Gestor de Cursos y Rutas Académicas

Danny Alejandro Bañol Osorio, Julián Andrés Bonilla Mortigo, José Luis Rativa Medina.

No. de equipo de trabajo: {*G*}

I. INTRODUCCIÓN

Según el direccionamiento de la asignatura Estructura de Datos, se plantea desde un proyecto grupal dar solución a un problema en el que se establezcan y se expongan funcionalidades en torno a las estructuras de datos con las que nos vamos familiarizando a lo largo del desarrollo de la asignatura; para ello decidimos desde nuestra perspectiva de estudiantes, plantear una solución a una de las problemáticas actuales sobre la gestión de asignaturas o planes académicos a lo largo de la carrera. El presente documento es, entonces, el desarrollo de la exploración de rutas a posibles soluciones a dicho planteamiento.

II. DESCRIPCIÓN DEL PROBLEMA A RESOLVER

Los estudiantes deben y necesitan planificar correctamente su plan de estudios, según sus proyectos personales y profesionales, sin las preocupaciones usuales de saber cuáles prerrequisitos tiene una asignatura. Para esto, el estudiante debe tener en cuenta si está cursando asignaturas por fuera de la línea de avance de un programa, si ha cambiado el estado de una asignatura respecto a un plan académico o si la plataforma del *Sistema de Información Académico* (SIA) está funcionando correctamente para acceder a esta información.

Esta necesidad inicial llevó a comprender que existe un problema de fondo asociado; que, sólo el 28% de los estudiantes se gradúan a tiempo de la Universidad [1] y que una de las razones por las cuales los estudiantes no pueden finalizar su plan académico a tiempo, radica en que no se tiene claridad en las asignaturas que deben seguir para terminar dicho programa [2].

La solución que se plantea al problema implicaría por ejemplo el desarrollo de una Progressive *Web App* (PWA) (más adelante se definirá qué es una PWA), como herramienta que permita a los estudiantes seleccionar los cursos que integrarán la ruta a seguir a lo largo de su plan de estudios. De la misma manera, facilitará la comprensión de la carga académica que comprende cada curso, y minimizará el impacto en la toma de decisiones [3] ya que el estudiante tendrá un plan a seguir en su plan de estudios y no se verá en la situación de tomar decisiones a última hora. Crear una ruta académica tiene que ser parte de una buena experiencia en la institución, no otro problema.

III. USUARIOS DEL PRODUCTO DE SOFTWARE

En esta sección se hace mención de las características y clasificación de los usuarios (perfiles/roles) que utilizarán el producto.

- Estudiantes: Los principales usuarios del producto serán los estudiantes que quieran planificar su ruta académica de acuerdo a sus elecciones.

IV. REQUERIMIENTOS FUNCIONALES DEL SOFTWARE

A continuación, se presentan las actividades que el producto debe realizar y ejecutar.

Entre los posibles requerimientos funcionales del sistema, se incluyen:

- Los usuarios deberán escoger, entre el catálogo de asignaturas, las que deseen cursar con el fin de generar sugerencias.
- Consultar las asignaturas disponibles.
- Los usuarios son los que pueden ingresar datos al sistema. Dichos datos pueden ser nombre de asignaturas, sus prerrequisitos, alguna ruta donde se guarden un archivo previamente generado por el programa.
- Los reportes se presentan inicialmente en consola al usuario. En próximos avances implementaremos blazor para crear un ambiente más interactivo.
- El sistema procesa la información de acuerdo a las elecciones del usuario.
- Menú:
  El usuario puede elegir entre algunas opciones que puede realizar el programa. En está pantalla el sistema muestra las opciones y procesa la opción escogida.
- Organizar asignaturas:

  En está pantalla el sistema pide el número de asignaturas a organizar por prerrequisitos, por lo tanto, pide para cada asignatura el nombre y seguidamente los prerrequisitos. Luego de esto, el sistema analiza y calcula un semestre

apropiado para cursar cada materia según los requisitos que está tenga.
- Selecciona asignaturas para generar rutas:

  En está pantalla el sistema pregunta el nombre de la asignatura, busca en las asignaturas del sistema y sugiere una carrera a cursar.
- Buscar asignatura:
  En está pantalla el sistema pregunta por el nombre de la asignatura, la busca en las asignaturas del sistema y le muestra información relevante al usuario acerca de estas.

- Cargar un archivo txt para ver mis asignaturas:

  En está pantalla el sistema requerirá una ruta donde se guarde un archivo txt previamente generado por el programa en la opción 1, lo leerá y lo mostrará al usuario para su visualización.

- Consultar asignaturas disponibles:
  En está pantalla el sistema simplemente muestra todas las asignaturas disponibles.

A continuación, se presentan las funcionalidades iniciales para el programa.

- ***Organizar ruta académica.***

- *Descripción:* El usuario ingresa el nombre de las asignaturas que va a cursar junto con los prerrequisitos que estas piden para ser cursadas y de acuerdo a esto el programa sugiere el semestre en el que debería ver la asignatura.

- *Acciones iniciadoras y comportamiento esperado*:

  1. Usuario selecciona opción Organizar ruta académica, seguido a esto el sistema le pregunta al usuario el número de asignaturas a organizar.
  2. El usuario ingresa el nombre de n asignaturas que desee cursar junto con el nombre de sus prerrequisitos.
  3. De acuerdo a los prerrequisitos de cada asignatura el sistema le asigna un semestre para cursarla.
  4. Una vez las asignaturas tengan un semestre asignado, el programa le muestra al usuario las asignaturas y el semestre que se le fue asignado para cursarla.
  5. El sistema genera un archivo de texto plano donde queda guardada la información para el usuario.

- ***Cargar archivo para ver asignaturas.***

- *Descripción:* El usuario podrá cargar un archivo txt para que pueda visualizar las asignaturas que va a cursar.

- *Acciones iniciadoras y comportamiento esperado*:

  1. Usuario selecciona opción *Cargar un archivo txt para ver mis asignaturas.*
  2. El sistema pregunta al usuario por la dirección del directorio donde se encuentra el archivo de texto plano. El usuario debe poner la dirección de la carpeta donde está el archivo.
  3. El sistema pregunta el nombre del archivo y el usuario debe poner el nombre del archivo sin la extensión .txt.
  4. Si la ruta es incorrecta, el sistema le informará al usuario y le pedirá que intente de nuevo.
  5. Si la ruta es correcta, el sistema lee el archivo y organiza las asignaturas por semestre y permite su visualización.

- ***Sugerir ruta académica.***

- *Descripción:* De acuerdo a la o las asignaturas que el usuario desee cursar, el software le sugerirá un programa académico que contiene está asignatura, así cómo la ruta a seguir para culminar el programa académico.

- *Acciones iniciadoras y comportamiento esperado*:
  6. El usuario ingresa el nombre de una asignatura que desee cursar.
  7. El sistema busca la asignatura.
  8. El sistema busca los programas asociados.
  9. De acuerdo al o los programas académicos a los que pertenece la asignatura, el sistema sugerirá una o varias rutas a seguir.

