Universidad de Magallanes
Facultad de Ingeniería
Departamento de Ingeniería en Computación

Informe de Tarea N°3

Estudiantes: Benjamin Moya, Guillermo Vargas
Carrera: Ingeniería Civil en Computación e Informática
Departamento: Ingeniería en Computación
Profesor: Christian Vásquez Rebolledo
Fecha: 17 de Noviembre, 2025

1. Resumen

Este proyecto implementa un sistema de indexación y búsqueda de genes utilizando un árbol 4-ario (trie) construido en lenguaje C. El sistema permite procesar una secuencia de ADN y almacenar todas las subcadenas de longitud *m* (genes), guardando en cada nodo hoja las posiciones donde aparece cada gen dentro de la secuencia.

El programa implementa una interfaz de línea de comandos que permite iniciar el sistema con un tamaño de gen dado, cargar una secuencia desde archivo, buscar genes específicos, visualizar los genes más y menos frecuentes, y listar todos los genes registrados con al menos una aparición.

La implementación utiliza técnicas fundamentales de estructuras de datos: recursividad, árboles, arreglos dinámicos, gestión de memoria con `malloc`/`free`, y validación de entrada. Los resultados demostraron un funcionamiento correcto del trie, con un manejo adecuado de memoria, una inserción eficiente de genes y respuestas precisas en todas las operaciones solicitadas en la tarea.

2. Introducción y Objetivos

2.1 Introducción

El análisis de secuencias de ADN requiere mecanismos eficientes para detectar patrones repetidos, contar ocurrencias de genes y localizar posiciones relevantes dentro de cadenas largas. Para este propósito, los árboles Trie son estructuras altamente eficientes, ya que representan cadenas de forma jerárquica y permiten búsquedas en tiempo proporcional al tamaño del gen.

Este proyecto implementa un Trie 4-ario, donde cada nivel corresponde a un carácter del gen (A, C, G, T). Este Trie almacena las posiciones de aparición de cada patrón y permite realizar consultas rápidas mediante comandos CLI.

2.2 Objetivos

- Comprender y aplicar la estructura de datos **trie 4-ario** en un problema real.
- Procesar secuencias de ADN identificando todas las subcadenas de longitud *m*.
- Implementar listas dinámicas para almacenar posiciones de ocurrencias.
- Desarrollar una interfaz CLI robusta capaz de procesar comandos del usuario.
- Utilizar memoria dinámica correctamente evitando fugas.
- Validar datos de entrada y archivo según especificaciones.
- Simular consultas típicas: búsqueda directa, máximo, mínimo y listado general.

3. Marco Teórico

3.1 Trie 4-ario

Un Trie es una estructura de árboles donde cada nodo representa un carácter y cada camino desde la raíz hasta un nodo hoja forma una cadena.
En este proyecto el Trie es:

- 4-ario (A, C, G, T)
- De profundidad fija igual al tamaño del gen
- Determinista: cada gen tiene un único recorrido
- Con hojas que almacenan posiciones mediante una lista dinámica

El Trie permite búsquedas en tiempo O(m), donde m es el largo del gen.

3.2 Listas dinámicas (arreglos redimensionables)

Cada hoja almacena una estructura:

```
typedef struct {
    int *posiciones;
    int cantidad;
    int capacidad;
} ListaEnteros;
```

Estas listas funcionan como arreglos dinámicos con expansión geométrica (doble capacidad).
Ventajas:

- Permite almacenar un número variable de posiciones.
- Amortiza costos de `realloc`.

3.3 Gestión de memoria dinámica

El proyecto utiliza:

- `malloc` para nodos del trie.
- `malloc` y `realloc` para listas de posiciones.
- `free` recursivo para liberar la estructura completa.

El manejo seguro de memoria es clave para evitar *memory leaks* y fallos de segmentación.

