Proyecto Entrega 2: Documento de diseño

https://github.com/CarefreeHarp/UsageOfSomeDataStructures

Guillermo Andrés Aponte Cárdenas

Daniel Felipe Ramírez Vargas

Pontificia Universidad Javeriana

Estructuras de Datos

Bogotá, 2025

Tabla de Contenido

Introducción

Este documento describe la aplicación de consola implementada para las Entregas 1 y 2 del proyecto. El objetivo es permitir manipulaciones básicas sobre genomas en formato FASTA usando estructuras lineales, incluyendo carga de archivos, consulta de secuencias, construcción de histogramas, búsqueda y enmascarado de subsecuencias, y persistencia en archivos. El documento consolida:

- La especificación formal de TADs (conjunto mínimo de datos y operaciones)- La especificación de cada función, método y operación, especificando entradas, salidas, precondiciones, postcondiciones y una explicación técnica de cada una.

- Análisis de complejidad de las funciones implementadas Se cuenta con un archivo FASTA que contiene una o varias secuencias. Cada secuencia inicia con una línea de cabecera cuyo primer carácter es '>' seguida del nombre/descripcion de la secuencia. Las líneas subsiguientes contienen las bases codificadas (A, C, G, T, U y otros símbolos extendidos), con justificación fija (mismo ancho por línea) salvo quizá la última. El sistema conserva esta justificación para lectura y escritura.

Adicional a lo mencionado, se complementaron los comentarios hechos por el profesor para la entrega anterior, los cuales fueron la falta de diagramas de relación entre TADS, falta de diagramas de relación entre módulos del proyecto, y falta del TAD Codigos en la primera entrega, todos estos comentarios son corregidos en el desarrollo del presente documento.

Estructura del proyecto

Archivos relacionados

main.cpp

Punto de entrada y consola interactiva.

componente1.hxx

Implementacion de funciones y métodos de TADs realizados para la primera entrega.

componente2.hxx

Implementacion de funciones y métodos de TADs realizados para la segunda entrega.

componente1.h

Definiciones de TADs de la primera entrega.

componente2.h

Definiciones de TADs de la segunda entrega.

archivo.fa

Ejemplo de genoma en FASTA

archivo.fabin

Ejemplo del genoma codificado mediante el árbol de huffman.

codigos.txt

Tabla de asociación símbolo → significado biológico.

comandos.guda

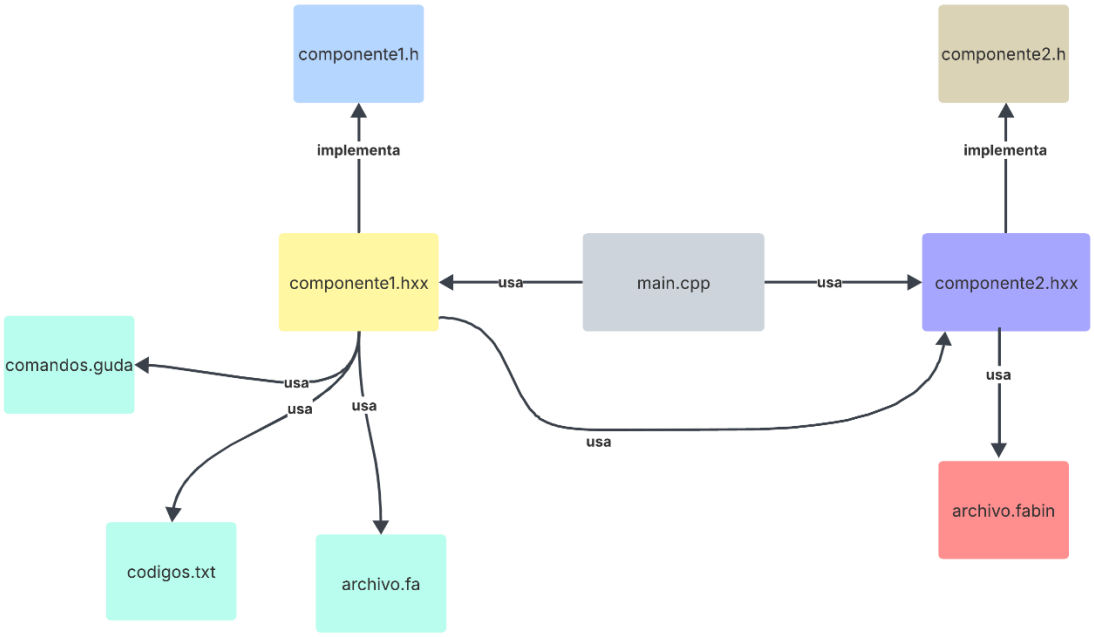
Archivo binario con definiciones de comandos.

Makefile

Compilar/ejecutar (Linux/macOS).

Gráfico

Gráfico 1, relación de los archivos del proyecto



TADS

Definición de TADS Primera Entrega

Comando

TAD Comando

Conjunto mínimo de datos:

nombre, cadena: identificador textual del comando.

argumentos, booleano: indica si el comando requiere argumento.

argumento, cadena: etiqueta/nombre del argumento requerido (si aplica).

descripcion, cadena: descripción corta del uso del comando.

posiblesSalidas, cadena: mensajes esperados que el comando imprime.

