Sara Cubero García-Conde

Estrategias de Programación y Estructuras de Datos

Índice:

1. Preguntas teóricas:
    a. Preguntas teóricas de la sección 2.1
    b. Preguntas teóricas de la sección 2.2
    c. Preguntas teóricas de la sección 2.3
    d. Preguntas teóricas de la sección 6
2. Diseño
3. Resultados del estudio empírico de costes

1. Preguntas teóricas

a. Preguntas teóricas de la sección 2.1

1. **¿Qué tipo de secuencia sería la más adecuada para devolver el resultado de la operación listStock(prefix)? ¿Qué consecuencias tendría el uso de otro tipo de secuencias? Razona tu respuesta.**

La secuencia más adecuada para devolver el resultado de la operación listStock(prefix) es una lista ordenada de StockPair. En el caso del código proporcionado, se utiliza la clase List<StockPair>, que es una implementación de la interfaz SequenceIF<StockPair>.

Una lista ordenada es adecuada porque permite una fácil inserción de nuevos elementos en el orden correcto y hace que la búsqueda de elementos sea más eficiente, especialmente si se utiliza una búsqueda binaria. Además, mantener la lista ordenada es útil para realizar la operación de listado de elementos que empiezan por un prefijo determinado, ya que podemos detener la búsqueda una vez que se encuentren todos los elementos que cumplen con el criterio.

El uso de otros tipos de secuencias, como por ejemplo una pila (Stack) o una cola (Queue), podría tener consecuencias negativas en el rendimiento y la eficiencia de las operaciones. Por ejemplo, si se utiliza una pila, la búsqueda de elementos sería ineficiente, ya que tendría que recorrerse toda la pila hasta encontrar el elemento buscado, y la inserción de elementos en orden requeriría desplazar todos los elementos anteriores. En el caso de una cola, la inserción y búsqueda también serían menos eficientes, ya que tendrían que recorrerse todos los elementos hasta encontrar el adecuado. En ambos casos necesitarías una pila o una cola auxiliar para poder ir moviendo los elementos.

Por lo tanto, el uso de una lista ordenada es la opción más adecuada para la operación listStock(prefix) debido a su eficiencia en la búsqueda e inserción de elementos.

b. Preguntas teóricas de la sección 2.2

1. **¿Qué tipo de secuencia sería la adecuada para realizar esta implementación? ¿Por qué? ¿Qué consecuencias tendría el uso de otro tipo de secuencias?**

Creo que una lista (List) como secuencia es la más adecuada para este caso. La razón es que las operaciones requeridas, como insertar elementos al final de la secuencia, iterar sobre los elementos y realizar búsquedas lineales, pueden realizarse eficientemente en una lista.

La consecuencia de usar otro tipo de secuencia, como una cola o una pila, es que algunas operaciones podrían ser menos eficientes. Por ejemplo, si se utilizara una cola, insertar un elemento al final de la secuencia requeriría recorrer todos los elementos previos, lo que resultaría en una mayor complejidad temporal. Además, las búsquedas podrían ser menos eficientes en comparación con una lista.

2. **¿Cuál sería el orden adecuado para almacenar los pares en la secuencia? ¿Por qué? ¿Qué consecuencias tendría almacenarlos sin ningún tipo de orden?**

El orden adecuado para almacenar los pares en la secuencia sería mantenerlos ordenados alfabéticamente por el nombre del producto. Esto se debe a que muchas operaciones en este contexto, como buscar un producto específico o listar productos con un prefijo común, se benefician de tener los elementos ordenados. Al mantener los datos ordenados, se pueden aplicar algoritmos de búsqueda más eficientes, como la búsqueda binaria, en lugar de la búsqueda lineal. Si los pares se almacenan sin ningún tipo de orden, algunas consecuencias serían:

Búsqueda más lenta: La búsqueda de un producto específico requeriría recorrer toda la secuencia en el peor de los casos, lo que resulta en una mayor complejidad temporal (O(n), donde n es el número de elementos en la secuencia).

Ineficiencia en operaciones de listado: Listar productos con un prefijo común sería menos eficiente, ya que se tendría que examinar cada elemento en lugar de detener la búsqueda una vez que se hayan encontrado todos los elementos que coinciden con el prefijo.

