Optimización de la Búsqueda de Patrones Genéticos mediante Árboles

Integrantes: Constanza Araya (GitHub/Cnstnnza)

Profesor: Christian Vasquez (GitHub/chvasquezr)

17 de noviembre de 2025

Índice

1. Introduccion

Este proyecto tiene como objetivo crear un programa que lea una secuencia de ADN (formada por letras A, C, G y T) y busque genes cortos dentro de ella. Para ello se utilizaran diversas estructuras de datos. Se implementara un arbol 4-ario (tipo trie) para guardar las combinaciones posibles de genes y listas enlazadas para guardar las posiciones donde aparecen. Se implementara la deteccion de comandos, entre ellos estaran:

- **bio start m:** crea un arbol 4-ario con genes de tamaño m
- **bio read archivo.txt:** lee la secuencia de ADN desde el archivo archivo.txt
- **bio search XX:** busca el gen XX en la secuencia
- **bio max:** muestra el gen mas repetido en la secuencia
- **bio min:** muestra el gen menos repetido en la secuencia
- **bio all:** muestra todos los gen que tienen frecuencia ¿0
- **bio exit:** libera memoria y sale del programa

El proyecto se desarrolla en el lenguaje de programacion C.

2. Objetivos

- Implementar un sistema de búsqueda de patrones utilizando árboles tries.
- Implementar y manipular estructuras de datos abstractas como árboles y listas enlazadas para almacenar y gestionar los datos manipulados.
- Desarrollar habilidades en programación en lenguaje C, centrándose en el manejo de memoria, punteros y eficiencia algorítmica.

3. Estructuras de datos

Para la implementación del proyecto se utilizaron las siguientes estructuras de datos:

- Lista enlazada (Linked List)
- Arbol 4-ario (Trie)

3.1. Lista enlazada

Una **lista enlazada** es una estructura de datos dinámica que consiste en una serie de elementos llamados nodos, donde cada nodo contiene un dato y una referencia al siguiente nodo en la secuencia. A diferencia de un array, las listas enlazadas permiten un crecimiento dinámico sin un tamaño fijo y facilitan la inserción y eliminación de elementos, ya que solo requieren actualizar referencias.

Sin embargo, las listas enlazadas presentan tanto ventajas como desventajas frente a los arreglos:

- **Ventajas:** Inserciones y eliminaciones en cualquier posición sin necesidad de mover todos los elementos. Crecimiento dinámico sin tamaño predefinido.
- **Desventajas:** La búsqueda de un dato específico requiere recorrer la lista nodo por nodo ($O(N)$), mientras que en un arreglo, si se conoce el índice, se puede acceder directamente en $O(1)$.

Para comprender mejor una lista enlazada es importante saber que es un nodo.

3.1.1. Nodo

En informática, un nodo es un elemento individual de una estructura de datos que almacena información y, en muchos casos, tiene referencias o enlaces a otros nodos. Los nodos son la base de muchas estructuras de datos dinámicas, como listas enlazadas, árboles y grafos.
Un nodo tiene dos componentes principales:

- **Dato**: La información que el nodo almacena, que puede ser de cualquier tipo de dato, como un número, una cadena de texto o un objeto más complejo.
- **Referencia**:Un puntero o referencia al siguiente nodo en la lista

En la siguiente imagen se puede apreciar de manera grafica que es un nodo y tambien un extracto de codigo en C que representa un nodo de una lista enlazada.

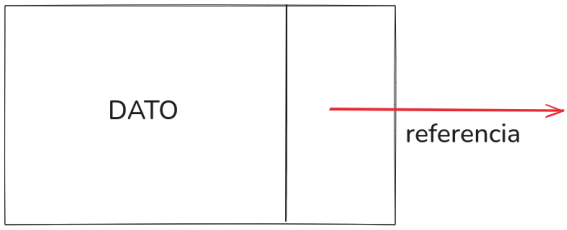

Figura 1: Nodo

```c
/*Definici n del nodo*/
typedef struct Nodo {
    int dato;               /*Valor almacenado*/
    struct Nodo* siguiente; /*Puntero al siguiente nodo*/
} Nodo;

/*Creaci n de un nuevo nodo*/
Nodo* crearNodo(int valor) {
    Nodo* nuevo = (Nodo*) malloc(sizeof(Nodo));
    nuevo->dato = valor;
    nuevo->siguiente = NULL;
    return nuevo;
}

