Tarea 1
INFO088 - Taller de Estructuras de Datos y Algoritmos

Instituto de Informática, Universidad Austral de Chile.

Marzo 2025

Resumen

El objetivo de este trabajo es 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 vector (array unidimensional) para almacenar nombres de archivo, y se estudiará detalladamente el rendimiento de distintas operaciones (búsqueda, inserción y eliminación) a través de experimentos controlados. Se hará especial énfasis en la elección, justificación y análisis asintótico (notación O) de los algoritmos involucrados, resaltando la importancia de las estructuras de datos en la optimización de procesos computacionales.

Descripción del problema

Se requiere simular un sistema de archivos con un único directorio utilizando un vector simple. El vector almacenará **strings** que representan nombres de archivo con su extensión (ej: "documento.txt"). Los archivos serán leídos desde un directorio real del sistema operativo Linux y almacenados en la estructura de datos. Posteriormente se realizarán operaciones de búsqueda, inserción y eliminación sobre este vector, evaluando su rendimiento a través de experimentos.

Los requisitos específicos son:

- Leer todos los archivos de un directorio especificado.
- Almacenar las rutas completas de los archivos en un vector unidimensional.

- Implementar operaciones de búsqueda para verificar la existencia de archivos.
- Implementar operaciones de eliminación sobre la estructura.
- Ordenar el array en orden lexicográfico.
- Implementar búsqueda eficiente para arreglos ordenados.
- Medir el rendimiento en la construcción de la estructura y en las operaciones de búsqueda, eliminación e inserción.

Descripción de la solución

La solución se divide en las siguientes 5 tareas, cada una acompañada de experimentos de rendimiento:

1. **Carga de datos:**
   - Utilizar la librería `filesystem` para recorrer un directorio base.
   - Por cada archivo encontrado, almacenar la ruta relativa como un `string` en el vector.
2. **Construcción de la Estructura y Experimentos:**
   - Construir el vector dinámico (ED principal) a partir de los datos leídos.
   - Medir el tiempo de creación del vector para distintos tamaños de directorios.
3. **Búsqueda y Experimentos:**
   a) **Búsqueda secuencial:**
      - Implementar y analizar un algoritmo de búsqueda secuencial que retorne el índice de un archivo por su nombre exacto o -1 si no existe.
      - Realizar experimentos con $REP = 100\,000$ búsquedas aleatorias y medir tiempos.

   b) **Búsqueda eficiente:**
      - Ordenar el vector utilizando un algoritmo de ordenamiento (por ejemplo, `std::sort` o un algoritmo visto en clase) y justificar la elección.

- Implementar un algoritmo de búsqueda eficiente que retorne el índice o -1.
- Realizar experimentos de búsqueda sobre el vector ordenado, registrando tiempos y comparandolos con la anterior solucion.

4. **Eliminación e Inserción con Experimentos:**
   - **Eliminación:**
     - Eliminar 100,000 elementos aleatorios utilizando la función `vector.erase()`.
     - Registrar los tiempos de ejecución, considerando el desplazamiento de elementos para evitar huecos en la memoria.
   - **Inserción:**
     - Insertar 100,000 elementos aleatorios de forma ordenada, buscando la posición correcta (usando, por ejemplo, una búsqueda adaptada) e insertando con `vector.insert()`.
     - Medir el tiempo de ejecución de estas operaciones.

Experimentación

Realizar mediciones de tiempo para las operaciones de **creación**, **búsqueda**, **eliminación** e **inserción** de la estructura de datos, considerando las siguientes configuraciones de directorios:

- Directorios pequeños (200,000 archivos)
- Directorios medianos (1,000,000 archivos)
- Directorios grandes (10,000,000 archivos)

Para cada caso se deberá:

1. Medir el tiempo de creación del array.
2. Realizar $REP = 100\,000$ búsquedas aleatorias, registrando los tiempos.
3. Eliminar $REP = 100\,000$ elementos aleatorios del vector, registrando los tiempos.
4. Generar gráficos comparativos de:

- Tiempo de creación vs cantidad de archivos.
- Tiempo promedio de búsqueda vs cantidad de archivos (una linea para el caso secuencial y otra para el eficiente).
- Tiempo de eliminación vs cantidad de archivos.

Requisitos de Entrega

Documento. [50 %] (Máximo 10 páginas, todo incluido)

El informe técnico deberá contener:

- **Abstract** con énfasis en la importancia de las estructuras de datos.
- **Introducción** al problema, con contexto sobre la optimización mediante estructuras de datos.
- **Metodología** de la implementación, agregando los pseudocódigos de los algoritmos de búsqueda secuencial y eficiente, describiendo los algoritmos de eliminación e inserción, y justificando su elección del algoritmo de ordenamiento.
- **Presentación de los resultados** mediante gráficos y descripción de los experimentos realizados.
- **Análisis asintótico:** Se debe incluir un análisis de los dos pseudocódigos de búsqueda (secuencial y eficiente) y describir los algoritmos de eliminación e inserción, indicando sus tiempos asintóticos basados en la descripción dada.
- **Conclusiones** comparando los resultados experimentales con las expectativas teóricas basadas en la notación O.

Código Fuente. [50 %] La implementación en C++ deberá incluir:

- Archivo `main.cpp` que reciba los argumentos y llame a la función de experimentación.
- `experimentacion.cpp`, donde se realicen todas las pruebas de velocidad y medición de memoria.
- `vector.cpp`, donde se implementen las funciones de búsqueda, ordenamiento, creación, eliminación e inserción en el vector.

- Lectura del directorio usando `filesystem`.
- Funciones de búsqueda

Fecha de Entrega

- Lunes 12 de Mayo 2025

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* [1]— 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

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, and Clifford Stein. Cota superior asintótica. `https://es.wikipedia.org/wiki/Cota_superior_asint%C3%B3tica`, 2023. Consultado en mayo de 2024.