4. Explicación del Código

4.1 Arquitectura del Sistema

El sistema está construido en un solo archivo `EDDGenes.c` e incluye:

- Definición del nodo del trie
- Definición de la lista dinámica
- Funciones de creación del trie
- Procesamiento de secuencias
- Comandos CLI
- Liberación de memoria
- Función `main()` como interfaz con usuario

4.2 Estructura del Nodo del Trie

```c
typedef struct NodoTrie
{
    struct NodoTrie* hijos[4];
    ListaEnteros* ocurrencias;
    int esHoja;
} NodoTrie;
```

Significado:

- `hijos[4]`: hijos para cada base.
- `es_hoja`: indica si el nodo está en nivel *m*.
- `lista`: puntero a lista dinámica (solo en hojas).

Invariantes:

- Solo hojas tienen lista.
- Los 4 hijos siempre existen (árbol completo).
- La raíz representa cadena vacía.

4.3 Función crear_nodo()

```c
NodoTrie* crear_nodo(int nivel, int nivel_max)
{
    NodoTrie* nodo = malloc(sizeof(NodoTrie));
    nodo->esHoja = (nivel == nivel_max);
    nodo->ocurrencias = nodo->esHoja ? malloc(sizeof(ListaEnteros)) : NULL;

    if (nodo->esHoja) {
        nodo->ocurrencias->posiciones = NULL;
        nodo->ocurrencias->cantidad = 0;
        nodo->ocurrencias->capacidad = 0;
    }

    for (int i = 0; i < 4; i++)
        nodo->hijos[i] = (nivel < nivel_max) ? crear_nodo(nivel + 1, nivel_max) : NULL;

    return nodo;
}
```

Crea un nuevo nodo del trie e inicializa sus hijos.

Responsabilidades:

- Asignar memoria.
- Inicializar punteros.
- Asegurar consistencia estructural.

4.4 Función agregar posición()

```
// Agrega la posición de un gen a la lista dinámica del nodo hoja.
// Maneja realocación dinámica para ampliar la lista.
void agregar_posicion(ListaEnteros* lista, int posicion)
{
   int* nuevas_posiciones;
   if (lista->cantidad == lista->capacidad)
   {
       lista->capacidad = (lista->capacidad == 0) ? 8 : lista->capacidad * 2;
       nuevas_posiciones = (int*)realloc(lista->posiciones, lista->capacidad *
sizeof(int));
       if (!nuevas_posiciones) return;
       lista->posiciones = nuevas_posiciones;
   }
   lista->posiciones[lista->cantidad] = posicion;
   lista->cantidad++;
}
```

Responsabilidades:

- Insertar una nueva posición en la lista dinámica asociada a un gen.
- Aumentar la capacidad del arreglo cuando está lleno mediante `realloc`.
- Mantener el arreglo ordenado por inserción sin perder datos.
- Garantizar que la estructura crezca de forma eficiente (doblando la capacidad).

4.5 Inicialización del Trie

```
// Inicializa la raíz del trie para genes de tamaño tam_gen.
NodoTrie* inicializar_trie(int tam_gen)
{
   NodoTrie* raiz = crear_nodo(0, tam_gen);
   return raiz;
}
// Inserta una ocurrencia de un gen en el trie con su posición dentro de la
secuencia.
void insertar_gen(NodoTrie* raiz, const char* gen, int tam_gen, int posicion)
{
   int i, indice;
   NodoTrie* actual = raiz;
   for (i = 0; i < tam_gen; i++)
   {
       indice = base_a_indice(gen[i]);
       if (indice == -1) return;
       actual = actual->hijos[indice];
   }
   agregar_posicion(actual->ocurrencias, posicion);
}
```

Responsabilidades:

- Crear la raíz del trie con altura igual al tamaño del gen.
- Construir todos los nodos necesarios mediante llamadas recursivas a `crear_nodo()`.
- Asegurar que cada nivel esté correctamente inicializado para aceptar bases A, C, G y T.
- Preparar la estructura para almacenar ocurrencias en nodos hoja.

