Universidad Austral de Chile

Instituto de Informática

Tarea 2

INFO088 – Taller de Ingeniería: Estructuras de Datos y Algoritmos

Integrantes:

Samantha Espinoza, Lirayen Martínez, Catalina Mora, Franco Navarrete

Profesores: Mauricio Ruiz-Tagle y Héctor Ferrada

Fecha de entrega:

11 de julio de 2025

Resumen

En este trabajo realizamos, la evaluación de un sistema de archivos simulado mediante un árbol k-ario en C++. La estructura de datos representa la jerarquía de directorios y archivos, manteniendo los nodos hijos (entradas de un directorio) en un orden lexicográfico constante. Se implementaron operaciones fundamentales de búsqueda, inserción y eliminación, las cuales garantizan la consistencia entre la representación en memoria y el sistema de archivos físico del sistema operativo. El objetivo es analizar la eficiencia de la estructura y comparar su rendimiento práctico con el análisis de complejidad teórica.

1. Introducción

La gestión de archivos constituye una función central en todo sistema operativo moderno, permitiendo organizar, almacenar y acceder de manera eficiente a grandes volúmenes de información. Un ejemplo es el sistema de archivos de Linux, el cual organiza los datos en una jerarquía de árbol, comenzando desde un directorio raíz (`/`). En este esquema, cada archivo y directorio posee una ruta única que describe su ubicación dentro de la estructura, lo que facilita las operaciones de búsqueda, inserción y eliminación de datos.

En el contexto de este proyecto, se explora el uso de un árbol k-ario como estructura de datos subyacente para simular un sistema de archivos jerárquico. Un árbol k-ario es una estructura en la que cada nodo puede tener hasta k hijos; en este caso, los nodos internos representan directorios, mientras que los nodos hoja representan archivos.

- Un nodo interno representa un directorio.
- Un nodo hoja (sin hijos) representa un archivo.

Una decisión clave en la implementación es que los hijos de cada nodo se almacenan en un vector ordenado lexicográficamente, permitiendo así realizar búsquedas eficientes mediante búsqueda binaria. Las operaciones sobre el sistema utilizan rutas relativas (por ejemplo, `documentos/reporte.txt`), que luego son transformadas en rutas absolutas para interactuar con el sistema de archivos real, asegurando que la estructura en memoria refleje el estado del disco.

Este enfoque simula el funcionamiento real de un sistema de archivos, y permite analizar experimentalmente el rendimiento de operaciones fundamentales, comprobando cómo la elección de la estructura de datos influye en la eficiencia de los procesos computacionales.

2. Metodología

Los algoritmos para las operaciones de **búsqueda**, **inserción** y **eliminación** fueron diseñados con el objetivo de ser eficientes y de mantener la integridad de los datos tanto en la estructura del árbol en memoria como en el sistema de archivos físico. La estrategia principal consiste en reflejar fielmente el estado del sistema de archivos real en la estructura en memoria, asegurando que nunca existan discrepancias entre ambos. Este enfoque es fundamental en sistemas modernos, donde la consistencia y la eficiencia son requisitos esenciales.

2.1. Búsqueda

El algoritmo de búsqueda (`buscar`) permite localizar un nodo (ya sea archivo o directorio) a partir de su ruta relativa.

1. **Descomposición de la ruta:** La ruta de entrada, por ejemplo, (`dir1/subdir/archivo.txt`), se fragmenta en sus componentes (`["dir1","subdir","archivo.txt"]`), lo que facilita el recorrido secuencial de la jerarquía del árbol.

2. **Recorrido desde la raíz:** La búsqueda comienza en el nodo raíz del árbol y, para cada componente, se realiza una búsqueda binaria (`busquedaBinaria`) sobre el vector de hijos del nodo actual.

3. **Verificación de existencia:** Si el componente es encontrado, el algoritmo avanza al siguiente nivel; en caso contrario, la búsqueda concluye informando que la ruta no existe.

Justificación: El mantenimiento de los vectores de hijos ordenados lexicográficamente en cada nodo permite aprovechar la búsqueda binaria, la cual reduce el tiempo de búsqueda en directorios grandes de $O(n)$ a $O(\log n)$ por cada nivel de la ruta. Esta mejora es importante, cuando los directorios contienen una gran cantidad de elementos.

2.2. Inserción

El algoritmo de inserción (`insertar`) añade un nuevo archivo o directorio al árbol y, en sincronía, al sistema de archivos físico.

1. **Localización del nodo padre:** Se utiliza el mismo mecanismo de búsqueda para ubicar el directorio padre donde se insertará el nuevo elemento. Si el padre no existe, la operación se cancela.

2. **Verificación de duplicados:** Mediante búsqueda binaria, se verifica que no exista otro elemento con el mismo nombre en el directorio padre.