Operaciones:

obtenerNombre(): retorna el nombre actual del comando.

fijarNombre(nuevoNom): reemplaza el nombre del comando.

obtenerArgumentos(): retorna si el comando requiere argumento.

fijarArgumentos(requiere): establece si requiere argumento.

obtenerArgumento(): retorna la etiqueta/nombre del argumento.

fijarArgumento(nuevoArg): actualiza la etiqueta/nombre del argumento.

obtenerDescripcion(): retorna la descripción del comando.

fijarDescripcion(nuevaDesc): actualiza la descripción.

obtenerPosiblesSalidas(): retorna el texto de salidas posibles.

fijarPosiblesSalidas(nuevasSalidas): actualiza el texto de salidas posibles.

Codigos

TAD Codigos

Conjunto mínimo de datos:

codigo, lista de caracteres: Códigos de carácter de cada base en FASTA.

significado, lista de cadenas: Significado biológico de cada código del FASTA

aux, lista de cadenas: Lista auxiliar que permite cargar la información sin perder datos.

Operaciones:

cargar(), carga en memoria el archivo que contiene el carácter FASTA y su significado, paralelamente en las listas del TAD,

Secuencia

TAD Secuencia

Conjunto mínimo de datos:

nombre, cadena (máx. 50): descripción/nombre de la secuencia.

contenido, lista de cadenas (líneas): líneas de bases tal como se cargaron.

ancho, entero: ancho de línea (justificación) de la secuencia.

Operaciones:

obtenerNombre(): retorna el nombre/descripcion de la secuencia.

fijarNombre(nuevoNom): actualiza el nombre/descripcion.

obtenerContenido(): retorna el arreglo de líneas de la secuencia.

fijarContenido(nuevoContenido): reemplaza el arreglo de líneas.

obtenerAncho(): retorna el ancho de línea usado.

fijarAncho(nuevoAncho): actualiza el ancho de línea.

ListaSecuencias

TAD ListaSecuencias

Conjunto mínimo de datos:

secuencias, lista de Secuencia: colección de secuencias en memoria.

Operaciones:

obtenerSecuencias(): retorna la lista completa en memoria.

fijarSecuencias(nuevasSecuencias): reemplaza la colección por otra.

cargar(nombreArchivo): carga las secuencias en memoria desde un archivo de texto.

listarSecuencia(): Indica el numero de secuencias cargadas y la cantidad de sus bases.

histograma(nombre): Indica la cantidad de codigos por linea de una secuencia dada.

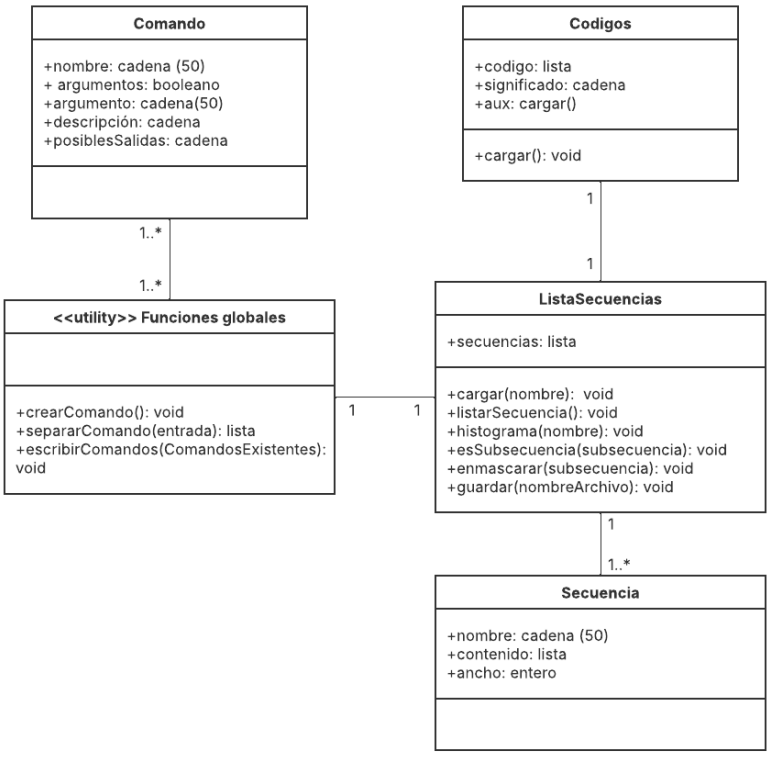
esSubsecuencia(subseq): Indica cuantas veces se repite subseq en todas las secuencias.

enmascarar(subseq): reemplaza cualquier ocurrencia de subseq por el código ‘X’ dependiendo de que tantos caracteres tenga.

guardar(nombreArchivo): guarda todas las secuencias en un archivo con nombre ‘nombreArchivo’.