- ***Obtener información de asignaturas.***

- *Descripción:* El usuario ingresa el nombre de una asignatura con la intención de obtener información

adicional, cómo el número de créditos, componente, entre otros.

o Descripción breve de la funcionalidad.

- *Acciones iniciadoras y comportamiento esperado*:

Usuario selecciona opción *obtener información de asignaturas.*
El sistema pregunta al usuario por el nombre de la asignatura a buscar.
El sistema busca la asignatura y muestra la información en pantalla al usuario.
Si no se encuentra la asignatura, el sistema comunicará al usuario que la asignatura no fue encontrada.

6. V. AVANCE EN LA IMPLEMENTACIÓN DE LA INTERFAZ DE USUARIO

Para la página inicial de la aplicación se pensó que tenga una barra de navegación con los botones de *inicio*, que llevará al usuario a la página inicial, *Planifica tu estudio,* que llevará al usuario al apartado donde elige asignaturas y planifica la ruta académica, *Tutorial,* que llevará al usuario a un video demostrativo de cómo usar el producto y por último un botón llamado *¿Qué somos?* que llevará al usuario a una sección donde se explica que es el producto y para qué sirve. También se mostrará el nombre del producto (que puede cambiar, no está decidido del todo) y algunas funciones cómo se muestra en la imagen [6]. En la página planifica estudio se dispondrán de links para dirigirse a las demás funcionalidades. En la imagen 8 se puede ver el boceto para la página de organización de tus asignaturas, en la que habrá un campo para ingresar el nombre de la asignatura, junto con sus créditos, y prerrequisitos. Sumado a esto un botón *Ingresar* para proporcionarle los datos al sistema y un botón *Organizar* para cumplir con la funcionalidad *Organiza tus asignaturas*

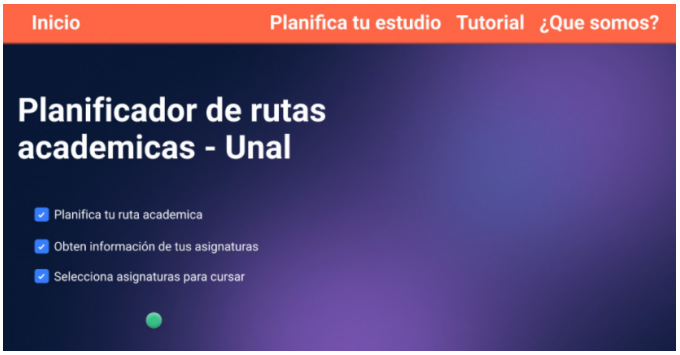

Imagen 1. Boceto página inicial.

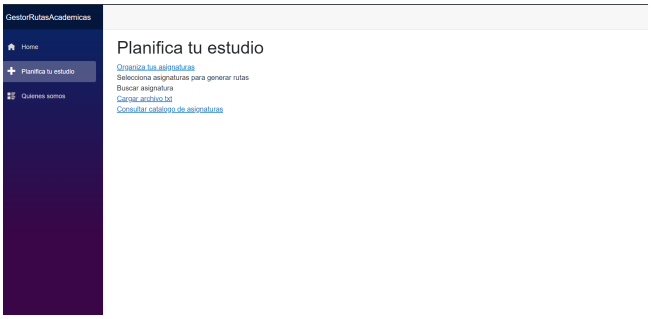

Imagen 2. Boceto página planifica tu estudio

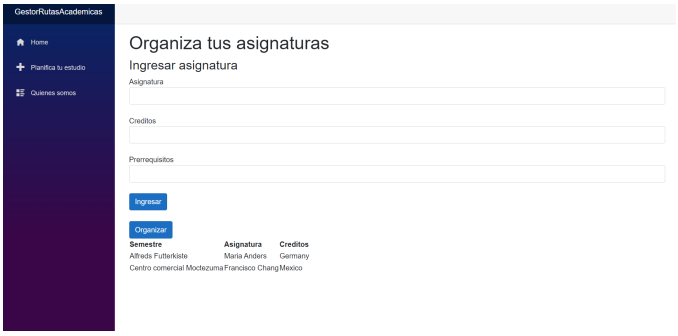

Imagen 3. Boceto página Organiza tus asignaturas.

VI. ENTORNOS DE DESARROLLO Y DE OPERACIÓN

El entorno de desarrollo usado es Blazor, el cual es un framework desarrollado por Microsoft que permite desarrollar PWA. Una PWA es cómo un híbrido entre una aplicación web y una aplicación de escritorio, es decir, una PWA es una página web cómo otras, pero utiliza service workers que permiten que sea ejecutada como una aplicación normal más que como aplicación web. Una vez que el usuario corre la aplicación en su navegador por primera vez mientras está Online el navegador descarga los recursos necesarios para poder operar Offline [4].

Blazor funciona principalmente bajo el lenguaje de programación C#, también desarrollado por Microsoft. Del mismo modo, utiliza Css para darle estilo a las páginas web y Html para darle la estructura a las páginas. Blazor es una alternativa para desarrollar aplicaciones web ya que usa C# en lugar de JavaScript (Js). A pesar de lo anterior, es posible llamar librerías de Js que se consideren útiles para el desarrollo del proyecto [5].

Debido a que lo que se quiere desarrollar es una PWA, ésta operará en el navegador del usuario y por tanto el sistema operativo en el que se ejecute puede ser cualquiera, siempre y cuando cuente con un navegador y del mismo modo, puede ser ejecutado por cualquier configuración de Hardware.

VII. DESCRIPCIÓN DEL PROTOTIPO DE SOFTWARE

Esté prototipo aplica, a diferencia del anterior, no sólo estructuras de datos lineales, sino que también emplea los tipos de datos abstractos de Árbol y la Tabla hash ocasionando que las funcionalidades del programa se vean reducidas en tiempos de ejecución mejorando la experiencia de usuario. Por último, no se logró cumplir con la aplicación del programa a una PWA, es decir no se logró que la aplicación tuviera una interfaz de usuario gráfica debido a que no alcanzó el tiempo durante el semestre.

VIII. IMPLEMENTACIÓN Y APLICACIÓN DE LAS ESTRUCTURAS DE DATOS

En este prototipo se aplicaron las siguientes estructuras de datos:

- **Lista encadenada:** La lista encadenada se usó para guardar la lista de los prerrequisitos de cada asignatura, además de que se usó una lista encadenada para la implementación de la tabla hash con direccionamiento cerrado.
- **Pila:** La pila se aplicó en el organizamiento de las líneas de los archivos txt generados. En estos archivos están escritas algunas asignaturas con el semestre sugerido y para que el usuario lo pueda visualizar de una mejor manera, se usaron las pilas para organizarlas de acuerdo al semestre.
- **Árbol AVL:** El árbol AVL se usó para almacenar las asignaturas que se reciben por teclado por parte del usuario y luego de esto asignarles un semestre para cursar las asignaturas de acuerdo a sus prerrequisitos.
- **Tabla hash:** La tabla hash se usó para almacenar las asignaturas disponibles y que cuando el usuario las vaya a consultar el tiempo de respuesta sea bastante rápido.