3. **Consistencia primero:** Se intenta crear primero el archivo o directorio en el sistema de archivos físico (`crearArchivoSistema` o `crearDirectorioSistema`). Si la operación falla, no se realizan cambios en la estructura en memoria.

4. **Actualización del árbol:** Solo tras una creación física exitosa, se instancia el nuevo nodo y se inserta en el vector de hijos del nodo padre, manteniendo el orden lexicográfico a través de la función `insertarOrdenado`. Este paso puede implicar el desplazamiento de otros elementos en el vector.

Justificación: Al priorizar la creación en el sistema de archivos físico antes de modificar el árbol, se garantiza que la estructura en memoria nunca represente un estado que no exista físicamente. Este enfoque previene inconsistencias y refleja prácticas recomendadas en la implementación.

2.3. Eliminación

El algoritmo de eliminación (`eliminar`) permite borrar archivos o directorios existentes, manteniendo la coherencia entre la estructura en memoria y el sistema de archivos.

1. **Localización del nodo objetivo y su padre:** Se utiliza el mismo método de búsqueda para ubicar tanto el nodo a eliminar como su nodo padre.

2. **Consistencia primero:** Se intenta eliminar primero el archivo o directorio (y todo su subárbol, en caso de directorio) del sistema de archivos físico (`eliminarDelSistema`). Si la operación física falla, el proceso se detiene sin alterar la estructura en memoria.

3. **Actualización del árbol y liberación de memoria:** Tras el éxito en la eliminación física, el puntero correspondiente se elimina del vector de hijos del nodo padre. Finalmente, se libera la memoria asociada al nodo eliminado y, si corresponde, a todo su subárbol (`eliminarSubarbol`), previniendo *memory leaks*.

3. Análisis asintótico

A continuación, se describen los algoritmos implementados y su análisis teórico utilizando notación O:

3.1. Búsqueda en árbol k-ario con búsqueda binaria

La operación de búsqueda consiste en localizar un nodo (archivo o directorio) a partir de su ruta relativa. El procedimiento es el siguiente:

- Se divide la ruta de entrada en sus componentes, por ejemplo: `dir1/subdir/archivo.txt` → `["dir1","subdir","archivo.txt"]`.
- Se comienza en el nodo raíz del árbol.
- Para cada componente, se realiza una búsqueda binaria en el vector de hijos del nodo actual.
- Si se encuentra el componente, se continúa al siguiente nivel. Si en algún punto no se encuentra, la búsqueda termina y se retorna que la ruta no existe.

Pseudocódigo:

```
Función buscar(nodo, ruta)
    componentes ← dividirRuta(ruta)
    actual ← nodo
    Para cada nombre en componentes hacer
        índice ← busquedaBinaria(actual.hijos, nombre)
        Si índice = -1 entonces
            retornar no encontrado
```

```
        actual ← actual.hijos[índice]
    FinPara
    retornar actual
Fin
```

La complejidad temporal de esta operación es $O(d \log k)$, donde d es la profundidad de la ruta y k es el número máximo de hijos en un nodo. El uso de búsqueda binaria reduce significativamente el tiempo en comparación con una búsqueda lineal, especialmente en directorios con muchos archivos o subdirectorios.

3.2. Inserción en árbol k-ario

La inserción de un nuevo archivo o directorio requiere ubicar correctamente el directorio padre y mantener el orden lexicográfico del vector de hijos. El procedimiento es el siguiente:

- Se divide la ruta de destino en sus componentes y se busca el directorio padre utilizando el mismo método de búsqueda anterior.
- Si el padre no existe, la operación falla.
- Se verifica mediante búsqueda binaria que no exista ya un elemento con el mismo nombre en el directorio padre.
- Si la verificación es exitosa, primero se crea el archivo/directorio en el sistema de archivos físico.
- Luego se crea un nuevo nodo y se inserta en el vector de hijos en la posición correspondiente, manteniendo el orden.

Pseudocódigo:

```
Función insertar(nodo, ruta, nuevoElemento)
    componentes ← dividirRuta(ruta)
    actual ← nodo
    Para i ← 0 hasta longitud(componentes) - 2 hacer
        índice ← busquedaBinaria(actual.hijos, componentes[i])
        Si índice = -1 entonces
            retornar no existe directorio padre
        actual ← actual.hijos[índice]
    FinPara
    índice ← busquedaBinaria(actual.hijos, nombreNuevo)
    Si índice  -1 entonces
        retornar ya existe
    crearArchivoSistema(nuevoElemento)
    insertarOrdenado(actual.hijos, nuevoElemento)
    retornar éxito
Fin
```

La complejidad de inserción es $O(d \log k + k)$, ya que primero se realiza una búsqueda binaria en cada nivel ($O(d \log k)$) y luego la inserción en el vector ordenado puede requerir desplazar elementos ($O(k)$).