Diagrama de Relación Primera Entrega

Grafico 2, relación de TADS primera entrega



Definición de TADS Segunda Entrega

NodoHuffman

TAD NodoHuffman

Conjunto mínimo de datos:

frecuencia, entero: Cantidad en que se repite un nombre identificado en el archivo FASTA.

nombre, cadena: nombre del nodo.

hijoIzquierdo, apuntador a NodoHuffman: hijo izquierdo del nodo.

hijoDerechi, apuntador a NodoHuffman: hijo derecho del nodo.

Operaciones:

NodoHuffman(nombre, valorFrecuencia, hijoDerecho, hijoIzquierdo): Constructor del TAD.

buscarHoja, cadena: Retorna el código necesario en el árbol para llegar a una hoja determinada.

ArbolDeCodificacionHuffman

TAD ArbolDeCodificacionHuffman

Conjunto mínimo de datos:

raiz, apuntador a NodoHuffman: raiz del árbol

Operaciones:

comprimirSecuencias(nombrefabin, secuenciasEnMemoria): recibe el nombre del archivo donde se comprimirán las secuencias y las secuencias en cuestión.

descomprimirSecuencias(nombrefabin, secuenciasEnMemoria): recibe el nombre del archivo donde se encuentran las secuencias comprimidas y las secuencias guardadas en memoria.

ElementoTablaDeHuffman

TAD ElementoTablaDeHuffman

Conjunto mínimo de datos:

nombre, cadena: Carácter único con el que se identifica un carácter presente en la secuencia y posteriormente guardado en la tabla.

numeroFrecuencia, entero: Cantidad de veces que se repite un carácter en la secuencia.

CompararFrecuencia

TAD CompararFrecuencia

Operaciones:

Operator(Nodo a, Nodo b), booleano: Recibe dos apuntadores a NodoHuffman, retorna cuál verdadero o falso según cual es mayor o menor y los ordena.

Diagrama de Relación Segunda Entrega

Gráfico 3, relación de TADS segunda entrega

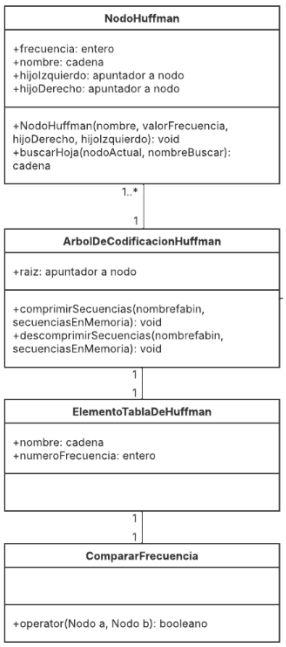
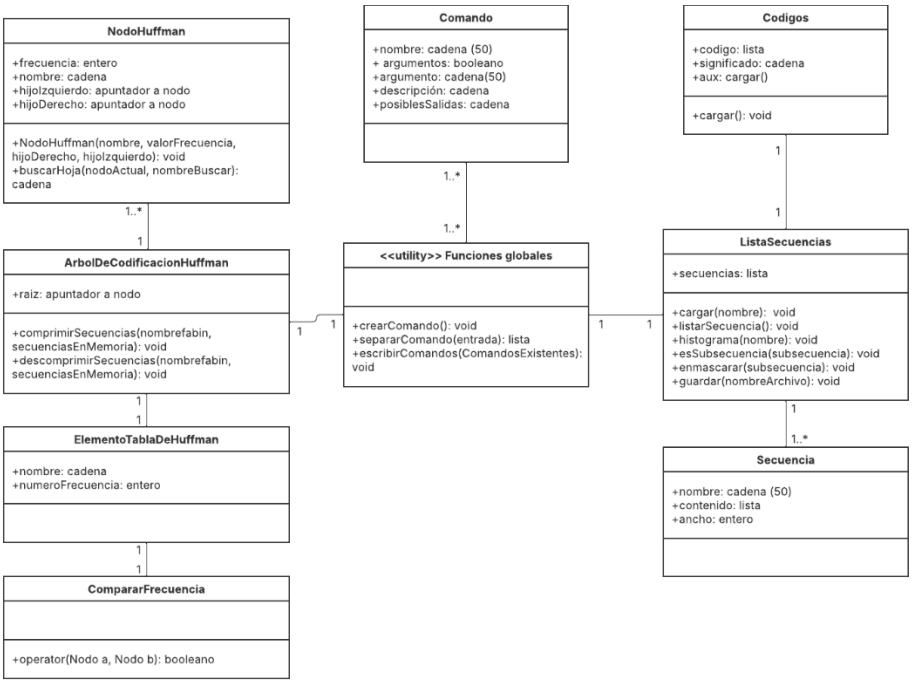


