Tarea 2
INFO088 - Taller de Estructuras de Datos y Algoritmos

Instituto de Informática
Universidad Austral de Chile

Junio 2025

Resumen

Este trabajo tiene como objetivo profundizar en el uso y análisis de estructuras de datos aplicadas a la simulación de un sistema de archivos. Se implementará un sistema basado en un árbol k-ario para almacenar nombres de directorios y archivos, estudiando el rendimiento de operaciones fundamentales (búsqueda, inserción y eliminación) mediante experimentos controlados. Se enfatizará la elección, justificación y análisis asintótico (notación $\mathcal{O}$) de los algoritmos, destacando la importancia de las estructuras de datos en la optimización de procesos computacionales.

Descripción del Problema

Se requiere simular un sistema de archivos jerárquico utilizando un árbol k-ario. Los nodos almacenarán cadenas que representan nombres de directorios (nodos intermedios) o archivos (nodos hoja). Los datos se leerán del sistema de archivos Linux y se almacenarán en la estructura. Posteriormente, se realizarán operaciones de búsqueda, inserción y eliminación evaluando su rendimiento experimentalmente.

Descripción de la Solución

Estructura de Datos

Se utilizará un árbol k-ario definido como:

```
struct NodoArbol {
    string nombre;
    vector<NodoArbol*> hijos; // Vector siempre ordenado lexicográficamente
};
```

- **nombre**: Nombre del directorio/archivo
- **hijos**: Vector de punteros a hijos, se mantiene ordenado alfabéticamente. Si está vacío, el nodo representa un archivo.

Tareas

La solución se organiza en 4 tareas con sus respectivos experimentos:

1. **Carga de Datos**
   - Usar la librería `<filesystem>` para recorrer directorios en Linux.
   - Insertar cada elemento (ruta relativa) en el árbol.
   - Considerar que el recorrido es en *preorden*.
2. **Construcción de la Estructura**
   - Construir el árbol k-ario con los datos leídos.
   - Medir tiempos de creación para diferentes tamaños.
3. **Búsqueda**
   - Implementar búsqueda por rutas relativas (ej: `raiz/directorio1/archivo1`)
   - Búsqueda binaria en cada nivel del árbol
   - Retornar:
     - `0` si existe el archivo.
     - `1` si no existe.
     - `2` si es un directorio.
   - Todo nodo que sea hoja, será considerado archivo.
4. **Eliminación e Inserción**
   - **Eliminación**:
     - Remover nodo en caso de ser archivo
     - Remover el nodo y su subárbol en caso de ser directorio.

- Retorna `true` en éxito, `false` si no existe.

- **Inserción**:
  - Buscar directorio padre. Ej: raiz/directorio1/directorioPadre/archivoInsertar
  - Insertar nuevo nodo manteniendo el orden lexicográfico del vector.
  - Retornar:
    - 0 en éxito.
    - 1 si el archivo ya existe.
    - 2 si no existe ruta (directorioPadre).

Experimentación

Realizar mediciones para:

- Configuraciones:
  - Pequeña: 20k directorios / 200k archivos
  - Mediana: 100k directorios / 1M archivos
  - Grande: 1M directorios / 10M archivos
- Operaciones ($REP = 100,000$):
  1. Tiempo de creación del árbol
  2. REP Búsquedas aleatorias
  3. REP Eliminaciones aleatorias
  4. REP Inserciones de archivos en 2,000 directorios aleatorios. Para cada uno de los archivos a insertar, elegir un directorio al azar e insertarlo.
     - Los nombres de los archivos pueden ser obtenidos desde un array estático.
- Gráficos de lineas comparativos: Una linea para la estructura de su tarea anterior y otra para la actual
  - Tiempo de creación vs. tamaño `n` (cantidad archivos)
  - Tiempo promedio de búsqueda vs. tamaño `n`
  - Tiempo de eliminación vs. tamaño `n`

Nota: Si los resultados no directamente comparables, entonces los gráficos deben presentarse por separado, es decir, uno por linea.

Requisitos de Entrega

Documento [45 %] (Máximo 12 páginas)

- **Abstract**: Resumen breve del documento, estructura de datos, operaciones, experimentos, etc.
- **Introducción**: Contexto y problemática, se deben introducir los conceptos tales como `sistema de archivos linux`, `arbol k-ario`, `rutas relativas`, etc.
- **Metodología**: Algoritmos de inserción, búsqueda, eliminación usados con su justificación.
- **Resultados**: Gráficos y presentación de los resultados obtenidos.
- **Análisis Asintótico**: Notación $\mathcal{O}$ para los algoritmos de búsqueda, inserción y eliminación.
  - Se recomienda el uso de parametros para el análisis. Tales como la altura del árbol y la cantidad máxima de elementos en un directorio.
- **Conclusiones**: Realizar una comparación entre el análisis teórico (análisis asintótico) y los resultados obtenidos experimentalmente. Contrastar el comportamiento de la estructura de datos previamente utilizada con la nueva implementación, explicando las posibles diferencias observadas. Enfatizar el papel que desempeñan las estructuras de datos en el rendimiento del algoritmo y cómo la forma en que se organiza la información influye en los resultados. .