Mayor complejidad en la actualización de stock: La actualización del stock de un producto específico podría requerir más tiempo, ya que se necesita buscar el producto en la secuencia antes de actualizarlo.

3. **¿Afectaría el orden a la eficiencia de alguna operación prescrita por StockIF? Razona tu respuesta.**

Sí, el orden de los elementos en la secuencia puede afectar la eficiencia de las operaciones prescritas por StockIF. La razón es la siguiente:

retrieveStock(String p): Si la secuencia está ordenada alfabéticamente por el nombre del producto, la búsqueda de un producto específico se puede realizar utilizando una búsqueda binaria, lo que tendría una complejidad de tiempo O(log n), siendo n el número de elementos en la secuencia. Sin embargo, si la secuencia no está ordenada, se necesita realizar una búsqueda lineal, lo que resulta en una complejidad de tiempo O(n).

updateStock(String p, int u): Cuando los pares están ordenados, la actualización del stock se puede realizar de manera más eficiente, ya que podemos encontrar rápidamente la posición correcta para insertar el nuevo producto o actualizar el stock existente. Si los elementos no están ordenados, sería necesario recorrer toda la secuencia en el peor de los casos para encontrar la posición correcta, lo que resulta en una complejidad de tiempo O(n).

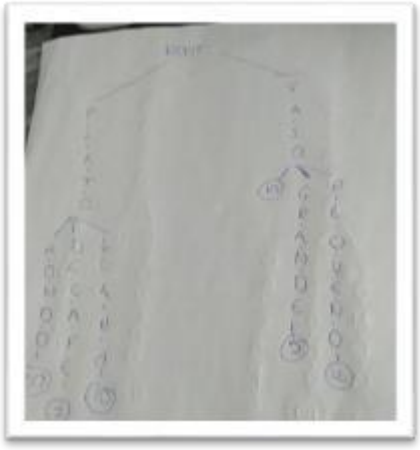
listStock(String prefix): Al tener los pares ordenados alfabéticamente, la generación de listas de productos con un prefijo específico es más eficiente, ya que solo es necesario buscar aquellos elementos con el prefijo dado. Si la secuencia no está ordenada, se debe recorrer toda la secuencia y aplicar un filtro a cada elemento, lo que podría aumentar la complejidad de tiempo.

c. Preguntas teóricas de la sección 2.3

1. **Utilizando papel y lápiz inserta más pares producto-unidades en el árbol del ejemplo. Compara al menos dos formas de colocar los hijos de un nodo y comenta qué ventajas tendría cada una de ellas. Razona tu respuesta.**

Ordenado por orden alfabético:

Se ordenarían en función del carácter. La ventaja es que la búsqueda es muy eficiente, ya que se puede hacer una búsqueda binaria en la lista ordenada de nodos hijo. Además creo que es más intuitivo para el ser humano.



Ordenado por cantidad:

Se ordenan según la cantidad de elementos. Las búsquedas son eficientes y la principal ventaja consiste en encontrar rápidamente los productos con más abastecimiento.

2. **La operación listStock(String prefix) nos indica que la secuencia de elementos de tipo StockPair que devuelve ha de estar ordenada de forma creciente según los productos. ¿Alguna de las opciones de colocar los hijos de un nodo de la pregunta anterior resulta más conveniente para mejorar la eficiencia de esta operación? Razona tu respuesta.**

En un trie, la búsqueda se realiza carácter por carácter, siguiendo las ramas del árbol que corresponden a cada carácter del prefijo. Si los nodos están ordenados alfabéticamente, podemos aprovechar esta estructura para realizar una búsqueda binaria en cada nivel del trie y acceder más rápidamente al nodo que corresponde al siguiente carácter del prefijo.
Esto resulta en una búsqueda más eficiente que si los nodos no estuvieran ordenados.

Además, el proceso de inserción de nuevos elementos en un trie también se beneficia de tener los nodos ordenados alfabéticamente. Durante la inserción, podemos encontrar rápidamente la posición correcta para agregar el nuevo nodo o avanzar al siguiente nivel en caso de que ya exista un nodo correspondiente al carácter actual.