Diagrama de Relación Final

Grafico 4, diagrama de relación total



Funcionamiento General

Desde la funcion main, el programa inicia cargando la tabla de códigos desde codigos.txt (vía Codigos::cargar) y leyendo el catálogo de comandos guardados en comandos.guda; luego muestra el prompt $ y entra en un bucle de consola donde lee cada línea, la tokeniza con la función separarComando, valida el nombre del comando y su cantidad de argumentos contra ese catálogo, e imprime ayuda general o específica cuando se invoca ayuda/ayuda mediante la funcion escribirComandos() que también sirve para digitar cualquier otro comando. Si el usuario elige crearComando, el sistema añade un nuevo registro al archivo binario de comandos para usos futuros. El main delega en los métodos de ListaSecuencias que ejecutan la lógica y emiten exactamente los mensajes de éxito o error definidos por el enunciado; al terminar cada comando, el control vuelve al prompt hasta que se reciba salir, que cierra el bucle

Funciones

crearComando()

Da la opción de crear un nuevo comando y guardarlo en el archivo de comandos.guda. Complejidad O(1) (trabaja sobre tamaños fijos y solo es una función que utiliza I/O)

separarComando(entrada)

Tokeniza el parámetro ‘entrada’ y guarda la información en un arreglo de string. El vector resultante tiene el mismo uso que argsv en una función main. Complejidad O(n) (revisa carácter por carácter de la entrada para tokenizarla)

escribirComandos(ComandosExistentes)

La función escribirComandos implementa el bucle principal de la consola del sistema. Recibe como entrada un vector de objetos Comando, que contiene la definición de todos los comandos disponibles en el programa (su nombre, si requieren argumentos, su descripción y las salidas esperadas). No retorna ningún valor explícito, sino que interactúa con el usuario por consola: imprime mensajes, procesa las entradas que este escribe y ejecuta las funciones asociadas. El bucle se mantiene en ejecución hasta que el usuario ingresa el comando salir. El flujo inicia mostrando el símbolo $ como prompt. En cada iteración lee lo que el usuario escribe, lo separa en tokens con ayuda de la función separarComando y revisa el primer elemento como posible nombre de comando. Si el usuario escribe ayuda, la función responde de dos maneras: si no se indica un nombre adicional, imprime la lista completa de comandos disponibles; si se proporciona un nombre de comando, busca dentro del vector de ComandosExistentes y, si lo encuentra, muestra su descripción, la indicación de si requiere argumento y los mensajes de salida asociados. En caso de que el comando buscado no exista, informa que no se encontró ayuda. Para los demás comandos, el sistema valida primero que el nombre exista en el vector y que la cantidad de argumentos coincida con lo esperado (uno o dos según el caso). Si la validación es correcta, se ejecuta la acción asociada invocando métodos del objeto secuenciasEnMemoria de tipo ListaSecuencias. En caso de que un comando no exista en el catálogo, o de que la cantidad de argumentos no coincida con lo que requiere, la función muestra mensajes de error: “El comando no fue encontrado” o “La cantidad de

argumentos es incorrecta". Con esto, escribirComandos centraliza la interfaz entre el usuario y las operaciones de manejo de secuencias, actuando como un intérprete que controla entradas (líneas escritas en consola) y salidas (mensajes, listados o resultados de operaciones). Complejidad O(n) (depende de la cantidad de comandos registrados, ya que debe comprobar si existe en el arreglo de comandos existentes).

Operaciones TAD Codigos

cargar()

Inicializa la tabla de códigos biológicos a partir del archivo de texto codigos.txt. No recibe parámetros y trabaja siempre sobre este archivo fijo. Intenta abrirlo con un flujo de entrada y, si falla, muestra un mensaje de error y termina sin modificar los datos internos. Si la apertura es exitosa, lee línea por línea con un límite de 50 caracteres, reservando memoria dinámica para copiar cada línea completa. 7 En cada iteración extrae el primer carácter y lo almacena en el vector codigo, toma desde la tercera posición en adelante de la línea y lo guarda en el vector significado como la descripción del símbolo, y conserva un puntero a toda la cadena en el vector aux. La entrada, entonces, es el contenido de codigos.txt, y la salida son los tres vectores de la clase poblados con la asociación entre símbolo y significado, listos para ser usados por las demás operaciones del sistema. Complejidad O(n) (Debe leer todo el contenido del archivo).

Operaciones TAD ListaSecuencias

cargar(const char nombre[])

Inicializa la tabla de códigos biológicos a partir del archivo de texto codigos.txt. No recibe parámetros y trabaja siempre sobre este archivo fijo. Intenta abrirlo con un flujo de entrada y, si falla, muestra un mensaje de error y termina sin modificar los datos internos. Si la apertura es exitosa, lee línea por línea con un límite de 50 caracteres, reservando memoria dinámica para copiar cada línea completa. En cada iteración extrae el primer carácter y lo almacena en el vector codigo, toma desde la tercera posición en adelante de la línea y lo guarda en el vector significado como la descripción del símbolo, y conserva un puntero a toda la cadena en el vector aux. La entrada, entonces, es el contenido de codigos.txt, y la salida son los tres vectores de la clase poblados con la asociación entre símbolo y significado, listos para ser usados por las demás operaciones del sistema. Complejidad O(n) (Debe leer cada carácter del archivo)

listarSecuencia()

