number):** Permite remover un elemento de la cola a partir de su posición.
- **heapSort()**: Realiza el algoritmo de ordenamiento a partir de la prioridad, en el mismo arreglo de la cola.

Algunos de estos métodos fueron utilizados únicamente al momento de generar las pruebas, debido a que no es necesaria su utilización en la aplicación.
Teniendo en cuenta que las implementaciones de la cola estándar y la pila se realizaron a partir de la lista enlazada implementada con anterioridad, se describen nuevamente las operaciones utilizadas para la implementación de las nuevas estructuras de datos, al igual que las principales operaciones funcionales de cada una de estas:

Lista enlazada

- **pushFront = (data: T)**: Permite insertar un nuevo nodo a partir de un dato en la cabeza de la lista
- **pushBack = (data: T)**: Permite insertar un nuevo nodo a partir de un dato en la cola de la lista.
- **removeNode(index : number)**: Permite remover un nodo de la lista a partir de la position en que se encuentra estrictamente en la lista.

Cola

- **enqueue(data: T):** Permite insertar un nodo con su data respectiva en la cola de la lista.
- **dequeue()**: Remueve el nodo en la primera posición de la lista, que corresponde al primero en la cola.

Pila

- **push(data: T)**: Permite insertar un nodo con su data respectiva en la primera posición de la lista.
- **pop()** : Remueve y retorna el elemento en la primera posición de la lista, es decir, el último añadido.

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 la última entrega, sin embargo, para las pruebas de 10 y 50 millones de datos se hace la inserción en cada estructura a partir de ciclos for, debido a la dificultad para generar archivos con tales cantidades de datos. Las diferentes implementaciones de las estructuras 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.

Teniendo en cuenta lo anterior, 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 del AVL tree, por ejemplo, en el landing se ve importada una función que crea una instancia del árbol, toma los objetos del archivo hashTagsData03, los inserta al árbol AVL y luego realiza diferentes operaciones sobre este.

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, 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 de usuarios

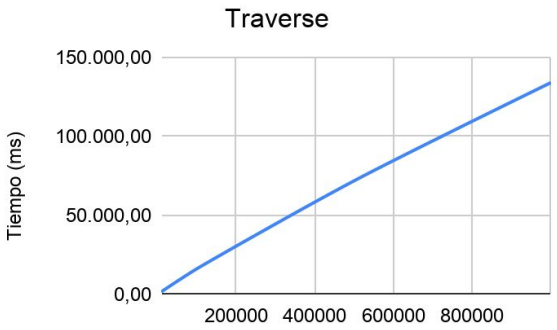
❖ Funcionalidad: Registro de usuarios (Fill List)

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

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).

❖ Funcionalidad: Reporte de usuarios (Traverse)

Número de datos	10000	100000	500000	1000000
Traverse	1.661,87	16187,48	71.907,57	134055,75



Se observa que en esta funcionalidad al ser necesario recorrer toda la lista para acceder e imprimir la información de los nodos, el tiempo que toma es lineal.

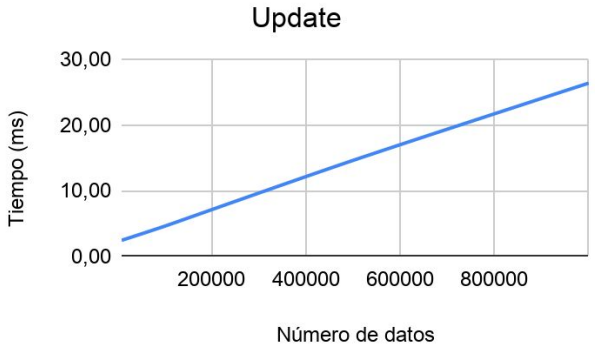
❖ Funcionalidad: Búsqueda de usuario (Search)

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

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.

- Funcionalidad: Actualización de usuario (Update)

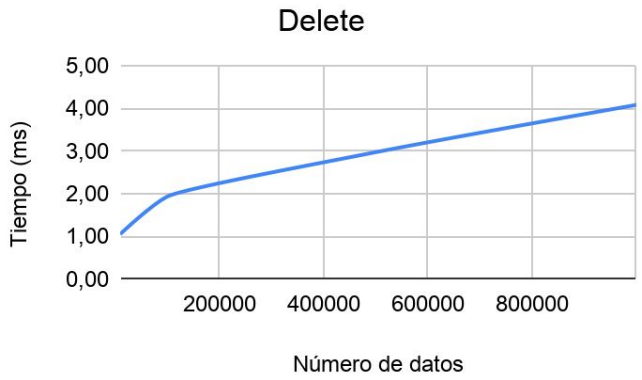
Número de datos	10000	100000	500000	1000000
Tiempo (ms)	2,46	4,58	14,60	26,37



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.

