PONTIFICIA UNIVERSIDAD JAVERIANA

ESTRUCTURA DE DATOS

TALLER#2

BUSCARSUB CADENAS

PRESENTA:

Brandon Garcia Rodriguez

PROFESOR

John Jairo Corredor Franco

BOGOTA D.C

2025

- **Resumen**

El procesamiento y análisis de texto es una necesidad recurrente en múltiples áreas, como la lingüística computacional y la minería de datos. Uno de los problemas comunes es la búsqueda de patrones en grandes volúmenes de texto de manera eficiente. Este proyecto aborda el problema de identificar palabras que cumplen con ciertos criterios en un archivo de texto, permitiendo buscar palabras que inicien con una subcadena específica, que la contengan en cualquier parte o que contengan la versión invertida de la subcadena. Se propone una solución basada en la estructura de datos "ArchivoTexto", que almacena las palabras organizadas por línea y permite consultas eficientes utilizando listas, colas y pilas. Los resultados obtenidos muestran que esta estructura facilita la búsqueda en distintas condiciones y optimiza la recuperación de información en archivos de texto.

- **Introducción**

El análisis de texto es una tarea fundamental en múltiples disciplinas, desde la inteligencia artificial hasta la recuperación de información. La capacidad de buscar palabras con características específicas en documentos grandes puede ser una tarea costosa si no se emplean estructuras de datos adecuadas. En este proyecto se implementa un sistema que permite la búsqueda eficiente de palabras en un archivo de texto basándose en tres criterios:

1. Palabras que comienzan con una subcadena.
2. Palabras que contienen la subcadena en cualquier parte.
3. Palabras que contienen la versión invertida de la subcadena.

Para lograrlo, se diseña una estructura de datos eficiente que almacena las palabras por línea y utiliza diferentes estructuras para facilitar las consultas. Esta solución permite realizar búsquedas en archivos grandes de forma rápida y estructurada.

- **Diseño**

TAD ArchivoTexto

Atributos:

- vector<vector<string>> lineasTexto: Almacena las líneas del archivo como listas de palabras.

Operaciones:

1. void AgregarListaPals(const vector<string>& n_lista): Agrega una lista de palabras al almacenamiento.
2. vector<vector<string>> ObtenerListaLineas() const: Devuelve todas las líneas almacenadas.
3. unsigned int ObtenerNumLineas() const: Retorna el número de líneas almacenadas.
4. list<Palabra> BuscarPrincipio(const string& subcadena): Retorna una lista de palabras que comienzan con la subcadena dada.
5. queue<Palabra> BuscarContiene(const string& subcadena): Retorna una cola de palabras que contienen la subcadena en cualquier posición.
6. stack<Palabra> BuscarInvertida(const string& subcadena): Retorna una pila de palabras que contienen la subcadena invertida.

Condiciones:

- Las líneas de texto se almacenan en orden de aparición en el archivo.
- Las palabras no se modifican, solo se almacenan y se consultan.
- Cada búsqueda debe devolver las palabras con la línea en la que aparecen.

- **Análisis**

El código implementa la funcionalidad de búsqueda de palabras en un archivo de texto utilizando una estructura basada en vector<vector<string>> para almacenar las líneas. Se presentan varios aspectos clave en su diseño y ejecución:

1. **Estructuras de Datos Utilizadas:**
    a. vector<vector<string>>: Estructura principal para almacenar las palabras organizadas por líneas.
    b. list<Palabra>: Se usa para almacenar los resultados de búsqueda de palabras que comienzan con una subcadena, ya que la lista permite iteraciones eficientes.
    c. queue<Palabra>: Se usa para almacenar palabras que contienen la subcadena en cualquier posición, lo que facilita la recuperación en orden de inserción.
    d. stack<Palabra>: Se usa para almacenar palabras que contienen la versión invertida de la subcadena, lo que permite una recuperación en orden LIFO (último en entrar, primero en salir).
2. **Métodos de Búsqueda:**
    a. BuscarPrincipio: Recorre las líneas del archivo y busca palabras cuyo prefijo coincida con la subcadena dada.

b. BuscarContiene: Recorre las líneas y busca palabras que contengan la subcadena en cualquier posición.
c. BuscarInvertida: Invierte la subcadena de búsqueda y luego recorre las palabras almacenadas para encontrar coincidencias.

3. **Manejo de Archivos:**
   a. Se abre el archivo de texto especificado como argumento en main().
   b. Se lee la cantidad n (número de líneas) y la subcadena a buscar.
   c. Se divide cada línea en palabras usando dividirPalabras() y se almacena en ArchivoTexto.
4. **Eficiencia y Complejidad:**
   a. La búsqueda en cada línea tiene una complejidad de O(m), donde m es el número de palabras en la línea.
   b. La búsqueda en todas las líneas tiene una complejidad de O(n*m), donde n es el número de líneas.
   c. El uso de listas, colas y pilas permite organizar los resultados según el tipo de búsqueda requerida.

- **Plan de pruebas**

```
[brandon_garcia@localhost ESTRUCTURA DE DATOS]$ g++ -o buscar_subcadenas  main.cpp
[brandon_garcia@localhost ESTRUCTURA DE DATOS]$ ./buscar_subcadenas entrada4.txt
Palabras que comienzan con 'oda': 0
Palabras que contienen 'oda': 3
podadera. (Línea 4)
acomodada (Línea 13)
toda (Línea 15)
Palabras que contienen la versión invertida de 'oda': 6
criado. (Línea 18)
encantado, (Línea 17)
graduado (Línea 13)
curado, (Línea 12)
madrugador (Línea 6)
sabados, (Línea 2)
[brandon_garcia@localhost ESTRUCTURA DE DATOS]$
```

- **Conclusión**

El código implementa un sistema eficiente para la búsqueda de palabras en un archivo de texto utilizando estructuras de datos adecuadas. Cada tipo de búsqueda ofrece un acceso organizado a los resultados: list para iteraciones ordenadas, queue para accesos en orden de aparición y stack para recuperación en orden inverso. Esto permite realizar análisis textuales en diferentes escenarios con una estructura de consultas flexible.

Además, la división del archivo en palabras facilita un procesamiento estructurado, y el uso de vector<vector<string>> proporciona una organización clara de los datos. Sin embargo, dado que cada búsqueda recorre todas las líneas y palabras almacenadas, la eficiencia podría mejorarse con índices u optimizaciones adicionales.

En términos prácticos, este código puede ser aplicado en análisis de texto, recuperación de información y filtrado de datos en archivos extensos. Su diseño modular permite adaptarlo a otras aplicaciones más avanzadas, como motores de búsqueda o sistemas de procesamiento de lenguaje natural. Optimizar su desempeño mediante técnicas de indexación sería un siguiente paso clave para mejorar su rendimiento en textos de gran escala.