No recibe parámetros y trabaja directamente sobre el vector interno secuencias previamente cargado. Como salida, imprime en consola la cantidad de secuencias en memoria y un resumen de cada una. Para cada secuencia, recorre línea por línea y carácter por carácter de su contenido; si el símbolo leído no es un guion -, lo cuenta como una base válida, pero si encuentra un guion marca la secuencia como incompleta. Al final del recorrido de cada secuencia muestra su nombre y la cantidad de bases encontradas. Si no se hallaron guiones, indica que la secuencia “contiene X bases”; si sí había guiones, aclara que “contiene al menos X bases”. De esta manera, la función transforma las secuencias almacenadas en memoria en un reporte textual que distingue entre secuencias

completas e incompletas y entrega al usuario la información esencial de forma clara. Complejidad O(n) (Debe pasar por cada carácter del archivo)

histograma(const char nombre[])

No recibe parámetros y trabaja directamente sobre el vector interno secuencias previamente cargado. Como salida, imprime en consola la cantidad de secuencias en memoria y un resumen de cada una. Para cada 8 secuencia, recorre línea por línea y carácter por carácter de su contenido; si el símbolo leído no es un guion -, lo cuenta como una base válida, pero si encuentra un guion marca la secuencia como incompleta. Al final del recorrido de cada secuencia muestra su nombre y la cantidad de bases encontradas. Si no se hallaron guiones, indica que la secuencia "contiene X bases"; si sí había guiones, aclara que "contiene al menos X bases". De esta manera, la función transforma las secuencias almacenadas en memoria en un reporte textual que distingue entre secuencias completas e incompletas y entrega al usuario la información esencial de forma clara. Complejidad O(n) (debe revisar cada letra de la secuencia)

esSubsecuencia(const char subsecuencia[])

Recibe como entrada una cadena de caracteres que representa la subsecuencia a buscar. Su objetivo es recorrer todas las secuencias que estén en memoria y contar cuántas veces aparece esa subsecuencia en su contenido. Si no hay secuencias cargadas, la salida inmediata es el mensaje "No hay secuencias cargadas en memoria". Cuando existen secuencias, la función recorre cada línea de cada secuencia y usa strstr para localizar la primera ocurrencia de la subsecuencia dentro de la cadena. Luego, en un

bucle, avanza el puntero carácter por carácter para detectar posibles coincidencias mas adelante en una misma linea, aumentando un contador cada vez que encuentra una nueva aparición. Al finalizar, si el contador está en cero, imprime que la subsecuencia no existe en memoria; en caso contrario, muestra cuántas veces se repite en total. Complejidad $O(n^2)$ (por cada carácter, debe revisar el resto de caracteres de la misma linea)

enmascarar(const char subsecuencia[])

Recibe como entrada una subsecuencia de caracteres que se quiere ocultar dentro de las secuencias cargadas. Su propósito es recorrer todas las secuencias en memoria, buscar cada ocurrencia de esa subsecuencia y reemplazar sus caracteres por la letra 'X'. Si no hay secuencias almacenadas, la función detiene su ejecución de inmediato mostrando el mensaje “No hay secuencias cargadas en memoria”. Cuando existen secuencias, la función revisa línea por línea y usa strstr para localizar coincidencias. Cada vez que encuentra una, reemplaza carácter por carácter de la subsecuencia con 'X', incrementa un contador y continúa avanzando para detectar más apariciones. Al final, si el contador queda en cero, informa que la subsecuencia no se encontró; en caso contrario, indica cuántas veces fue enmascarada dentro de todas las secuencias. Complejidad $O(n^2)$ (por cada carácter, debe revisar el resto de caracteres de la misma linea)

guardar(const char nombreArchivo[])

Recibe como entrada el nombre de un archivo de salida donde se desea almacenar el contenido de las secuencias en memoria. Si no existen secuencias cargadas, lo primero que hace es notificarlo con el mensaje “No hay secuencias cargadas en memoria” y cerrar

el flujo de salida sin escribir nada. Si el archivo no puede abrirse correctamente, muestra un mensaje de error indicando que hubo un problema guardando en la ruta especificada. Cuando la apertura es exitosa y existen secuencias, la función recorre el vector interno y para cada elemento imprime en el archivo la cabecera con el carácter > seguido del nombre de la secuencia y luego cada línea de su contenido, manteniendo la justificación (anchura) original. Una vez completada la escritura, cierra el archivo y confirma al usuario con el mensaje “Las secuencias han sido guardadas en [nombreArchivo]”. Complejidad O(n) (lee todos los caracteres de todas las secuencias en el archivo).

Operaciones TAD CompararFrecuencia

operator()(const NodoHuffman* a, const NodoHuffman* b)

es un operador sobrecargado que actúa como función de comparación para ordenar nodos dentro de una cola de prioridad utilizada en la construcción del árbol de Huffman. Recibe dos punteros a nodos de y devuelve un valor booleano que indica si el primer nodo debe tener menor prioridad que el segundo dentro de la estructura. En este caso, la comparación devuelve true cuando el nodo a tiene mayor frecuencia que el nodo b. Este criterio está diseñado intencionalmente para que los nodos de menor frecuencia queden en el tope de la cola de prioridad, ya que son los primeros que deben combinarse en el proceso de construcción del árbol de Huffman. De esta manera, el operador permite que la estructura mantenga automáticamente un orden ascendente basado en las frecuencias de los nodos. Complejidad O(1), ya que solo compara dos enteros.