- Funcionalidad: Eliminación de usuario usando índice (Delete)

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



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.

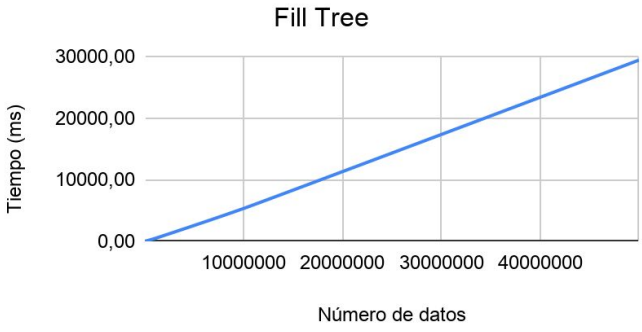
El resumen de la complejidad operacional en la notación Big O de cada una de las funcionalidades se encuentra a continuación:

Funcionalidad	Complejidad
Fill	O(n)
Search	O(n)
Update	O(n)
Delete	O(n)
Traverse	O(n)

2) Árbol AVL para tendencias.

- Funcionalidad: Registro de hashtags (Fill Tree)

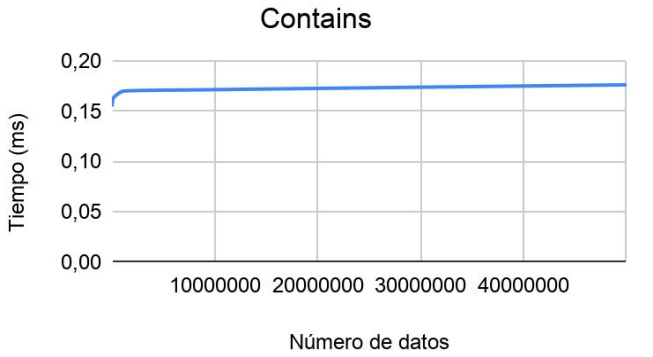
Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	10,42	50,01	452,13	5333,00	29395,56



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)

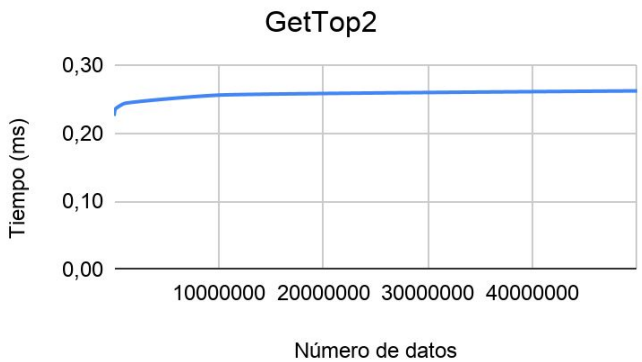
Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	0,15	0,16	0,17	0,17	0,18



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)

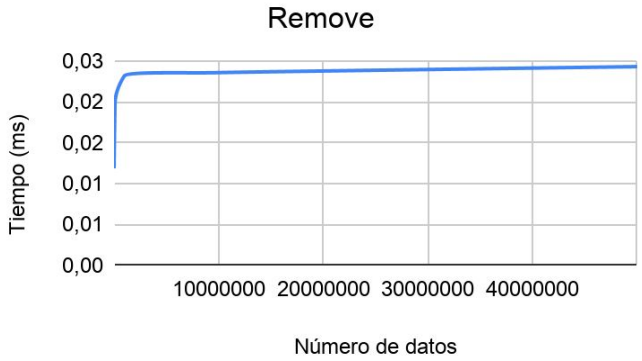
Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	0,23	0,23	0,24	0,26	0,26



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.