En contraste, si los nodos no están ordenados alfabéticamente, la búsqueda e inserción de elementos en el trie se realiza de manera secuencial, lo que podría aumentar el tiempo requerido para completar estas operaciones.

Por lo tanto, mantener un orden alfabético en los nodos de un trie puede mejorar la eficiencia de la operación listStock(String prefix) y facilitar la búsqueda y actualización de elementos en la estructura de datos.

d. Preguntas teóricas de la sección 6

1. Defina el tamaño del problema y calcule el coste asintótico temporal en el caso peor de las operaciones updateStock y retrieveStock para ambas implementaciones.

updateStock:

Para la implementación basada en List (**StockSequence**), el coste en el peor caso de updateStock es O(n). Esto se debe a que, en el peor caso, el elemento a actualizar se encuentra al final de la lista o no está en la lista, por lo que es necesario recorrer todos los elementos de la lista para encontrarlo o agregarlo.

Para la implementación basada en árbol (**StockTree**), el coste en el peor caso de updateStock es O(h), donde h es la altura del árbol. En el peor caso, el árbol se degenera en una lista enlazada y su altura es igual a n, por lo que el coste sería O(n). Sin embargo, si el árbol está equilibrado, el coste sería O(log n). retrieveStock:

Para la implementación basada en List (**StockSequence**), el coste en el peor caso de retrieveStock es O(n). Esto se debe a que, en el peor caso, el elemento a recuperar se encuentra al final de la lista o no está en la lista, por lo que es necesario recorrer todos los elementos de la lista para encontrarlo.

Para la implementación basada en árbol (**StockTree**), el coste en el peor caso de retrieveStock es O(h), donde h es la altura del árbol. Al igual que en updateStock, en el peor caso, el árbol se degenera en una lista enlazada y su altura es igual a n, por lo que el coste sería O(n). Sin embargo, si el árbol está equilibrado, el coste sería O(log n).

Complejidades asintóticas de las operaciones:

O(n): f(n) = nx + b

O(log n): f(n) = a * log(n) + b

n representa el tamaño del problema, que en este caso se refiere al número de elementos en la lista o árbol.

A y b son constantes que dependen de factores como el tiempo necesario para realizar comparaciones y asignaciones.

En la función de complejidad asintótica para updateStock, se ha utilizado la notación O (log n) para la implementación basada en árbol, y O(n) para la implementación basada en lista.

En la función de complejidad asintótica para retrieveStock, se ha utilizado la notación O(n) para ambas implementaciones.

2. Compare el coste asintótico temporal obtenido en la pregunta anterior con los costes empíricos obtenidos. ¿Coincide el coste calculado con el coste real?

Para calcular los costes empíricos orientados a estas funciones he generado archivos de prueba basados en las pruebas proporcionadas por el equipo docente sin listados, para que se centren en los tiempos de retrieve y update.

Tamaño del problema	**Implementación basada en List (StockSequence)**	**Implementación basada en árbol (StockTree)**
95 líneas	51 ms	52 ms
937 líneas	59 ms	58 ms
16.306 líneas	1558 ms	132 ms