```

Listing 1: Creacion de Nodo

Las listas enlazadas tienen un nodo inicial llamado **head** el cual apunta al primer elemento de la lista. Si se llega a un nodo que no apunta a ningún otro, se dice que dicho nodo apunta a **null** o final de la lista.
En la siguiente imagen se puede apreciar de manera grafica que es una lista enlazada y su implementacion en C

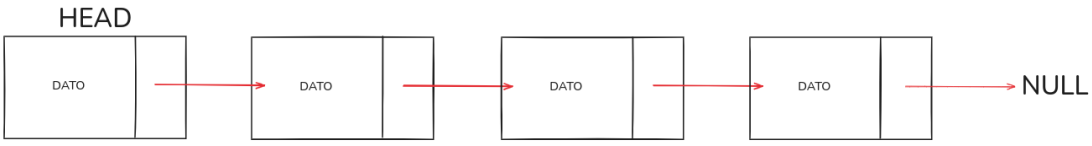

Figura 2: Lista enlazada

```c
/*Inserta un nodo al final de la lista*/
void insertarFinal(Nodo** cabeza, int valor) {
    Nodo* nuevo = crearNodo(valor);
    if (*cabeza == NULL) {
```

```
            *cabeza = nuevo;
            return;
        }
        Nodo* actual = *cabeza;
        while (actual->siguiente != NULL) {
            actual = actual->siguiente;
        }
        actual->siguiente = nuevo;
    }

```

Listing 2: Estructura lista enlazada

La lista enlazada, se utiliza dentro del arbol 4-ario para almacenar las posiciones donde aparecen los genes. En el nodo hoja del Trie (aquel que corresponde al final de un gen de longitud m), se almacena un puntero a una lista (`ListaInt* lista_posiciones`). Cada nodo de esta lista (ListaInt) guarda un entero: la posición (índice 0-basado) donde ese gen comienza en la secuencia S. Es decir cada vez que el programa detecta un gen (una secuencia de longitud m) dentro de la secuencia principal de ADN, se almacena su posición inicial (el índice donde empieza).

Como un mismo gen puede aparecer muchas veces en distintas partes de la secuencia, se necesita una estructura que permita agregar fácilmente nuevas posiciones sin un tamaño fijo ni un límite predefinido. Aqui es donde se utiliza la **lista enlazada**. Tambien permite que el programa no solo diga si un gen existe, sino también cuántas veces (bio search, bio max, bio min) y dónde comienza cada aparición.

3.2. Arboles

Los Árboles son las estructuras de datos mas utilizadas, pero también una de las mas complejas, Los Árboles se caracterizan por almacenar sus nodos en forma jerárquica y no en forma lineal como las Listas enlazadas, Colas,Pilas,etc.

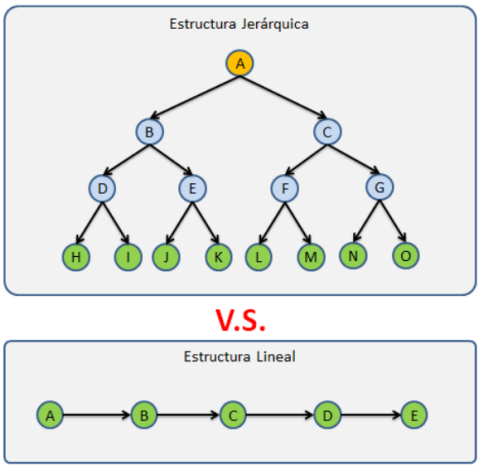

Figura 3: Visualización de árbol vs lista

Para comprender que es un arbol, es importante entender como esta estructurado y cuales son sus componentes. Algunos de ellos son:

- **Nodo:** Se le llama nodo a cada elemento que contiene un arbol.
- **Raiz:** Es el nodo principal del arbol, es el primer nodo del arbol, solo puede haber un nodo como raiz.
- **Padre:** Este termino representa a cualquier nodo que tiene al menos un hijo.
- **Hijo:** Los hijos son los nodos que tienen un padre.
- **Nodo hermano:** Los nodos hermanos son aquellos que comparten el mismo padre.
- **Hoja:** Son los nodos que no tienen hijos, por lo tanto siempre se encuentran en los extremos de un arbol.