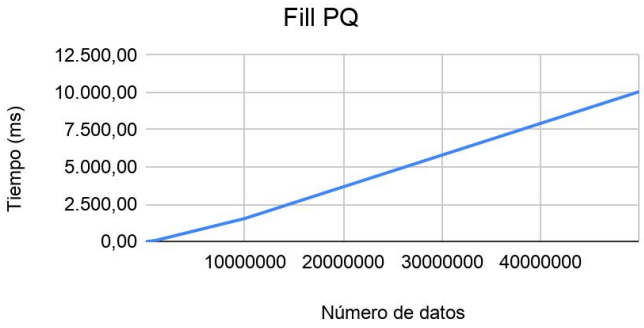
El resumen de la complejidad operacional en la notación Big O de cada una de las funcionalidades se encuentra a continuación:

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)

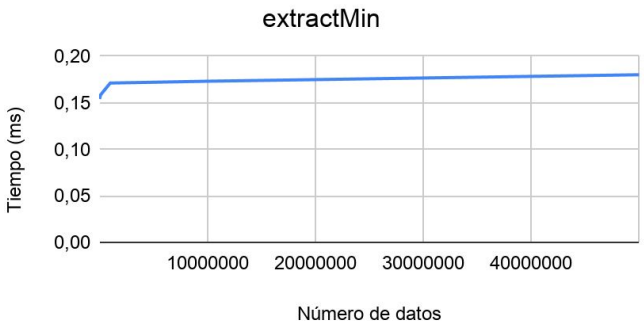
Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	3,26	12,53	95,57	1.557,53	10.044,07



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).

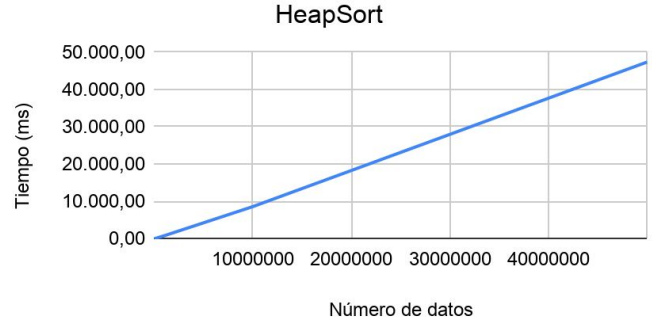
- Funcionalidad: Eliminación un elemento de la cola (Remove)

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.*

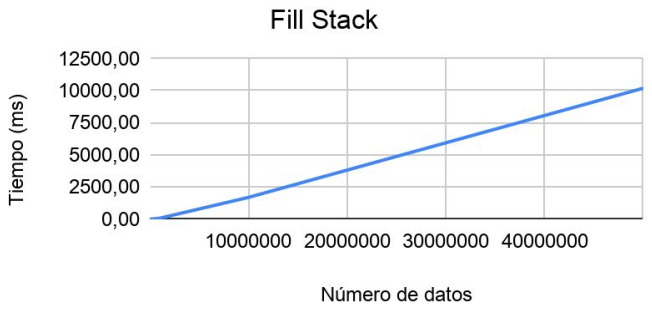
El resumen de la complejidad operacional en la notación Big O de cada una de las funcionalidades se encuentra a continuación:

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)

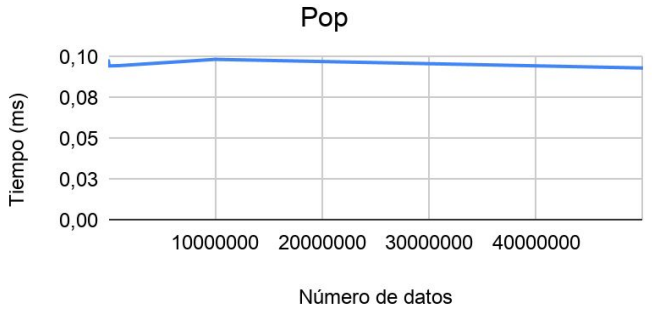
Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	2,68	12,85	81,28	1702,59	10189,42



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.