Operaciones TAD NodoHuffman

buscarHoja(NodoHuffman *nodoActual, std::string nodoBuscar)

buscarHoja es una función recursiva que permite obtener el código Huffman asociado a un símbolo específico dentro del árbol de Huffman. Recibe como parámetros un puntero al nodo actual desde donde comienza la búsqueda (nodoActual) y el carácter objetivo (nombreBuscar) cuyo código binario se desea encontrar. La función navega el árbol de forma profundidad primero (DFS), explorando primero el hijo izquierdo y luego el derecho. En cada rama agrega un '0' si baja a la izquierda o un '1' si baja a la derecha, construyendo así el código Huffman correspondiente al recorrido desde la raíz hasta la hoja buscada. Si encuentra el nodo hoja que contiene el símbolo buscado, retorna una cadena vacía que se usará en las llamadas anteriores para formar el código final. Si al finalizar una rama no encuentra el símbolo buscado, retorna 'F' como marca de fallo que indica que debe seguir buscando en otra rama. De esta forma, la función construye de forma progresiva y recursiva el código Huffman para cada símbolo. Complejidad O(n) en el peor caso, donde *n* es la cantidad de nodos del árbol, ya que en el peor escenario debe recorrer todo el árbol para encontrar el símbolo.

Operaciones TAD ArbolDeCodificacionHuffman

comprimirSecuencias(std::string nombrefabin, ListaSecuencias secuenciasEnMemoria)

comprimirSecuencias es el núcleo del proceso de codificación Huffman en el proyecto. Esta función recorre todas las secuencias genéticas cargadas en memoria,

calcula la frecuencia de cada símbolo presente, construye el árbol de Huffman correspondiente y genera la tabla de códigos binarios necesaria para comprimir la información. Finalmente, empaqueta los datos comprimidos siguiendo la estructura del archivo .fabin exigida por el proyecto y los escribe en un archivo binario. Este procedimiento reduce significativamente el espacio utilizado por las secuencias originales, permitiendo almacenarlas de forma compacta y eficiente.

La función inicia recorriendo todas las secuencias almacenadas en memoria para identificar cada símbolo genético, contando cuántas veces aparece cada uno. Para esto se utiliza un vector tablaDeHuffmanDesordenada, donde cada posición almacena un carácter único y su frecuencia asociada. Si un símbolo ya ha sido registrado antes, se incrementa su contador; si no, se agrega como nuevo. Complejidad O(m·k), donde *m* es la cantidad total de caracteres y *k* el número de símbolos únicos encontrados.

Con la tabla de frecuencias lista, cada símbolo se transforma en un nodo de Huffman (NodoHuffman) y se inserta en una cola de prioridad (priority_queue) utilizando el comparador CompararFrecuencia. Esta estructura garantiza que siempre se pueda acceder rápidamente a los nodos de menor frecuencia, que son esenciales para construir el árbol de manera óptima. Complejidad O(n log n), siendo *n* la cantidad de símbolos distintos.

A continuación, se construye el árbol combinando iterativamente los dos nodos de menor frecuencia: se extraen de la cola, se unen en un nuevo nodo padre cuya frecuencia es la suma de ambas, y este nuevo nodo se vuelve a insertar. Este proceso continúa hasta

que solo queda un nodo en la cola: la raíz del árbol de Huffman..Complejidad O(n log n) por operaciones repetidas de extracción e inserción.

Para cada símbolo registrado, se genera su código binario único a partir del árbol. Esto se hace recorriendo el árbol desde la raíz hasta cada hoja usando la función recursiva buscarHoja, que asigna '0' cuando baja por la izquierda y '1' cuando baja por la derecha. Los códigos resultantes se almacenan en dos vectores paralelos: caracteresHuffman y codigosHuffman. Complejidad O(n·h), donde *h* es la altura del árbol.

Con la tabla de códigos lista, se reemplaza cada letra de cada secuencia original por su correspondiente código Huffman. Así se forma una representación comprimida en forma de cadena de bits. Todas las secuencias codificadas se almacenan en el vector secuenciasCodificadas. Complejidad O(m·k), similar al recorrido anterior.

Finalmente, se crea el archivo binario con la estructura definida por las reglas del proyecto. Se escribe:

1. Número de símbolos distintos (n).
2. Cada símbolo y su frecuencia (ci, fi).
3. Número de secuencias (ns).
4. Nombre de cada secuencia (li y caracteres).
5. Longitud real de cada secuencia (wi).
6. Ancho por línea (xi).
7. Secuencias comprimidas en bits empaquetadas en bytes.

La cadena binaria de cada secuencia se rellena con ceros hasta ser múltiplo de 8 bits para almacenarla correctamente byte a byte. Complejidad O(m**)**.