Puede ser algo dificil de comprender, por lo que a continuacion se muestra una imagen que ejemplifica los conceptos antes mencionados.

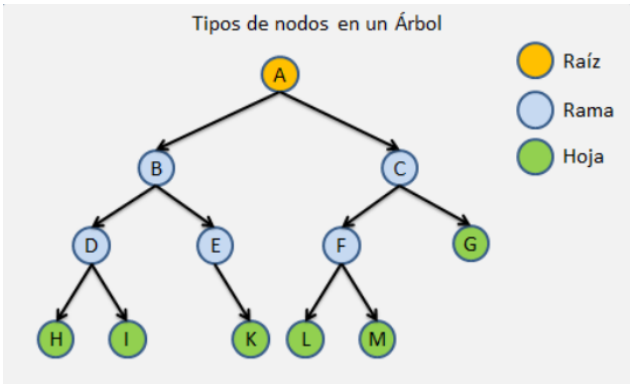

Figura 4: Componentes de un arbol

Los árboles a demas de nodos tienen otras propiedades importantes como:

- **Nivel:** El nivel es cada generacion dentro del arbol, por ejemplo: si se tiene un nodo hoja y se le agrega un hijo, entonces el nodo hoja se convierte en un nodo rama, pero a demas el arbol aumenta un nivel mas.
- **Altura:** La altura de un arbol es la cantidad de niveles que tiene el arbol.

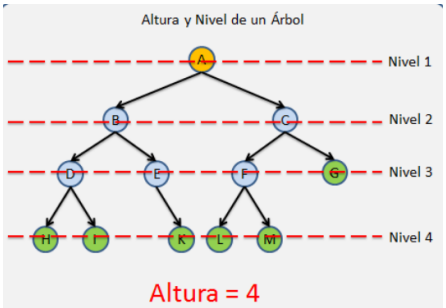

Figura 5: Nivel y altura de un arbol

- **Peso:** El peso corresponde a la cantidad de nodos que tiene un arbol, es un concepto muy importante ya que permite tener una idea de que tan grande es el arbol y el tamaño de memoria que usara en ejecucion.

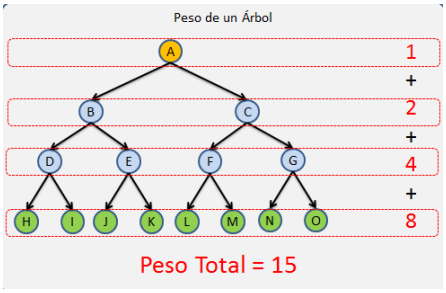

Figura 6: Peso de un arbol

A continuacion un estracto de codigo en C de un arbol binario:

```c
/*Estructura para un Nodo de  rbol  Binario (2 hijos)*/
typedef struct NodoArbol {
    int dato; /*El valor que almacena el nodo*/
    /*Punteros a los hijos (recursivos)*/
    struct NodoArbol* izquierda; /*Puntero al sub rbol izquierdo*/
    struct NodoArbol* derecha;  /*Puntero al sub rbol derecho*/
} NodoArbol;

```

Listing 3: Arbol binario

3.2.1. Arbol 4-ario (Trie)

Existen muchas formas de implementar un arbol, por ejemplo un arbol binario, pero en este proyecto se utilizara un arbol 4-ario (tipo trie). Este tipo de arbol tiene 4 hijos posibles, uno por cada letra del ADN (A, C, G y T). Los tries se usan comunmente en sistemas de busqueda de patrones porque permiten recorrer y almacenar grandes cantidades de datos de forma ordenada y eficiente. Gracias a su estructura, es posible encontrar patrones dentro de una secuencia de

manera rapida sin necesidad de comparar cada combinacion manualmente.