3.3. Eliminación en árbol k-ario

La eliminación de un archivo o directorio sigue un proceso similar a la búsqueda para localizar el nodo y su padre:

- Se localiza el nodo objetivo y su padre utilizando búsqueda binaria.
- Se intenta eliminar el archivo/directorio en el sistema de archivos físico. Si la operación falla, no se modifica el árbol en memoria.
- Si la eliminación física es exitosa, se elimina el puntero del vector de hijos y se libera la memoria asociada al nodo y su subárbol.

Pseudocódigo:

```
Función eliminar(nodo, ruta)
    componentes ← dividirRuta(ruta)
    actual ← nodo
    Para i ← 0 hasta longitud(componentes) - 2 hacer
        índice ← busquedaBinaria(actual.hijos, componentes[i])
        Si índice = -1 entonces
            retornar no encontrado
        actual ← actual.hijos[índice]
    FinPara
    índice ← busquedaBinaria(actual.hijos, nombreAEliminar)
    Si índice = -1 entonces
        retornar no encontrado
    eliminarDelSistema(nombreAEliminar)
    eliminar actual.hijos[índice] y liberar memoria
    retornar éxito
Fin
```

La complejidad es $O(d \log k + k)$: búsqueda binaria en cada nivel y posible desplazamiento de elementos en el vector de hijos al eliminar el nodo. La eliminación recursiva en memoria asegura que no haya fugas de memoria.

4. Conclusiones

La implementación de un árbol k-ario con hijos ordenados lexicográficamente en C++ resulta ser una opción robusta para modelar la jerarquía de un sistema de archivos de forma eficiente. A partir del trabajo en este proyecto conclusiones generales:

- **Búsqueda.** La operación de localización de rutas tiene complejidad $O(H \log K)$, donde H es la profundidad del árbol y K el número máximo de hijos por nodo. Esto garantiza escalabilidad logarítmica incluso en jerarquías muy profundas o con muchos elementos en cada nivel.
- **Inserción y eliminación.** Ambas operaciones combinan la búsqueda del punto de inserción (o del nodo a eliminar) —con costo $O(H \log K)$— y la gestión de un vector ordenado de hasta K hijos, lo que añade un término $O(K)$. En conjunto, su complejidad es $O(H \log K + K)$, un balance entre rapidez de consulta y coste de mantenimiento del orden.
- **Comparación con lista no ordenada.** En una estructura que mantuviera los hijos en una lista sin ordenar, la búsqueda pasaría a costar $O(H \cdot K)$, lo cual resulta ineficiente cuando los directorios contienen muchos elementos. Aunque la inserción en la lista no ordenada podría realizarse en $O(1)$ (tras comprobar duplicados en $O(H \cdot K)$), el incremento en el tiempo de búsqueda —la operación más frecuente en sistemas de archivos— no compensa la ganancia en actualización.

En resumen, la organización ordenada de los hijos y el uso de búsqueda binaria constituyen un pilar fundamental para optimizar el rendimiento de un sistema de archivos simulado.

Referencias

[1] "How do I create a library in C++," GeeksForGeeks. [Online]. Available: `https://www.geeksforgeeks.org/how-do-i-create-a-library-in-cpp/`

[2] "File System Library in C++ 17," GeeksForGeeks. [Online]. Available: `https://www.geeksforgeeks.org/file-system-library-in-cpp-17/`

[3] "Chrono in C++," GeeksForGeeks. [Online]. Available: `https://www.geeksforgeeks.org/chrono-in-c/`

[4] "Measure Execution Time of a Function in C++," GeeksForGeeks. [Online]. Available: `https://www.geeksforgeeks.org/measure-execution-time-function-cpp/`

[5] "Lower_bound in C++," GeeksForGeeks. [Online]. Available: `https://www.geeksforgeeks.org/lower_bound-in-cpp/`

[6] "File Handling C++ Classes," GeeksForGeeks. [Online]. Available: `https://www.geeksforgeeks.org/file-handling-c-classes/`

[7] "How to Work with File Handling in C++," GeeksForGeeks. [Online]. Available: `https://www.geeksforgeeks.org/how-to-work-with-file-handling-in-c/`

[8] "Deletion in a Binary Tree," GeeksForGeeks. [Online]. Available: `https://www.geeksforgeeks.org/dsa/deletion-binary-tree/`

[9] "Insertion in a Binary Tree in level order," GeeksForGeeks. [Online]. Available: `https://www.geeksforgeeks.org/dsa/insertion-in-a-binary-tree-in-level-order/`