Código Fuente [45 %]

- Estructura modular (usar repositorio guía)
- Archivos principales: `main.cpp`, `experimentacion.cpp`, `arbol.cpp`
- Implementación con `<filesystem>`
- Código documentado y organizado

Correcciones Tarea 1 [10 %]

- Entregar versión revisada de la Tarea 1.

Entrega y Revisión

- **Entrega del documento y código**: Lunes 7 de Julio de 2025 vía Siveduc.
- **Revisión**: Mediante reuniones presenciales la semana del 7 de Julio.
- **Inscripción**: Selección de horario en hoja de cálculo. compartida mediante correo electronico el día viernes 4 de Julio.

A. Notación O grande

- ¿Sabe cómo se calcula la complejidad asintótica del algoritmo quicksort o del de búsqueda binaria?

La notación O grande es utilizada para el cálculo asintótico y es posible gracias a un modelo matemático de comparación llamado *Random Access Machine* (RAM). En este modelo no nos preocupamos por el tamaño de la memoria y asumimos que podemos representar un entero de cualquier tamaño en una palabra de memoria (por ejemplo de 64 bits). Además, es compatible con el modelo y arquitectura de Von Neumann —respetanto que hay un procesador central (CPU) que contiene una ALU, registros de memoria donde se cargan tanto datos como instrucciones de código, y una unidad de control (UC) que posee un contador o cabezal indicando el registro de memoria que contiene la siguiente instrucción a ejecutar. También una memoria principal y un bus o canal de conexión por donde fluyen los datos. Por tanto, el modelo RAM se adapta a las máquinas que hoy en día utilizamos, y asume que solo se pueden cargar valores enteros en las celdas o registros de memoria —al fin de cuentas, todo es discreto en las memorias de nuestros computadores y todo se representa con unos y ceros. Una característica importante que nos permite realizar cálculos asintóticos, es que se asume que las operaciones sencillas tardan una cantidad constante de unidades de tiempo para su ejecución. Entre las operaciones sencillas encontramos: sumar o restar, multiplicar o dividir, inicialización de una variable, comparar dos variables y llamar a una rutina. Por otro lado, los ciclos *for* o *while* o rutinas de cálculo compuestas, como una raíz cuadrada, no son considerados como operaciones sencillas y se les debe realizar un análisis particular para cada caso a fin de estimar la complejidad de su tiempo de ejecución.
Cuando nos referimos al performance de un algoritmo se hace muy necesario tener alguna herramienta que nos permita medir que tan bueno es una

solución algorítmica a un problema bien definido. La notación O[1] —que en ingles es llamada *Big O Notation* [**?**]— permite establecer una medida que da cuenta del rendimiento de un algoritmo en términos asintóticos. La notación O nos permite dar una nomenclatura o simbología a la complejidad de los algoritmos para medir el consumo de un recurso en particular; siendo el recurso más importante y estudiado, el tiempo de ejecución del algoritmo. Sin embargo, no estime como menos importantes otros recursos como el espacio o el consumo energético —más aún hoy en día con todo el tema del *Big Data*. Para estos otros recursos también es posible determinar su complejidad con la notación O si ajustamos adecuadamente el modelo de cómputo.

Tenga en cuenta que no nos estamos refiriendo al tiempo exacto que tardará un algoritmo en ser ejecutado. La notación O nos da las herramientas para un análisis teórico de un algoritmo y que no depende de la implementación en particular de este. El tiempo exacto depende de factores como: el hardware de la máquina en la que se ejecuten los experimentos, la dedicación exclusiva o no de esta en el momento de ejecución, del tamaño de los datos de entrada e incluso de la forma o característica de los datos de entrada en cada ejecución. Note la importancia de que esta medida es solo en términos asintóticos y no absolutos; es decir, nos dice acerca del comportamiento con entradas significativamente grandes. Una justificación de esto es que para escenarios pequeños, muchas veces, no tiene sentido o importancia saber cuál es el costo computacional de la solución. Si, por ejemplo, evaluamos a los algoritmos de ordenamiento, hemos visto en clases que *insertionSort*, que no tiene un tiempo óptimo asintóticamente, es de los mejores para entradas relativamente pequeñas o semi-ordenadas; y para entradas muy grandes, *quickSort* es de los más utilizados, incluso a pesar de ofrecer tiempo cuadrático (al igual que *insertionSort*) en su peor caso —resultado que se aleja muchísimo del óptimo $O(n \log n)$ para algoritmos de ordenamientos basados en comparaciones de sus elementos.