El arbol 4-ario se puede representar mediante la siguiente imagen:

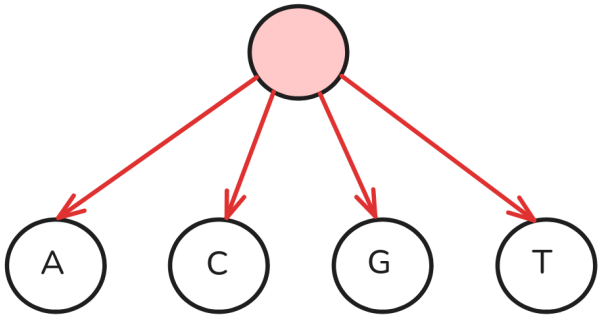

Figura 7: Arbol 4-ario

4. Implementacion de comandos

En este proyecto se implementaron principalmente siete comandos para interactuar con el programa. Estos comandos permiten al usuario iniciar el arbol, leer la secuencia de ADN, buscar genes,etc.

4.1. bio start m

Este comando crea un arbol de orden m, donde m es el tamaño de la secuencia de ADN.

```
            Nodo* bio_start(int m) {
            if (m <= 0) {
                printf("Error: El tamano del gen (m) debe ser mayor a 0.\n");
                return NULL;
            }
            if (TRIE_ROOT != NULL) {
                bio_exit();
            }
            GEN_SIZE = m;
            TRIE_ROOT = generar_arbol_recursivo(0, m);

            if (TRIE_ROOT != NULL) {
                printf("Arbol creado con tamano %d\n", m);
            } else {
                printf("Error al crear el arbol.\n");
            }
            return TRIE_ROOT;
        }

```

Listing 4: Implementacion bio start m

4.2. bio read archivo.txt

Este comando lee la secuencia de ADN desde el archivo archivo.txt.

```
            /*Logica para leer toda la secuencia S*/
            fseek(file, 0, SEEK_END);
            long n_size = ftell(file);
            fseek(file, 0, SEEK_SET);

            char* secuencia_S = (char*)malloc(n_size + 1);
            /*Verificaci n de malloc*/
            size_t read_len = fread(secuencia_S, 1, n_size, file);
            fclose(file);
```

```

```

Listing 5: Fragmento clave read archivo.txt

Aginacion dinamica: Se utiliza fseek y ftell para determinar el tamaño exacto del archivo (n) y luego se asigna dinámicamente el buffer `secuencia_S` con malloc. Esto garantiza que la secuencia genética completa se cargue en la memoria (heap) de un solo golpe con fread, mejorando la eficiencia de lectura del archivo.

4.3. bio search XX

La función `bio_search` implementa el núcleo del sistema de búsqueda. Su objetivo es encontrar la frecuencia y todas las posiciones de inicio de un gen específico (de longitud m) dentro de la secuencia genética S previamente cargada en el Árbol Trie

```
            /** Verificar que el arbol esta inicializado **/
            if (TRIE_ROOT == NULL || GEN_SIZE == 0) return -1;
            /** Validar que el gen tenga el largo correcto **/
            if ((int)strlen(gen_G) != GEN_SIZE) return -1;
            /** Recorrer el trie segun cada caracter del gen **/
            for (int i = 0; i < GEN_SIZE; i++) {
                int index = get_base_index(gen_G[i]);
                if (index == -1) return -1;     /** caracter invalido **/
                actual = actual->hijos[index];  /** avanzar en el arbol **/
            }
            /** Contar cuantas veces aparece el gen **/
            int frecuencia = contar_posiciones(actual->lista_posiciones);
            if (frecuencia == 0) return -1;
            /** Imprimir el gen, su frecuencia y posiciones **/
            printf("%s %d", gen_G, frecuencia);
            imprimir_posiciones(actual->lista_posiciones);

```

Listing 6: Fragmento clave bio search XX

4.4. bio max, bio min, bio all

Estos comandos permiten obtener el gen mas repetido (bio max), el gen menos repetido (bio min) y todos los gen que tienen frecuencia ¿0 (bio all). A continuacion extractos de cada uno de estos comandos

4.4.1. bio max

Busca dentro del árbol el gen con mayor frecuencia. Se realiza un recorrido completo (DFS) y en cada hoja se compara la frecuencia actual con la maxima encontrada hasta el momento.

```
                void bio_max() {
                    if (TRIE_ROOT == NULL || GEN_SIZE == 0) { printf("Error: Arbol no
    cargado.\n"); return; }
                    char* gen_buffer = (char*)malloc(GEN_SIZE + 1);
                    if (gen_buffer == NULL) { perror("Error de asignacion de memoria");
    return; }
                    int max_freq = 0;
                    recorrido_extremos(TRIE_ROOT, gen_buffer, 0, 1, &max_freq);
                    if (max_freq == 0) { printf("No hay genes presentes.\n"); free(
    gen_buffer); return; }
                    imprimir_recorrido(TRIE_ROOT, gen_buffer, 0, max_freq);
                    free(gen_buffer);
                }