IX. PRUEBAS DEL PROTOTIPO DE SOFTWARE

1. Árbol: Se realizaron dos implementaciones para el árbol, la primera es un árbol AVL y la segunda es un árbol BST convencional, es decir, sin operaciones de rotaciones para su balanceo. Se realizaron 10 mediciones del tiempo de ejecución y se les realizó el promedio a estos tiempos, para distintas cantidades de datos y de esta forma poder comparar estas implementaciones de la estructura de datos en cuestión. Si bien el objetivo principal es comparar el árbol AVL y el árbol BST desbalanceado, también se realizaron mediciones para la lista enlazada en las mismas funcionalidades ya que en el prototipo anterior empleamos la lista enlazada para guardar datos y se deseaba comparar con lo que ya se tenía y decidir usar el árbol AVL, BST o la lista enlazada en el proyecto.

- **Inserción de datos:**

Para está funcionalidad se obtuvieron los siguientes resultados:

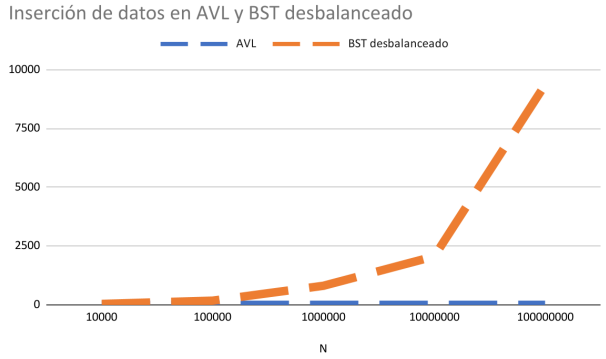

Gráfico 1. Comparación de los tiempos de ejecución en la inserción de datos en AVL, BST desbalanceado.

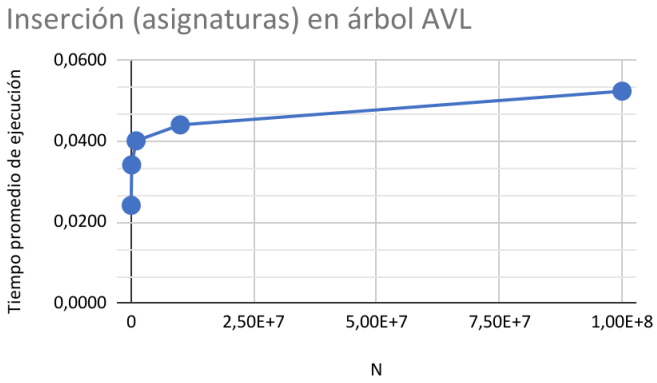

Gráfico 2.Tiempos de ejecución en la inserción de datos en AVL.

En el gráfico 1 se evidencia que el crecimiento en tiempo de ejecución para la inserción de datos en el árbol BST desbalanceado es mucho mayor que en el árbol AVL tanto que parece que el árbol AVL creciera en tiempo constante, lo cual no es de está manera, porque analizando el gráfico 2 se evidencia que el crecimiento en tiempo de ejecución del árbol AVL es logarítmico, lo cual sigue siendo mucho mejor que el crecimiento lineal del árbol BST desbalanceado que en el gráfico 1 parece

cuadrático pero es debido que el gráfico 1 está en una escala diferente.

Tabla 1. Tiempos de ejecución para la inserción de datos en AVL, BST, lista enlazada.

Nombre de la funcionalidad	Tipo(s) de estructura de datos	Cantidad de datos probados	Análisis realizado (Notación Big O)	Tiempos de ejecución (ms)
Inserción de datos (Asignaturas)	Árbol AVL	10000	O(logn)	0,0242
		100000	O(logn)	0,0342
		1000000	O(logn)	0,0401
		10000000	O(logn)	0,0441
		100000000	O(logn)	0,0524
	Árbol BST	10000	O(n)	39,3727
		100000	O(n)	178,7520
		1000000	O(n)	811,5341
		10000000	O(n)	2061,2967
		100000000	O(n)	9358,2871
	Lista enlazada	10000	O(1)	0,0050
		100000	O(1)	0,0066
		1000000	O(1)	0,0055
		10000000	O(1)	0,0063
		100000000	O(1)	0,0073

Como se puede apreciar en la tabla al momento de insertar datos el árbol AVL tiene un mejor desempeño que el árbol BST desbalanceado ya que en el AVL gracias a sus rotaciones al momento de insertar datos, garantiza que no hayan demasiados niveles y por tanto se garantiza que la inserción de datos sea O(logn) mientras que el árbol BST desbalanceado, en el peor caso, si el valor que se inserta es siempre mayor que el valor máximo del árbol, siempre insertará en el hijo derecho del valor máximo del árbol y si es siempre menor que el valor mínimo del árbol, insertará siempre en el hijo izquierdo. En cualquiera de estos dos casos, el árbol tendrá n niveles, donde n es el número de datos y el árbol se comportaría como una lista enlazada sin referencia a la cola teniendo que recorrer todos los nodos del árbol para insertar un dato. Por tanto, el árbol AVL mejora significativamente al árbol BST desbalanceado.

Del mismo modo, la lista enlazada muestra un mejor desempeño en la inserción de datos ya que como tiene referencia a la cola, lo único que hace es agregar un nodo siguiente a la cola y actualiza la referencia de la cola a este nuevo nodo por lo que la operación se hace en tiempo constante O (1). Tenemos que, para la inserción de datos, el árbol AVL es mejor que el árbol BST desbalanceado y la lista enlazada con referencia a la cola es mejor que el árbol AVL.

- **Búsqueda de datos:**

Para está funcionalidad se obtuvieron los siguientes resultados:

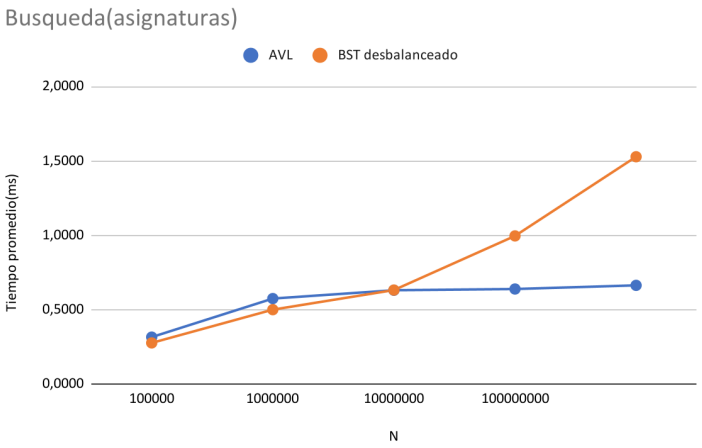

Gráfico 3. Comparación de los tiempos de ejecución en la búsqueda de datos en AVL y BST desbalanceado.