4.6 Insertar un gen

```
void insertar_gen(NodoTrie* raiz, const char* gen, int tam_gen, int posicion)
{
    NodoTrie* actual = raiz;
    for (int i = 0; i < tam_gen; i++)
    {
        int indice = base_a_indice(gen[i]);
        if (indice == -1) return;
        actual = actual->hijos[indice];
    }
    agregar_posicion(actual->ocurrencias, posicion);
}
```

- Navega los m niveles usando base → índice
- Llega a la hoja
- Agrega la posición

4.7 Procesar una secuencia

```
void procesar_secuencia(NodoTrie* raiz, const char* secuencia, int largo, int
tam_gen)
{
    char gen[TAM_GEN_MAXIMO + 1];
    for (int i = 0; i <= largo - tam_gen; i++)
    {
        strncpy(gen, secuencia + i, tam_gen);
        gen[tam_gen] = '\0';
        insertar_gen(raiz, gen, tam_gen, i);
    }
}
```

- Lee el archivo
- Extrae cada subcadena de largo *m*
- Llama a `insertar_gen`
- Valida cada base

Este proceso recorre la secuencia una sola vez.

4.8 Buscar un gen

```
ListaEnteros* buscar_gen(NodoTrie* raiz, const char* gen, int tam_gen)
{
    NodoTrie* actual = raiz;
    for (int i = 0; i < tam_gen; i++)
    {
        int indice = base_a_indice(gen[i]);
        if (indice == -1) return NULL;
        actual = actual->hijos[indice];
        if (!actual) return NULL;
    }
    return actual->ocurrencias;
}
```

- Valida entrada
- Navega por los nodos
- Si encuentra hoja imprime posiciones
- Si está vacío imprime “-1”

4.9 Búsqueda de máximos y mínimos

```
// Función recursiva para recorrer el trie y encontrar genes con máximo y mínimo número
de repeticiones.
// Guarda los genes y cantidades máximas y mínimas encontradas.
void recorrer_trie(NodoTrie* nodo, int nivel, int nivel_max, char* buffer,
                   int* max_count, int* min_count)
{
    int i;

    if (!nodo) return;

    // Caso base: llegamos a una hoja (gen completo)
    if (nivel == nivel_max)
    {
        int cantidad = nodo->ocurrencias->cantidad;

        if (cantidad > 0)
        {
            buffer[nivel] = '\0';

            // --- MAX ---
            if (cantidad > *max_count)
            {
                *max_count = cantidad;
                max_count_genes = 0;  // reiniciar lista de genes max
```

```
                strcpy(max_genes[max_count_genes++], buffer);
            }
            else if (cantidad == *max_count)
            {
                strcpy(max_genes[max_count_genes++], buffer);
            }

            // --- MIN ---
            if (*min_count == -1 || cantidad < *min_count)
            {
                *min_count = cantidad;
                min_count_genes = 0;  // reinicia lista min
                strcpy(min_genes[min_count_genes++], buffer);
            }
            else if (cantidad == *min_count)
            {
                strcpy(min_genes[min_count_genes++], buffer);
            }
        }
        return;
    }

    // Paso recursivo: se exploran los 4 hijos (A, C, G, T)
    for (i = 0; i < 4; i++)
    {
        buffer[nivel] = indice_a_base(i);
        recorrer_trie(nodo->hijos[i], nivel + 1, nivel_max, buffer, max_count,
min_count);
    }
}
```

Para determinar cuáles genes son los más y menos repetidos dentro de la secuencia completa, el programa realiza un recorrido exhaustivo del Trie usando una función recursiva.

Responsabilidades:

- Recorrer todo el Trie de forma recursiva hasta las hojas.
- Evaluar cuántas veces aparece cada gen.
- Actualizar las listas globales de genes más repetidos (máximos) y menos repetidos (mínimos).
- Construir el gen actual mediante el buffer mientras se desciende por el árbol.
- Garantizar que todos los genes posibles sean examinados.