```

Listing 7: Funcion bio max

4.4.2. bio min

Similar a bio max, pero busca el gene con frecuencia minima ¿0. Ignora hojas sin apariciones. Comparte la misma estrategia de recorrido pero con la condicion inversa.

```
            void bio_min() {
                if (TRIE_ROOT == NULL || GEN_SIZE == 0) { printf("Error: Arbol no cargado.\n
    "); return;
                char* gen_buffer = (char*)malloc(GEN_SIZE + 1);
                if (gen_buffer == NULL) { perror("Error de asignacion de memoria"); return;
    }
                int min_freq = INT_MAX;
                recorrido_extremos(TRIE_ROOT, gen_buffer, 0, 0, &min_freq);
                if (min_freq == INT_MAX) { printf("No hay genes presentes.\n"); free(
    gen_buffer); return; }
                imprimir_recorrido(TRIE_ROOT, gen_buffer, 0, min_freq);
                free(gen_buffer);
            }

```

Listing 8: Funcion bio min

4.4.3. bio all

Recorre todas las hojas del arbol y muestra solo aquellas cuyo conteo de apariciones es mayor a cero. Es una enumeración completa de todos los genes presentes en los datos cargados.

```
            void bio_all() {
                if (TRIE_ROOT == NULL || GEN_SIZE == 0) { printf("Error: Arbol no cargado.\n
    "); return; }
                char* gen_buffer = (char*)malloc(GEN_SIZE + 1);
                if (gen_buffer == NULL) { perror("Error de asignacion de memoria"); return;
    }
                /*Funcion recursiva interna para ALL (imprime todos los que tengan
    frecuencia > 0)*/
                void bio_all_recursivo(Nodo* nodo, char* buffer, int nivel) {
                    if (nodo == NULL) return;
                    if (nivel == GEN_SIZE) {
                        int frecuencia = contar_posiciones(nodo->lista_posiciones);
                        if (frecuencia > 0) {
                            buffer[GEN_SIZE] = '\0';
                            printf("%s %d", buffer, frecuencia);
                            imprimir_posiciones(nodo->lista_posiciones);
                            printf("\n");
                        }
                        return;
                    }
                    const char bases[] = {'A', 'C', 'G', 'T'};
                    for (int i = 0; i < 4; i++) {
                        buffer[nivel] = bases[i];
                        bio_all_recursivo(nodo->hijos[i], buffer, nivel + 1);
                    }
                }
                bio_all_recursivo(TRIE_ROOT, gen_buffer, 0);
                free(gen_buffer);
            }

```

Listing 9: Funcion bio all

El arreglo `gen_buffer` se utiliza para reconstruir el gen letra por letra durante el recorrido del árbol. Como cada nivel del Trie representa una base del ADN, el buffer permite almacenar la 'ruta' tomada para llegar a cada hoja, reconstruyendo asi el gen completo sin necesidad de almacenarlo textualmente en cada nodo (lo cual ahorra memoria).

Complejidad computacional

- El recorrido completo del Trie tiene complejidad $O(4^m)$ donde $m = GEN_SIZE$ (longitud del gen).

- Esto es eficiente porque el tamaño del alfabeto es fijo (A,C,G,T) y m es pequeño y constante en el contexto biologico.
- Las funciones bio max, bio min y bio all tienen la misma complejidad porque requieren recorrer todas las hojas.