El gráfico 3 muestra que para la búsqueda de datos el tiempo de ejecución del BST desbalanceado crece de manera lineal y para pequeñas cantidades de datos, es un poco mejor, no mucho, en comparación con el árbol AVL el cual crece de manera logarítmica y para grandes cantidades de datos el árbol AVL es mucho más rápido en tiempos de ejecución que el árbol BST desbalanceado, motivo por el cual se justifica que el árbol AVL es bastante mejor que el árbol BST desbalanceado.

Tabla 2. Tiempos de ejecución para la búsqueda de datos en AVL, BST, lista enlazada.

Nombre de la funcionalidad	Tipo(s) de estructura de datos	Cantidad de datos probados	Análisis realizado (Notación Big O)	Tiempos de ejecución (ms)
Búsqueda de datos (asignaturas)	Árbol AVL	10000	O(logn)	0,3167
		100000	O(logn)	0,5758
		1000000	O(logn)	0,6317
		10000000	O(logn)	0,6400
		100000000	O(logn)	0,6646
	Árbol BST	10000	O(n)	0,2777
		100000	O(n)	0,5012
		1000000	O(n)	0,6329
		10000000	O(n)	0,9968
		100000000	O(n)	1,5289
	Lista enlazada	10000	O(n)	1,189
		100000	O(n)	12,784
		1000000	O(n)	96,919
		10000000	O(n)	1056,623
		100000000	O(n)	8689,08

La tabla 2 muestra que la búsqueda de datos es mejor en el árbol AVL que en el árbol BST desbalanceado y también es mejor que en la lista enlazada. Esto se debe a que el balanceo del AVL garantiza que tenga la búsqueda sea O(logn) mientras que en el árbol desbalanceado en el peor caso tiene n niveles y al igual que la lista enlazada tiene que recorrer todos los nodos hasta encontrar el dato deseado. Por tanto, en cuanto a búsqueda, el mejor es el Árbol AVL.

- **Eliminación de datos:**
  Para está funcionalidad se obtuvieron los siguientes resultados.

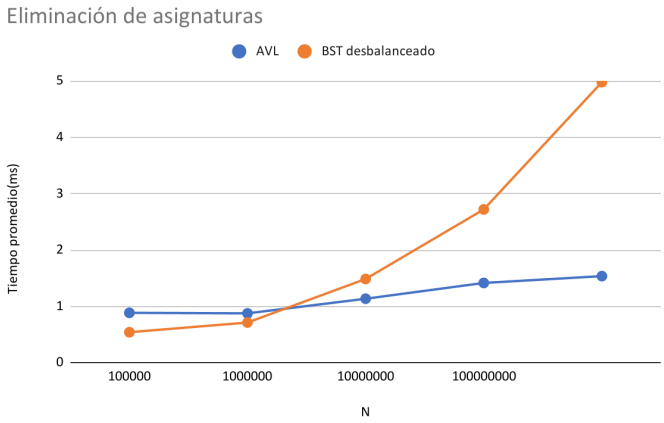

Gráfico 4. Comparación de los tiempos de ejecución en eliminación de datos en AVL y BST desbalanceado.

El gráfico 4 muestra que para la eliminación de datos el tiempo de ejecución del BST desbalanceado crece de manera lineal(en el gráfico parece cuadrático pero es debido a la escala en la que está) y para pequeñas cantidades de datos, es un poco mejor, no mucho, en comparación con el árbol AVL el cual crece de manera logarítmica (en el gráfico parece constante pero es debido a la escala en la que está) y para grandes cantidades de datos el árbol AVL es mucho más rápido en tiempos de ejecución que el árbol BST desbalanceado, motivo por el cual se justifica que el árbol AVL es bastante mejor que el árbol BST desbalanceado.

Tabla 3. Tiempos de ejecución para la eliminación de datos en AVL, BST, lista enlazada.

Nombre de la funcionalidad	Tipo(s) de estructura de datos	Cantidad de datos probados	Análisis realizado (Notación Big O)	Tiempos de ejecución (ms)
Eliminación de datos (asignaturas)	Árbol AVL	10000	O(logn)	0,8866
		100000	O(logn)	0,8771
		1000000	O(logn)	1,1366
		10000000	O(logn)	1,4167
		100000000	O(logn)	1,5377
	Árbol BST	10000	O(n)	0,5421
		100000	O(n)	0,7140
		1000000	O(n)	1,4872
		10000000	O(n)	2,7216
		100000000	O(n)	4,9806
	Lista enlazada	10000	O(n)	2,7263
		100000	O(n)	29,3203
		1000000	O(n)	222,2903
		10000000	O(n)	2.423,4480
		100000000	O(n)	24.339,1731

En cuanto a eliminación de un dato en específico, ocurre algo muy similar a la búsqueda, pues el AVL muestra ser mejor que el árbol BST desbalanceado y mejor que la lista enlazada ya que para eliminar un dato, primero se tiene que buscar el nodo en el que está y después removerlo. Por tanto, en cuanto a eliminación de datos el árbol AVL es la mejor opción.

De acuerdo a lo anterior, el árbol AVL es bastante mejor que el árbol BST desbalanceado pues todas sus operaciones ocurren en tiempo O(logn) mientras que en el árbol BST desbalanceado, en el peor caso, ejecuta sus operaciones en tiempo lineal O(n).

Por último, para el proyecto se había decidido cambiar la lista enlazada por el árbol AVL para guardar las asignaturas ya que consideramos que es mejor opción para el almacenamiento de datos pues la lista enlazada solo supera al árbol AVL en inserción de datos mientras que el árbol AVL la supera en búsqueda y eliminación de datos pero al analizar la tabla hash decidimos cambiar el arbol AVL por la tabla hash para almacenar las asignaturas disponibles (esto se explica en el análisis de tiempos de ejecución de las funcionalidades de la tabla hash).

2. Cola de prioridad: Se realizaron dos implementaciones para la cola de prioridad, la primera es un max heap (montículo) y la segunda es una cola de prioridad basada en lista enlazada. Se realizaron 10 mediciones del tiempo de ejecución y se les realizó el promedio a estos tiempos, para distintas cantidades de datos y de esta manera poder comparar estas implementaciones de la estructura de datos en cuestión.

- **Desencolamiento:**
  Para el montículo se obtuvieron los siguientes resultados:

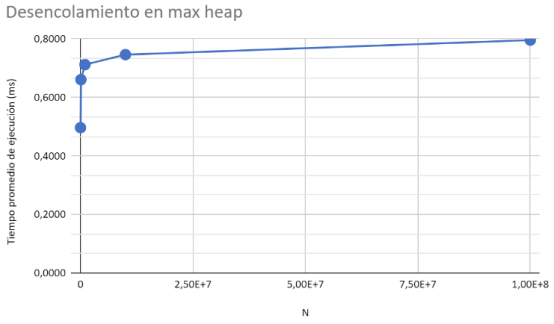

Gráfico 5 . Tiempos de ejecución método “desencolar” en max heap.