En resumen, *big O notation* es utilizado en ciencias computacionales para describir la complejidad del rendimiento de un algoritmo. Generalmente describe el peor escenario —que es lo que (al menos) se pide en este trabajo; es decir, el máximo trabajo que es proporcional al tiempo que tardará su ejecución en el peor de los casos posibles —esto es cuando la forma como viene la entrada obliga a nuestro algoritmo a trabajar y tardar el mayor tiempo que este puede alcanzar.

[1] Un sencillo video para una primera introducción `https://www.youtube.com/watch?v=dyw0SohyEkw`

Ejemplos clásicos del uso de la notación O

Sea la entrada de un algoritmo de tamaño n elementos; por ejemplo, para un algoritmo de ordenamiento serán los n números a ordenar. Describimos algunos casos populares de Notación O grande:

- **Orden Constante:** $O(1)$.
  Esta expresión indica tiempo asintóticamente constante; es decir, **No depende del tamaño de la entrada** y su ejecución siempre tarda un tiempo que es prácticamente idéntico. Visto desde otro modo, la ejecución del algoritmo no se verá afectado por la cantidad de datos de entrada.
  Por ejemplo, almacenar el entero x en la posición i del arreglo de enteros $A[0 \dots n-1]$, con $0 \leq i < n$; lo cual se realiza simplemente haciento $A[i] = x$. Así, el tiempo que tarda no depende del tamaño del arreglo A y para cualquier valor de n tomará un pequeño tiempo constante muy parecido.

- **Orden Logarítmico:** $O(\log n)$.
  Esta expresión indica tiempo asintóticamente logarítmico; es decir, la ejecución siempre tarda un tiempo proporcional a $\log_b(n)$, para alguna base cualquiera b y para el n de entrada. En términos sencillos el tiempo que tarda el algoritmo se puede establecer como $T(n) = c \cdot \log_b n$, con $c > 0$ una constante y b una base válida. Visto de otro modo, indica que el tiempo $T(n)$ aumenta linealmente, mientras que n sube exponencialmente (con una base $b = 10$); lo que quiere decir que si se tarda 1 unidad de tiempo (t) con una entrada de 10 elementos, se necesitarán 2 t para 100, 3 t para 1000 y así sucesivamente.
  Por ejemplo, suponga que tiene que descender por un árbol binario de búsqueda perfectamente balanceado, buscando un elemento. El tiempo que tarda en el peor caso, cuando debe llegar a una hoja, es proporcional a la altura del árbol y por tanto podemos decir que tarda un tiempo logarítmico; ya que la altura del árbol es $h = \lceil \log_2 n \rceil$; Así el tiempo de esta búsqueda es $O(\log n)$.

- **Orden Lineal:** $O(n)$.
  Esta expresión indica tiempo asintóticamente lineal; es decir, la ejecución siempre tarda un tiempo proporcional al tamaño n de entrada. En términos sencillos el tiempo que tarda el algoritmo se puede establecer como $T(n) = c \cdot n$, para alguna constante $c > 0$. Por ejemplo, buscar el mínimo en un arreglo $A[0 \dots n-1]$ se puede hacer con un simple reco-

rrido de A, o más preciso, con un sencillo recorrido lineal del arreglo; ya que basta atravesarlo una sola vez de esxtremo a extremo.

- **Orden Cuadrático:** $O(n^2)$.
  Esta expresión indica tiempo asintóticamente cuadrático; es decir, la ejecución siempre tarda un tiempo proporcional al valor n^2 . En términos sencillos el tiempo que tarda el algoritmo se puede establecer como $T(n) = c \cdot n^2$, para alguna constante $c > 0$. Por ejemplo, si deseamos buscar cuál es el valor más repetido en un arreglo $A[0 \ldots n-1]$, podríamos diseñar una solución que para cada valor $A[i]$, con $0 \leq i < n$, realice un recorrido lineal del arreglo contabilizando cuantas ocurrencias de $A[i]$ existen en A, y, para cada $x = A[i]$, nos vamos quedando siempre con el x para el cual encontremos una mayor cantidad de ocurrencias.

  A continuación el pseudocódigo de esta solución, este es el formato de pseudocódigo que se espera para las rutinas de los algoritmos de búsqueda, inserción y eliminación que se piden para cada estructura:

- **Input**: un arreglo de n enteros $A[0 \ldots n-1]$
  Output: u, el elemento más repetido en A

```
masRepetido(A, n) {
    u = A[0]
    occ = 0
    for i = 0 to n-1 do {
        count = 0
        x = A[i]
        for j = 0 to n-1 do
            if (A[j] == x) then
                count = count+1
        if (count > occ) then {
            occ = count
            u = x
        }
    }
    return u
}
```

Referencias