4.5. bio exit

void bio$_{e}xit()liberar_{a}rbol(TRIE_{R}OOT); TRIE_{R}OOT = NULL; GEN_{S}IZE = 0; printf("Limpiando cache y saliendo del programa...");

4.6. help

Este comando es un adicional, ya que no estaba en los requerimientos iniciales, pero se considero importante para el usuario. Ya que al iniciar el programa no hay nada que indique como funciona si el usuario no tiene conocimientos previos. Para todo lo relacionado con la salida por terminal se utilizo definicion de macros con el fin de tener una interfaz mas atractiva para el usuario.
A continuacion la salida por terminal de la funcion `help`:

```
COMANDOS DISPONIBLES:
bio start m: crea un arbol 4-ario con genes de tamaño m
bio read archivo.txt: lee la secuencia de ADN desde el archivo archivo.txt
bio search XX: busca el gen XX en la secuencia
bio max: muestra el gen mas repetido en la secuencia
bio min: muestra el gen menos repetido en la secuencia
bio all: muestra todos los gen que tienen frecuencia > 0
bio exit: libera memoria y sale del programa
>bio
```

Figura 8: Salida por terminal del comando help

5. Diseño del programa (arquitectura general)

Al ejecutar el programa pasa por los siguientes pasos:

- bio start 4: Crea un arbol 4-ario con genes de tamaño 4.
- bio read adn.txt: Lee la secuencia de ADN desde el archivo adn.txt.
- bio search XX: Busca el gen XX en la secuencia.
- bio maz/min/all: recorren todo el arbol y muestran los genes que cumplen con la condición.
- bio exit: libera memoria y sale del programa.

Al ejecutar se tiene la siguiente salida por terminal:

```
>bio start 4
Arbol creado con tamano 4
>bio read adn.txt
Secuencia S leida desde el archivo
>bio search GATA
GATA 2 0 9
>bio search ATAC
ATAC 2 1 10
>bio search TOCA
-1
>bio min
AATT 1 5
ACAA 1 3
ACAG 1 12
ATTG 1 6
CAAT 1 4
TGAT 1 8
TTGA 1 7
>bio max
ATAC 2 1 10
GATA 2 0 9
TACA 2 2 11
>bio all
AATT 1 5
ACAA 1 3
ACAG 1 12
ATAC 2 1 10
ATTG 1 6
CAAT 1 4
GATA 2 0 9
TACA 2 2 11
TGAT 1 8
TTGA 1 7
>bio exit
Limpiando cache y salendo del programa...
```

Figura 9: Salida por terminal del programa

6. Manejo de memoria

El programa hace un uso intensivo de memoria dinamica, ya que tanto el arbol (trie) como las listas enlazadas se construyen en tiempo de ejecucion a partir de la secuencia de ADN cargada. Para ello se utilizan los siguientes mecanismos:

- **malloc:** Se emplea para reservar memoria de forma dinamica para cada nodo del arbol y para cada nodo de las listas enlazadas donde se almacenan las posiciones de aparicion de cada gen.
- **free:** Se utiliza para liberar toda la memoria asignada una vez que el programa finaliza o cuando se ejecuta el comando `bio exit`. Esto evita fugas de memoria (memory leaks).
- **Estructuras dinamicas:** El arbol se implementa como un trie 4-ario, donde cada nodo contiene un arreglo de cuatro punteros (`A, C, G, T`). En las hojas del arbol, cada gen almacena una lista enlazada con todas las posiciones donde aparece en la secuencia.

Cada nodo del arbol se reserva mediante `malloc`, incluyendo su arreglo de punteros a hijos. De forma similar, cada posicion encontrada en la secuencia se inserta en una lista enlazada, cuyos nodos tambien son reservados dinamicamente. Para asegurar una correcta administracion de memoria, el programa realiza un recorrido postorden del arbol al momento de liberar los recursos. En este proceso:

1. Se liberan primero todas las listas enlazadas asociadas a cada hoja.
2. Luego se libera cada nodo del arbol recursivamente.
3. Finalmente, se libera la memoria utilizada por la secuencia principal y demas buffers auxiliares.

Esto garantiza que ninguna referencia quede perdida y que toda la memoria solicitada al sistema sea correctamente devuelta.

7. Decisiones de Diseño del Proyecto

Durante el desarrollo de *BioSearch* se tomaron diversas decisiones de diseño con el objetivo de asegurar un funcionamiento eficiente, modular y escalable del sistema. En esta sección se describen las principales decisiones adoptadas y sus motivaciones.

7.1. Estructura Principal del Árbol Trie

Se decidió utilizar un **trie 4-ario**(teniendo en cuenta que era un requisito del proyecto) como estructura principal para almacenar las secuencias genéticas. Este trie contempla cuatro posibles hijos por nodo, correspondientes a las bases nitrogenadas `A, C, G, T`. La elección de un trie responde a las siguientes consideraciones:

- Permite realizar **búsquedas de patrones de tamaño fijo** en tiempo lineal respecto a la longitud del gen.
- Ofrece una representación compacta y ordenada de las secuencias.
- Facilita el almacenamiento independiente de cada gen mediante rutas únicas en el árbol.