La gráfica 5 evidencia que los tiempos de ejecución para el desencolamiento de datos en un montículo crece de manera logarítmica.

Para la cola de prioridad basada en lista enlazada se obtuvieron los siguientes resultados:

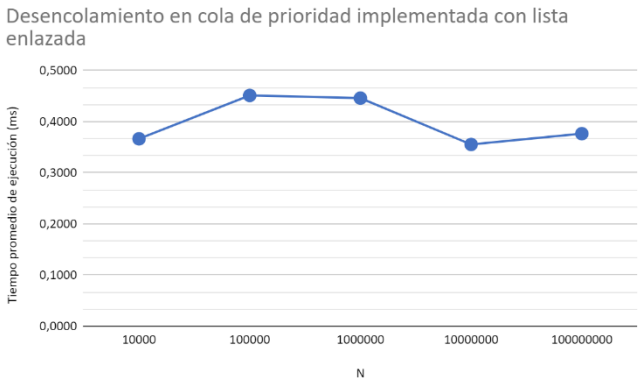

Gráfico 6. Tiempos de ejecución método "desencolar" en cola de prioridad implementadas con Listas enlazadas.

La gráfica 6 evidencia que los tiempos de ejecución para el desencolamiento de datos en una cola de prioridad basada en lista enlazada ocurre en tiempo constante.

Tabla 4. Tiempos de ejecución para el desencolamiento de datos en max heap, cola de prioridad basada en lista enlazada.

Nombre de la funcionalidad	Tipo(s) de estructura de datos	Cantidad de datos probados	Análisis realizado (Notación Big O)	Tiempos de ejecución (ms)
Desencolamiento	Max Heap	10000	O(logn)	0,4965
		100000	O(logn)	0,6608
		1000000	O(logn)	0,7119
		10000000	O(logn)	0,7461
		100000000	O(logn)	0,7955
	Cola de prioridad basada en Lista enlazada	10000	O(1)	0,3664
		100000	O(1)	0,4511
		1000000	O(1)	0,4459
		10000000	O(1)	0,3553
		100000000	O(1)	0,3764

La tabla 4 muestra que el desencolamiento en la cola de prioridad basada en la lista enlazada O(1) tiene mejor desempeño que en el montículo O(logn). Esto se debe a que para desencolar un valor de la cola basada en lista enlazada lo único que se hace es remover el nodo que está al inicio de la cola ocurriendo en tiempo constante mientras que en el montículo se tiene que intercambiar el valor que está en la última posición del arreglo con el que está en la posición 1 y ejecutar la operación de *percolateDown* la cual ocurre en O(logn). Por tanto, para desencolar es mejor la cola de prioridad basada en lista enlazada.

- **Encolamiento al inicio de la cola (máxima prioridad):**

  Para el montículo se obtuvieron los siguientes resultados:

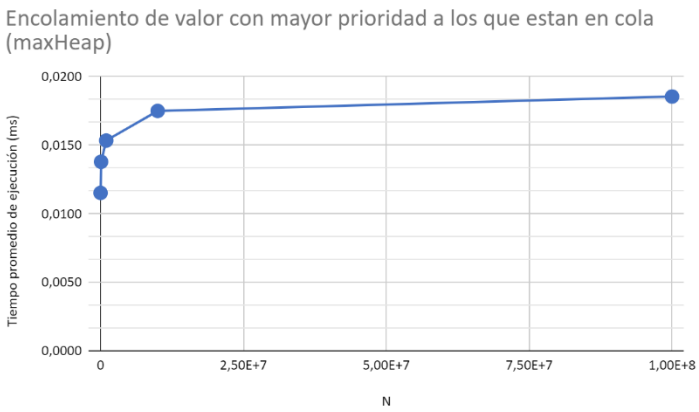

Gráfico 7. Tiempos de ejecución método "encolar" de valores con prioridad máxima en max heap.

La gráfica 7 evidencia que los tiempos de ejecución para el encolamiento de datos con prioridad máxima (al inicio de la cola) en un montículo crece de manera logarítmica.

Para la cola de prioridad basada en lista enlazada se obtuvieron los siguientes resultados:

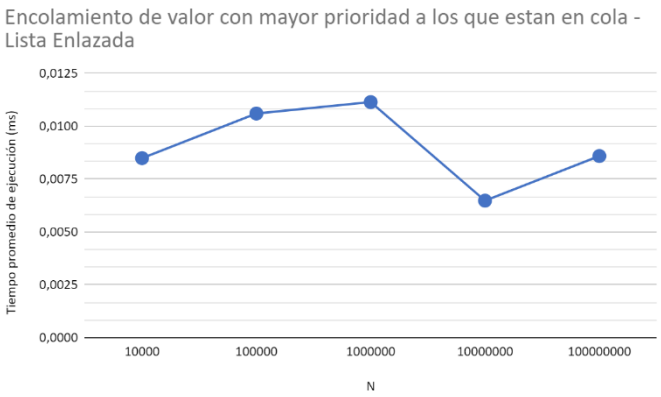

Gráfico 8. Tiempos de ejecución método "encolar" de valores con prioridad máxima en cola de prioridad basada en lista enlazada.

La gráfica 8 evidencia que los tiempos de ejecución para el desencolamiento de datos con prioridad máxima (al inicio de la cola) en una cola de prioridad basada en lista enlazada ocurre en tiempo constante, no depende del número de datos.

Tabla 5. Tiempos de ejecución para el encolamiento de datos con prioridad máxima en max heap y cola de prioridad basada en lista enlazada.

Nombre de la funcionalidad	Nombre de la funcionalidad	Nombre de la funcionalidad	Análisis realizado (Notación Big O)	Tiempos de ejecución (ms)
Encolamiento de valor con mayor prioridad a los que estan en la cola (inicio de la cola)	Max Heap	10000	O(logn)	0,012
		100000	O(logn)	0,014
		1000000	O(logn)	0,015
		10000000	O(logn)	0,018
		100000000	O(logn)	0,02
	Cola de prioridad basada en Lista enlazada	10000	O(1)	0,0085
		100000	O(1)	0,0106
		1000000	O(1)	0,0111
		10000000	O(1)	0,0065
		100000000	O(1)	0,0086

La tabla 5 muestra que el encolamiento de un valor con prioridad máxima en la cola de prioridad basada en la lista enlazada O(1) tiene mejor desempeño que en el montículo O(logn). Esto se debe a que para encolar un valor con prioridad máxima (inicio de la cola) en la cola basada en lista enlazada lo único que se hace es agregar un nodo al inicio de la cola ocurriendo en tiempo constante mientras que en el montículo se tiene que agregar el valor que en la última posición del arreglo o en el "hueco" siguiente y ejecutar la operación de *percolateUp* la cual ocurre en O(logn). Por tanto, para encolar un valor de máxima prioridad es mejor la cola de prioridad basada en lista enlazada, aunque como las operaciones de asignación en un arreglo son bastante veloces, el tiempo de ejecución se ve reducido.