Ejemplo Práctico.

Se realizará un ejemplo de cómo se crea el árbol para la cadena.

Cadena.

>SEQ1

AABAC

Conteo de Secuencias.

A, 3.

B, 1

C, 1

Cola de Prioridad.

Tabla 1, cola de prioridad ejemplo primer paso

Símbolo	Frecuencia
B	1
C	1
A	3

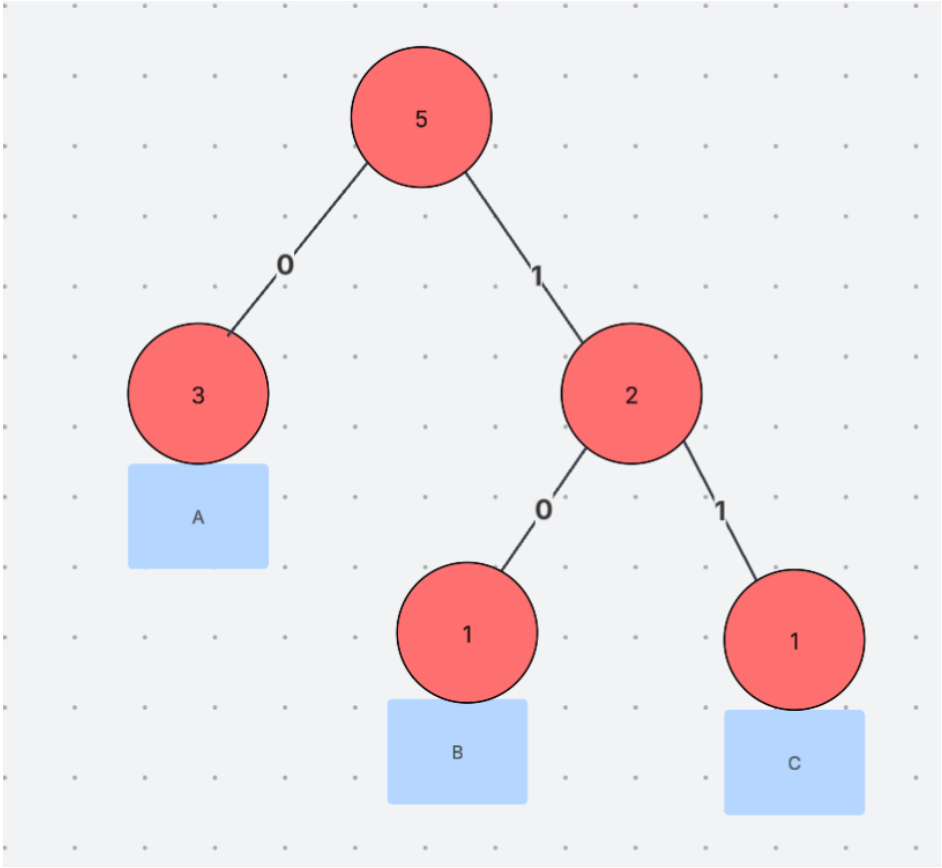
Se expulsan los primeros dos símbolos, se crean sus nodos, se combina su frecuencia y se vuelven a colocar en la cola

Tabla 2, cola de prioridad ejemplo segundo paso

Simbolo	Frecuencia
BC	2
A	3

Finalmente, se expulsan los dos últimos símbolos, se crean sus nodos y su combinación al ser los últimos sería la raíz, se colocan los respectivos códigos, o para la izquierda y 1 para la derecha.

Gráfico 5, árbol ejemplo



Siendo así, el código de huffman final sería, A: 0, B: 10, C: 11

descomprimirSecuencias(std::string nombrefabin, ListaSecuencias/ secuenciasEnMemoria)

Esta función lee un archivo binario previamente comprimido (fabin), reconstruye el árbol de Huffman necesario para decodificar las secuencias, interpreta los datos binarios almacenados y genera las secuencias originales en memoria con su estructura de líneas y ancho original. Este procedimiento permite recuperar la información comprimida de manera completa y fiel a su formato original, lista para su uso en el proyecto.

La función inicia liberando la memoria asociada a cualquier árbol Huffman existente en memoria (raiz), para evitar fugas y conflictos al construir uno nuevo a partir del archivo comprimido.

A continuación, se abre el archivo binario usando ifstream en modo binario. Si el archivo no puede abrirse, la función emite un mensaje de error y termina. Una vez abierto, se lee el primer dato: un número de 2 bytes (uint16_t n) que indica la cantidad de símbolos distintos presentes en las secuencias.

Luego, por cada símbolo se lee su carácter y su frecuencia (1 byte para el carácter, 8 bytes para la frecuencia) y se almacenan en el vector tablaDeHuffmanDesordenada. Esta estructura temporal permite reconstruir posteriormente el árbol de Huffman. Complejidad O(n), donde n es la cantidad de símbolos distintos.

Cada símbolo en tablaDeHuffmanDesordenada se convierte en un nodo Huffman (NodoHuffman) y se inserta en una cola de prioridad (priority_queue) usando el

comparador CompararFrecuencia, de manera que los nodos de menor frecuencia puedan combinarse primero, siguiendo la lógica de Huffman. Complejidad O(n log n).