7.2. Almacenamiento de Posiciones

Para registrar todas las apariciones de un gen en el archivo de entrada se optó por utilizar una **lista enlazada simple**. Esta estructura permite:

- Insertar nuevas posiciones de forma eficiente.
- Guardar múltiples ocurrencias sin necesidad de redimensionar arreglos.
- Iterar sobre las posiciones para imprimir resultados en las operaciones `max`, `min` y `all`.

7.3. Gestión del Largo del Gen (`m`)

El proyecto requiere trabajar con patrones de tamaño fijo. Por este motivo se decidió:

- Solicitar el valor de `m` mediante el comando `start`.
- Mantenerlo como una variable global para permitir su uso en funciones críticas como lectura, inserción y búsqueda.

7.4. Estructura del Programa y Modularidad

El código se dividió en múltiples módulos con el fin de lograr una arquitectura clara y mantenible:

- `trie.h` / `trie.c`: construcción, búsqueda y manejo del árbol.
- `lista.h` / `lista.c`: manejo de posiciones.
- `main.c`: procesamiento de comandos y flujo principal del programa.

Esta separación mejora la legibilidad, facilita las pruebas individuales y permite modificar o extender partes del sistema sin afectar al resto.

7.5. Interfaz de Comandos

Se definió una pequeña interfaz interactiva que utiliza comandos como `start`, `read`, `search`, `max`, `min` y `all`. Este diseño permite:

- Ofrecer al usuario un control directo sobre cada operación.
- Simplificar la ejecución sin depender de argumentos por línea de comandos.
- Mostrar mensajes de error claros cuando los argumentos son incorrectos.

7.6. Manejo de Memoria

Se dio especial importancia al manejo correcto de memoria dinámica:

- Cada nodo del trie se asigna mediante `malloc()`.
- Las listas enlazadas para posiciones se crean dinámicamente.
- Se implementó la función `liberar_arbol` para liberar recursivamente todo el trie.
- El comando `exit` garantiza un cierre seguro y sin fugas de memoria.

7.7. Justificación General

En conjunto, estas decisiones permiten que el sistema:

- Tenga un rendimiento adecuado incluso con archivos grandes.

- Sea fácil de extender con nuevas funcionalidades.
- Mantenga una estructura clara y coherente.
- Administre correctamente memoria y errores.

8. Conclusion

El desarrollo de este proyecto permitio implementar un sistema eficiente para el analisis de secuencias de ADN mediante el uso de estructuras de datos dinamicas. El arbol 4-ario (trie) demostro ser especialmente adecuado para almacenar y buscar genes de longitud fija, permitiendo realizar consultas en tiempo O(m) sin depender del tamano total de la secuencia. Complementariamente, las listas enlazadas utilizadas en cada nodo hoja permitieron registrar todas las posiciones de aparicion de cada gen, proporcionando informacion completa y facilmente accesible. Ademas, la gestion dinamica de memoria mediante `malloc` y `free` aseguro un uso responsable de los recursos, evitando fugas y permitiendo la liberacion ordenada de todas las estructuras al finalizar la ejecucion del programa. En conjunto, el proyecto integra de manera efectiva conceptos fundamentales como arboles, listas enlazadas, manejo de memoria y diseño modular. El resultado es una herramienta funcional que permite procesar y consultar secuencias geneticas de forma rapida, clara y eficiente.

Referencias

esantos. (2024, noviembre). *¿Qué es una Lista Enlazada?* Consultado el 13 de noviembre de 2025, desde https://www.ervinsantos.com/post/50-%C2%BFqu%C3%A9-es-una-lista-enlazada

oblancarte. (2021, agosto). *Estructura de datos Árboles*. Consultado el 13 de noviembre de 2025, desde https://www.oscarblancarteblog.com/2014/08/22/estructura-de-datos-arboles/

wikipedia. (2024, diciembre). *Nodo (informática)*. Consultado el 13 de noviembre de 2025, desde https://es.wikipedia.org/wiki/Nodo_(inform%C3%A1tica)