- **Encolamiento al final de la cola (mínima prioridad):**

  Para el montículo se obtuvieron los siguientes resultados:

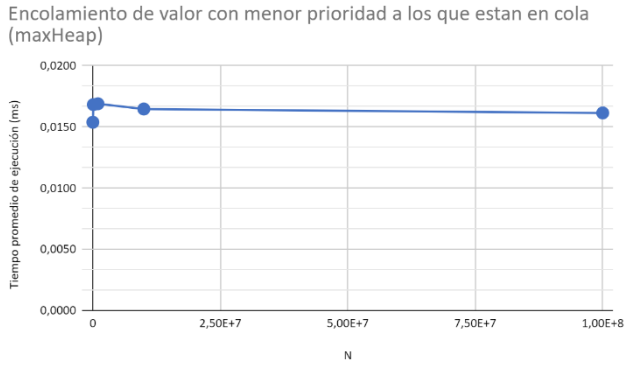

Gráfico 9 . Tiempos de ejecución método "encolar" con menor prioridad a los que están en cola (max heap).

La gráfica 9 evidencia que los tiempos de ejecución para el encolamiento de datos con prioridad mínima (al final de la cola) en un montículo ocurre en tiempo constante, no depende del número de datos.

Para la cola de prioridad basada en lista enlazada se obtuvieron los siguientes resultados:

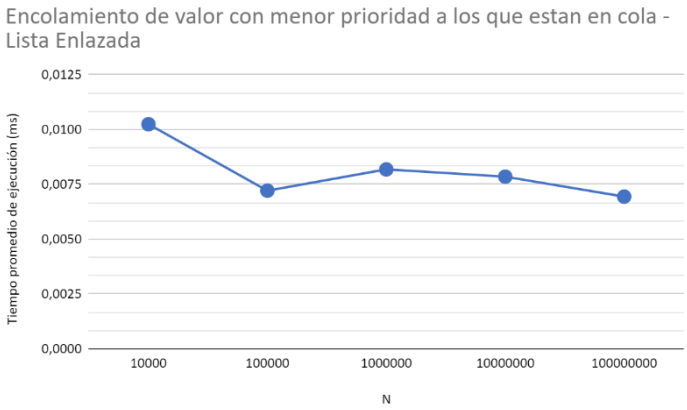

Gráfico 10. Tiempos de ejecución método "encolar" con menor prioridad a los que están en cola de prioridad implementadas con Listas enlazadas.

La gráfica 10 evidencia que los tiempos de ejecución para el encolamiento de datos con prioridad mínima (al final de la cola) en una cola basada en lista enlazada ocurre en tiempo constante, no depende del número de datos.

Tabla 6. Tiempos de ejecución para el encolamiento de datos con prioridad mínima en max heap, cola de prioridad basada en lista enlazada.

Nombre de la funcionalidad	Nombre de la funcionalidad	Nombre de la funcionalidad	Análisis realizado (Notación Big O)	Tiempos de ejecución (ms)
Encolamiento de valor con menor prioridad a los que estan en la cola (final de la cola)	Max Heap	10000	O(1)	0,0154
		100000	O(1)	0,0168
		1000000	O(1)	0,0169
		10000000	O(1)	0,0165
		100000000	O(1)	0,0161
	Cola de prioridad basada en Lista enlazada	10000	O(1)	0,0102
		100000	O(1)	0,0072
		1000000	O(1)	0,0082
		10000000	O(1)	0,0079
		100000000	O(1)	0,0069

La tabla 6 muestra que el encolamiento de un valor con prioridad mínima en la cola de prioridad basada en la lista enlazada O(1) tiene igual desempeño que en el montículo O(l). Esto se debe a que para encolar un valor con prioridad mínima (final de la cola) en la cola basada en lista enlazada lo único que se hace es agregar un nodo al final de la cola ocurriendo en tiempo constante mientras que en el montículo se tiene que agregar el valor que en la última posición del arreglo o en el "hueco" siguiente y ejecutar la operación de *percolateUp* la cual ocurre en O(logn) pero como se está insertando un valor con prioridad mínima, la operación percolateUp no se ejecuta, pues no se cumple la condición de que el hijo sea mayor al padre (max Heap) y no debe hacer intercambios en las posiciones del arreglo. Por tanto, para encolar un valor de mínima prioridad el desempeño es igual en ambas implementaciones.

- **Encolamiento promedio:**
  Para el montículo se obtuvieron los siguientes resultados:

Gráfico 11. Tiempos de ejecución método "encolar" con valor de prioridad media con respecto a los que están en cola (max heap).

La gráfica 11 evidencia que los tiempos de ejecución para el encolamiento de datos con prioridad media (posiciones internas de la cola) en un montículo crece de manera logarítmica.

Para la cola de prioridad basada en lista enlazada se obtuvieron los siguientes resultados:

Gráfico 12. Tiempos de ejecución método "encolar" con valor de prioridad medio con respecto a los que están en cola de prioridad implementadas con Listas enlazadas.

La gráfica 12 muestra que los tiempos de ejecución para el encolamiento de datos con prioridad media (posiciones internas de la cola) en una cola basada en lista enlazada crece de manera lineal.

Tabla 7. Tiempos de ejecución para el encolamiento de datos con prioridad promedio en max heap, cola de prioridad basada en lista enlazada.

Nombre de la funcionalidad	Nombre de la funcionalidad	Nombre de la funcionalidad	Análisis realizado (Notación Big O)	Tiempos de ejecución (ms)
Encolamiento promedio (posiciones internas de la cola)	Max Heap	10000	O(logn)	0,0066
		100000	O(logn)	0,0097
		1000000	O(logn)	0,0127
		10000000	O(logn)	0,0148
		100000000	O(logn)	0,0210
	Cola de prioridad basada en Lista enlazada	10000	O(n)	1,7673
		100000	O(n)	60,5152
		1000000	O(n)	169,1087
		10000000	O(n)	655,8323
		100000000	O(n)	3.314,6218

La tabla 7 muestra que el encolamiento de un valor con prioridad media en la cola de prioridad basada en la lista enlazada O(n) tiene peor desempeño que en el montículo O(logn). Esto se debe a que para encolar un valor con prioridad media (final de la cola) en la cola basada en lista enlazada se tiene que recorrer la cola hasta encontrar la posición correcta en la que debe ir el dato, es decir, ocurre en tiempo lineal O(n) mientras que en el montículo se tiene que agregar el valor que en la última posición del arreglo o en el "hueco" siguiente y ejecutar la operación de *percolateUp* la cual ocurre en O(logn). Por tanto, para encolar un valor de prioridad media (posiciones internas de la cola) el desempeño es mejor en el montículo.