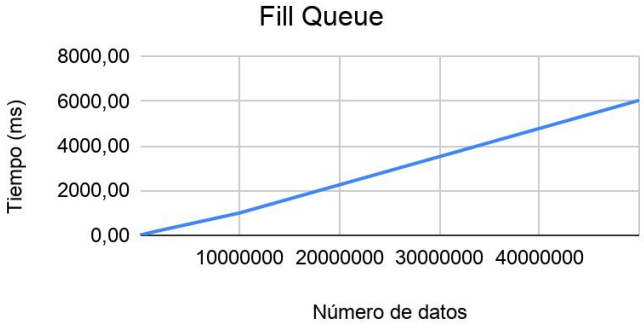
El resumen de la complejidad operacional en la notación Big O de cada una de las funcionalidades se encuentra a continuación:

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

5) Cola de preguntas

● Registro de las preguntas (Fill Queue)

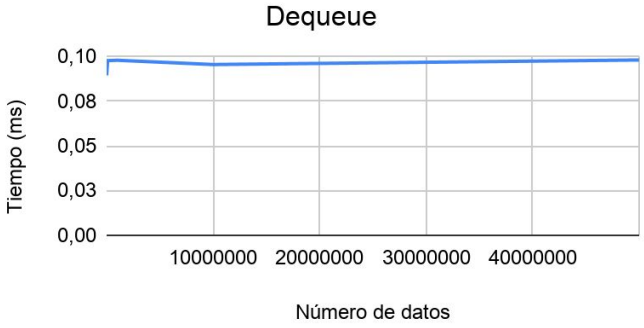
Número de datos	10000	100000	1000000	10000000	50000000
Tiempo (ms)	2,45	11,76	117,87	999,84	6034,72



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

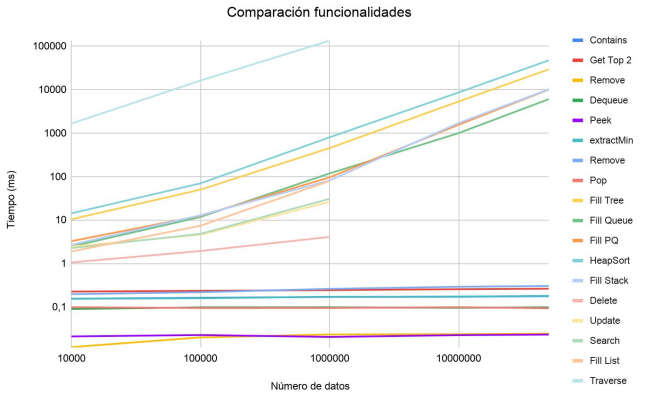
Se observa que el tiempo que toma visualizar el primer elemento de la cola es de complejidad O(1), debido que la implementación de la cola se hace sobre una lista enlazada y solo se debe retornar la información de la cabeza de la lista. En comparación con la operación de dequeue, al no tener que actualizar 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:

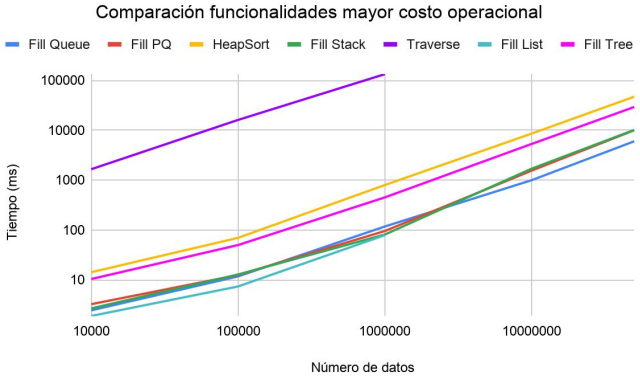
Funcionalidad	Complejidad
Fill Queue	O(n)
Dequeue	O(1)
Peek	O(1)

I. Análisis Comparativo

Para el análisis comparativo, inicialmente se realizó una gráfica en escala logarítmica 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, la operación traverse de la lista de usuarios y el HeapSort de la cola de prioridad. A continuación se procede a realizar una gráfica de tan solo estas funcionalidades en escala logarítmica, la cual se observa a continuación:



De la gráfica se puede concluir, que la funcionalidad con mayor costo operacional es generar el reporte de todos los usuarios (traverse), ya que se debe acceder a cada nodo y además imprimir la información contenida en el mismo. 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 List	Lista encadenada	O(n)
Fill Tree	Árbol binario	O(nlogn)
Fill Stack	Pila	O(n)
Traverse	Lista encadenada	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 traverse 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 traverse es lo suficientemente grande para que los tiempos crezcan más rápido que en las otras funcionalidades.

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 cada una de estas.

Durante el desarrollo del segundo prototipo se evidenció la necesidad de hacer un uso eficiente de los recursos y las estructuras de datos para de esta forma optimizar los tiempos que toman ejecutar las 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].