4.10 Comandos Implementados

Comando	Función actualizada
iniciar m	Construye el trie completo para genes de tamaño *m*.
leer archivo	Procesa la secuencia del archivo e inserta todos los genes en el trie.
buscar X	Muestra todas las posiciones donde aparece el gen X.
max	Muestra todos los genes con la mayor cantidad de apariciones, usando la lista `max_genes[]`.
min	Muestra todos los genes con la menor cantidad de apariciones (>0), usando la lista `min_genes[]`.
all	Imprime todos los genes que aparecen al menos una vez, junto con todas sus posiciones.
salir	Libera toda la memoria del trie y cierra el programa.

4.11 main()

El `main()` implementa un ciclo infinito que:

- Lee comandos
- Valida parámetros
- Usa comparaciones `strcmp`
- Llama a funciones respectivas

Incluye mensajes claros para errores y manejo de estado.

5. Datos Obtenidos

Ejemplo típico de ejecución:

```
bio> iniciar 2
Arbol creado con altura 2

bio> leer secuencia.txt
Secuencia leída del archivo

bio> buscar AC
0 5 9

bio> max
AC 0 5 9

bio> min
TT 3

bio> all
AA 7
AC 0 5 9
AG 2
AT 10
CA 4
CG 8
CT 1
GA 6
GC 11
GT 3
TA 12
TT 3

bio> salir
Limpiando memoria y saliendo
```

Las operaciones responden de forma correcta y consistente.

6. Análisis y Discusión de Resultados

6.1 Comportamiento del Trie

El trie cumplió con las propiedades esperadas:

- Inserciones correctas
- Búsquedas rápidas (O(m))
- Registros completos para cada gen
- Recorridos recursivos para obtener máximos/mínimos

6.2 Complejidad

- Construcción del trie: **O(4^m)** nodos
- Procesamiento secuencia: **O(n·m)**
- Búsqueda: **O(m)**

Ventaja: consultas rápidas
Desventaja: consumo elevado de memoria

6.3 Validación de entrada

El código valida:

- Tamaño del gen
- Archivo existente
- Base válida (A,C,G,T)
- Comandos válidos

6.4 Gestión de Memoria

Fortalezas:

- Uso correcto de `malloc`, `realloc` y `free`
- Liberación recursiva completa del trie
- Sin fugas detectadas en pruebas manuales

7. Desafíos Encontrados y Soluciones

7.1 Desafío 1 – Creación total del árbol

Problema: memoria de O(4^m) puede ser grande.

Solución: Se mantiene el árbol completo porque así lo exige el enunciado.
Alternativa futura: nodos *lazy*.

7.2 Desafío 2 – Validación de secuencia

Problema: secuencias grandes pueden superar el buffer fijo.

Solución: el código actual usa un buffer de 4096; se recomienda lectura dinámica.

7.3 Desafío 3 – Recorridos para max/min/all

Problema: recorrer un trie completo requiere lógica recursiva cuidadosa.

Solución: se implementó función `recorrer_trie` con buffer para construir cadenas.

7.4 Desafío 4 – Identificación de máximos y mínimos

Problema: algunos genes presentan empates en su frecuencia, por lo que es necesario registrar múltiples resultados para `max` y `min`.

Solución: se utilizaron listas globales que almacenan todos los genes empatados y se actualizan durante el recorrido del trie.

8. Metodología de Desarrollo

1. Diseño de estructuras (trie + listas).
2. Implementación de creación recursiva.
3. Diseño de inserción eficiente.
4. Manejo de archivo y procesamiento secuencial.
5. Implementación de consultas del usuario.
6. Liberación completa de memoria.
7. Pruebas unitarias de cada módulo.
8. Pruebas con secuencias reales.
9. Documentación e informe.

9. Conclusiones

El sistema desarrollado cumple satisfactoriamente con los objetivos de la tarea:

- Implementación correcta de un Trie de ADN.
- Inserción y búsqueda eficientes basadas en navegación determinística.
- Identificación precisa de patrones más y menos frecuentes.
- Interfaz CLI funcional y clara.
- Gestión de memoria adecuada utilizando funciones de liberación recursiva.

El programa ofrece una base sólida para análisis genético básico y puede ampliarse para soportar secuencias más grandes, trie dinámico o técnicas más eficientes de indexación.