De acuerdo a lo anterior, la cola de prioridad basada en lista enlazada es mejor al encolar al inicio y final de la cola (máxima y mínima prioridad) y para el desencolamiento ya que ocurre en tiempo constante y el montículo es mejor para el encolamiento común o de prioridad media. Por tanto, podría decirse que la cola en lista enlazada es mejor, pero se debe tener en cuenta que no siempre se va a encolar un valor de máxima ó mínima prioridad, de hecho, la mayoría de las veces se encola un valor que en cuanto a prioridad este en una posición intermedia por tanto consideramos que el montículo es una mejor opción para implementar la cola de prioridad.

3. Tabla Hash: Se realizaron dos implementaciones para la tabla hash, la primera maneja las colisiones mediante direccionamiento cerrado y la segunda maneja las colisiones mediante direccionamiento cerrado. Se realizaron 10 mediciones del tiempo de ejecución y se les realizó el promedio a estos tiempos, para distintas cantidades de datos y de esta manera poder comparar estas implementaciones de la estructura de datos en cuestión.

- **Inserción de datos:**
  Para está funcionalidad se obtuvieron los siguientes resultados:

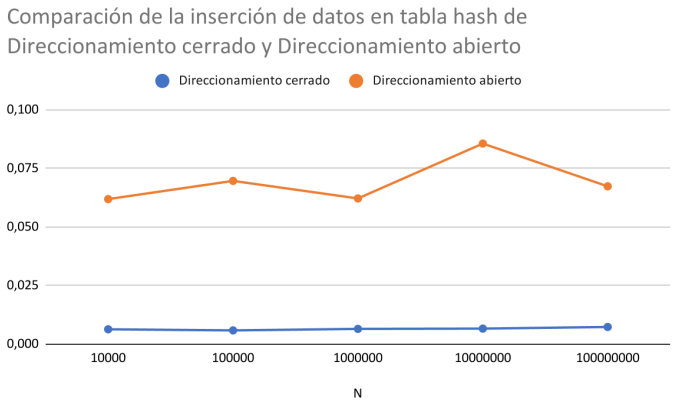

Gráfico 13. Comparación en los tiempos de ejecución en la inserción de datos en tabla hash con direccionamiento abierto y cerrado.

Se logra apreciar en el gráfico 13 que el crecimiento en tiempos de ejecución para la inserción de datos en la tabla Hash es constante (no dependen de la cantidad de datos) tanto para la tabla hash con direccionamiento abierto cómo para la que maneja las colisiones con direccionamiento cerrado, pero de la misma manera se aprecia que el tiempo de ejecución en la tabla Hash con direccionamiento cerrado es menor que la tabla hash con direccionamiento abierto. Consideramos que esto se debe a que para la inserción en direccionamiento cerrado se calcula la posición del arreglo en la que se va a guardar el elemento y en está posición hay una lista enlazada que almacenara el elemento y sólo se agrega el elemento al final de está lista (tiempo constante) mientras que el direccionamiento abierto calcula la posición del arreglo en la que se va aguardar el elemento y si ya está ocupada, se hace el sondeo lineal para encontrar una posición disponible lo cual puede tardar más tiempo.

Tabla 7. Tiempos de ejecución en la inserción de datos en tabla hash con direccionamiento abierto y cerrado.

Nombre de la funcionalidad	Nombre de la funcionalidad	Nombre de la funcionalidad	Análisis realizado (Notación Big O)	Tiempos de ejecución (ms)
Inserción de datos	Tabla Hash - Direccionamiento cerrado	10000	O(1)	0,0064
		100000	O(1)	0,0059
		1000000	O(1)	0,0066
		10000000	O(1)	0,0067
		100000000	O(1)	0,0074
	Tabla Hash - Direccionamiento abierto	10000	O(1)	0,0619
		100000	O(1)	0,0697
		1000000	O(1)	0,0622
		10000000	O(1)	0,0856
		100000000	O(1)	0,0674

- **Búsqueda de datos:**
  Para está funcionalidad se obtuvieron los siguientes resultados:

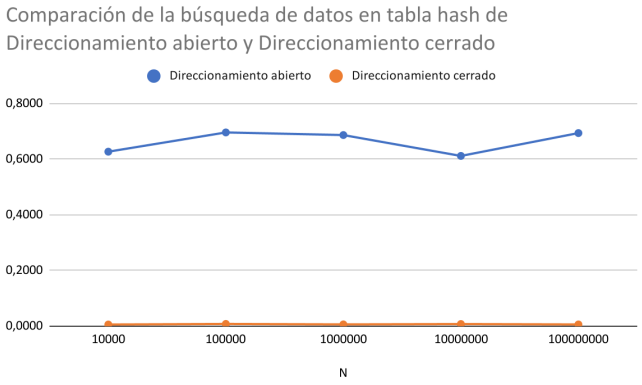

Gráfico 14. Comparación en los tiempos de ejecución en la búsqueda de datos en tabla hash con direccionamiento abierto y cerrado.

Se logra apreciar en el gráfico 14 que el crecimiento en tiempos de ejecución para la búsqueda de datos en la tabla Hash es constante tanto para la tabla hash con direccionamiento abierto cómo para la que maneja las colisiones con direccionamiento cerrado, pero de la misma manera se aprecia que el tiempo de ejecución en la tabla Hash con direccionamiento cerrado es mayor que la tabla hash con direccionamiento abierto. Esto se debe a que para la búsqueda en direccionamiento cerrado se encuentra la posición del arreglo en la que está la lista enlazada que contiene el dato que queremos obtener y luego se debe recorrer esta lista hasta encontrar el dato mientras que el direccionamiento abierto no tiene que recorrer ninguna lista, sólo calcula la posición del arreglo y retorna el elemento.

Tabla 8. Tiempos de ejecución en la búsqueda de datos en tabla hash con direccionamiento abierto y cerrado.

Nombre de la funcionalidad	Nombre de la funcionalidad	Nombre de la funcionalidad	Análisis realizado (Notación Big O)	Tiempos de ejecución (ms)
Búsqueda de datos	Tabla Hash - Direccionamiento cerrado	10000	O(1)	0,6268
		100000	O(1)	0,6959
		1000000	O(1)	0,6865
		10000000	O(1)	0,6116
		100000000	O(1)	0,6936
	Tabla Hash - Direccionamiento abierto	10000	O(1)	0,0060
		100000	O(1)	0,0081
		1000000	O(1)	0,0064
		10000000	O(1)	0,0078
		100000000	O(1)	0,0063

- **Eliminación de datos:**
  Para está funcionalidad se obtuvieron los siguientes resultados:

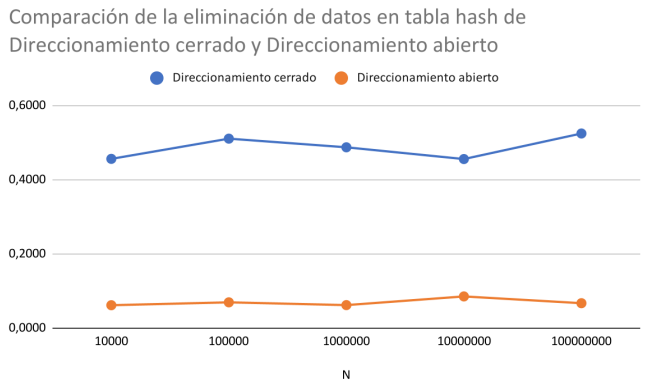

Gráfico 15. Comparación en los tiempos de ejecución en la eliminación de datos en tabla hash con direccionamiento abierto y cerrado.