A continuación, se construye el árbol de Huffman combinando iterativamente los dos nodos de menor frecuencia: se extraen de la cola, se unen en un nuevo nodo padre cuya frecuencia es la suma de ambos, y se inserta nuevamente en la cola. El proceso continúa hasta que solo queda un nodo, que se asigna como raíz del árbol (raiz). Complejidad O(n log n).

Con el árbol listo, la función genera los códigos binarios para cada símbolo mediante la función recursiva buscarHoja, que recorre el árbol desde la raíz hasta cada hoja asignando '0' para la rama izquierda y '1' para la derecha. Los códigos resultantes se almacenan en los vectores paralelos caracteresHuffman y codigosHuffman para decodificación directa. Complejidad O(n·h), donde h es la altura del árbol.

Seguidamente, se lee la cantidad de secuencias almacenadas en el archivo (uint32_t ns) y se limpia la memoria de cualquier secuencia previamente cargada en secuenciasEnMemoria. Para cada secuencia, se leen primero la longitud del nombre (uint16_t li) y luego el nombre como tal, asegurando que termine en '\0'. Esto permite reconstruir los identificadores de las secuencias.

Luego, para cada secuencia, se leen la longitud real (wi) y el ancho por línea (xi). La decodificación de los datos comprimidos se realiza byte a byte: cada byte se transforma en bits que se acumulan en codigoActual hasta que coinciden con un código Huffman registrado. Al coincidir, se agrega el símbolo correspondiente a

secuenciaDecodificada y se reinicia codigoActual. Este proceso se repite hasta reconstruir la secuencia completa de longitud wi. Complejidad O(m·k), donde m es la longitud total de las secuencias y k la cantidad de símbolos distintos.

Finalmente, la secuencia decodificada se divide en líneas de ancho xi y se almacena en secuenciasEnMemoria, reservando memoria dinámica para cada línea (char*) y preservando la estructura original de la secuencia. Este paso garantiza que la representación en memoria sea idéntica a la original antes de la compresión. Complejidad O(m).

Al terminar, el archivo binario se cierra y todas las secuencias originales quedan disponibles en memoria, completamente reconstruidas y listas para su uso en el proyecto.

Ejemplo Practico.

Tabla 2, codificación fabin

Campo codificado	Descripción	Valor
n	Cantidad de bases diferentes	2
c1	Símbolo 1	A
f1	Frecuencia del símbolo 1	2
c2	Simbolo 2	B
f2	Frecuencia del símbolo 2	2
ns	Cantidad de secuencias	1

l1	Tamaño del nombre de la secuencia 1	2
s11s12	Nombre de la secuencia 1	‘S’, ‘1’
w1	Longitud de la secuencia 1	4
x1	Ancho de línea	4
binary_code1	Bits codificados	0 1 1 0

Tabla 3, tabla de frecuencia construida

A	2
B	2

Grafico 6, árbol generado

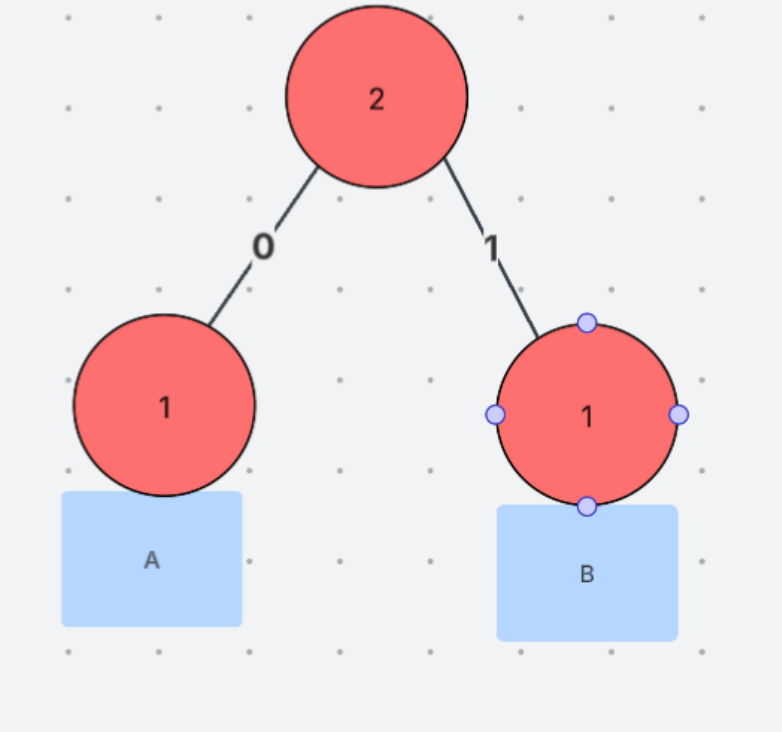


Tabla 4, códigos de los caracteres

A	0
B	1

Gráfico 7, decodificación de secuencia