Se logra apreciar en el gráfico 15 que el crecimiento en tiempos de ejecución para la eliminación de datos en la tabla Hash es constante tanto para la tabla hash con direccionamiento abierto cómo para la que maneja las colisiones con direccionamiento cerrado, pero de la misma manera se aprecia que el tiempo de ejecución en la tabla Hash con direccionamiento cerrado es mayor que la tabla hash con direccionamiento abierto. Esto se debe a que para la eliminación en direccionamiento cerrado se encuentra la posición del arreglo en la que está la lista enlazada que contiene el dato que queremos eliminar y luego se debe iterar sobre la lista hasta esta lista hasta encontrar el dato y eliminarlo mientras que el direccionamiento abierto no tiene que recorrer ninguna lista, sólo calcula la posición del arreglo y elimina el elemento.

Tabla 9. Tiempos de ejecución en la eliminación de datos en tabla hash con direccionamiento abierto y cerrado.

Nombre de la funcionalidad	Tipo(s) de estructura de datos	Cantidad de datos probados	Análisis realizado (Notación Big O)	Tiempos de ejecución (ms)
Eliminación de datos	Tabla Hash - Direccionamiento cerrado	10000	O(1)	0,4561
		100000	O(1)	0,5106
		1000000	O(1)	0,4873
		10000000	O(1)	0,4556
		100000000	O(1)	0,5245
	Tabla Hash - Direccionamiento abierto	10000	O(1)	0,0044
		100000	O(1)	0,0060
		1000000	O(1)	0,0063
		10000000	O(1)	0,0063
		100000000	O(1)	0,0047

De acuerdo a lo anterior y a las tablas 7,8 y 9 a pesar de que ambas implementaciones de la tabla Hash realizan sus operaciones en tiempo constante, el direccionamiento abierto tiene un desempeño un poco mejor que el direccionamiento cerrado para la búsqueda y eliminación de datos y, de la misma manera, el direccionamiento cerrado es un poco mejor que el direccionamiento abierto para insertar datos, por tanto llegamos a la conclusión que el direccionamiento abierto es mejor opción para usar en el proyecto, a demás de que no se necesita de una lista enlazada para su implementación.

Debido a que la tabla Hash presenta tiempos de ejecución constante O(1) para las operaciones de búsqueda, eliminación e inserción, se decidió cambiar el árbol AVL, el cual tiene operaciones O(logn),por una tabla hash para guardar las asignaturas disponibles en el programa.

Todas estas pruebas fueron realizadas en un procesador Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.60 GHz.

X. ACCESO AL REPOSITORIO DE SOFTWARE

En el siguiente enlace se encuentra el repositorio donde se encuentra el ejecutable del programa y su código fuente, además de varios documentos adicionales.
https://github.com/Jrativa/Gestor-de-Cursos-y-Rutas-Academicas

XI. ACCESO AL VIDEO DEMOSTRATIVO DE SOFTWARE

En el siguiente enlace se encuentra un video demostrativo del prototipo:
https://www.loom.com/share/5ef0b74b6351437db75824863b441278

XII. ROLES Y ACTIVIDADES

INTEGRANTE	ROL(ES)	ACTIVIDADES REALIZADAS (Listado)
Julián Bonilla Mortigo	Líder	Consulta a miembros de equipo acerca de sus habilidades
	Experto	Coordinación de actividades
Danny Bañol Osorio	Observador	Pendiente de Actividades a realizar
	Animador	Charlas motivacionales
	Secretario	Organización de tareas
José Luis Rativa Medina	Coordinador	Programación de reuniones

	Investigador	Consultas acerca de blazor y aspnet.

XIII. DIFICULTADES Y LECCIONES APRENDIDAS

- Una de las primeras dificultades que surgieron al principio fue la de definir qué estructuras de datos usar para cada funcionalidad, y con esto aprendimos que es buena práctica hacer varias comparaciones entre diferentes estructuras para definir la "mejor" para cada problema. Sin embargo, esta dificultad fue solucionada gracias a los análisis asintóticos y comparaciones entre las estructuras de datos.

- La tabla Hash con direccionamiento abierto es superior en las funciones de búsqueda y eliminación de datos con respecto al direccionamiento cerrado, así mismo, la función de inserción es superior en el direccionamiento cerrado, a pesar de que en las dos tablas el tiempo de ejecución es constante para las tres funciones.

- El implementar un tipo de dato abstracto que dependa de una estructura de datos adicional cómo en el caso de la tabla hash de direccionamiento cerrado puede ocasionar que algunas de sus operaciones sean un poco más complejas de utilizar y si no se tiene un control adecuado, una lista enlazada en un índice de la tabla se puede volver demasiado grande haciendo que los tiempos de búsqueda y eliminación de datos se demore más de los esperado.

- Otra de las dificultades, fue con respecto a las tablas Hash, hubo dificultades, puesto que en ciertas ocasiones calculaba el mismo hash para algunos (diferentes) valores y fue necesario realizar varios ajustes.

- Realizar un cambio "pequeño" en la implementación de un tipo de dato abstracto cómo el árbol de búsqueda binaria puede ocasionar que sus operaciones corran en menos tiempo que la implementación anterior cómo nos ocurrió con el árbol AVL y el árbol BST desbalanceado.

- Y por último, una de las mayores dificultades, fue el tiempo, ya que este estuvo muy limitado, causando que no se pudiesen realizar algunas cosas como una interfaz gráfica para el prototipo.

XIV. REFERENCIAS BIBLIOGRÁFICAS

1. "Sólo el 28% de los estudiantes se gradúan a tiempo de la Universidad ",2010 Portafolio https://www.portafolio.co/economia/finanzas/28-estudiantes-gradua-universidad-144190
2. Ming Chen, Xuan Huang, Hongyu Chen, Xuemei Su & Jasmine Yur-Austin (2021): Data driven course scheduling to ensure timely graduation, International Journal of Production Research, DOI: 10.1080/00207543.2021.1916118
3. Gonzalo Gabriel Méndez, Luis Galárraga and Katherine Chiluiza. 2021. Showing Academic Performance Predictions during Term Planning: Effects on Students' Decisions, Behaviors, and Preferences. In CHI Conference on Human Factors in Computing Systems(CHI'21), May 8–13, 2021, Yokohama, Japan. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3411764.3445718
4. Guardex, "Build Progressive Web Applications with ASP.NET Core Blazor WebAssembly", *Microsoft,*[Online], Available: https://docs.microsoft.com/en-us/aspnet/core/blazor/progressive-web-app?view=aspnetcore-6.0&tabs=visual-studio
5. Blazor," *Wikipedia*. Sep. 22, 2021. Accessed: Nov. 26, 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Blazor&oldid=1045703149