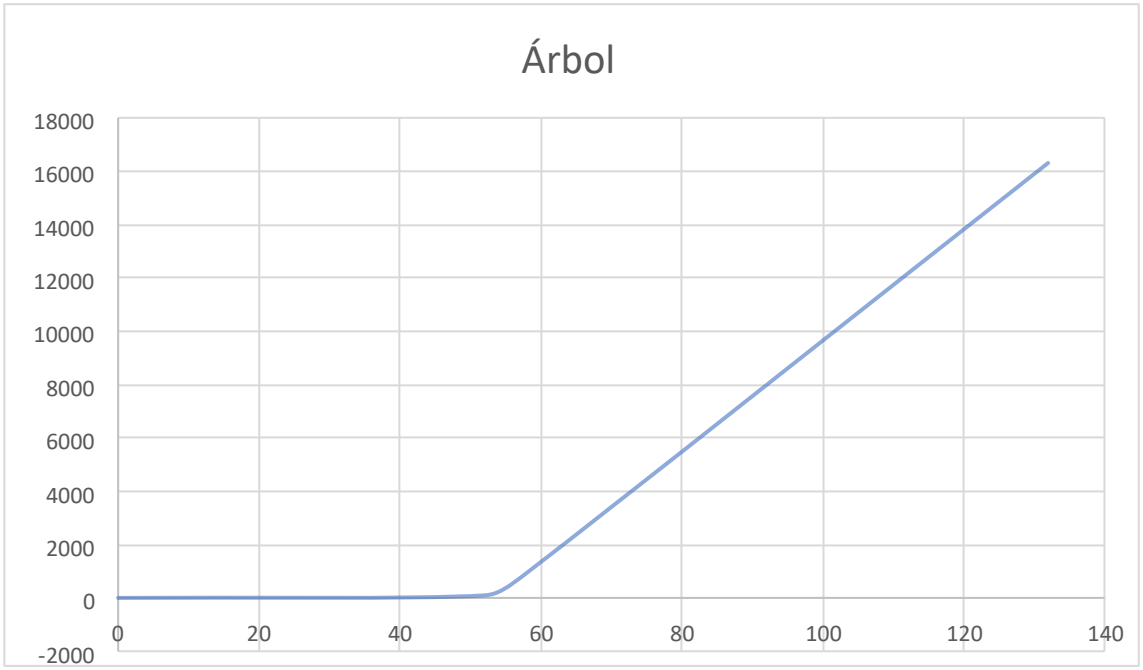
La comparación de los costes asintóticos temporales estimados sugiere que la implementación del árbol general es más eficiente que la implementación de la secuencia. Al observar los costes empíricos obtenidos en las pruebas, podemos confirmar esta afirmación:

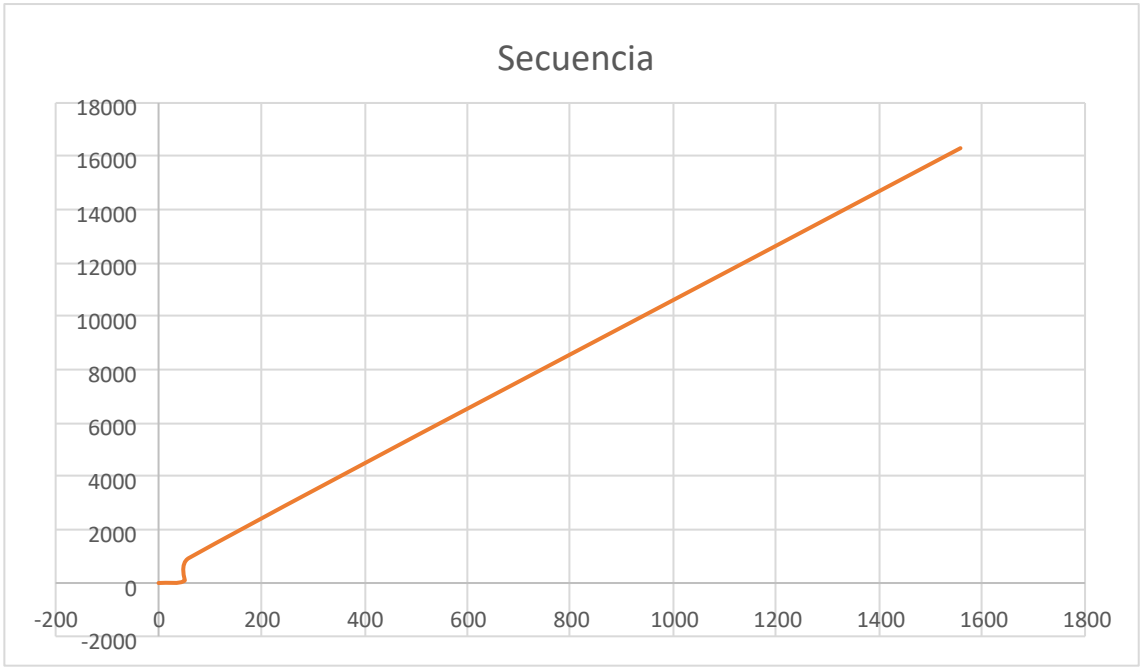
En el caso del árbol general, el tiempo de ejecución crece mucho más lentamente que en el caso de la secuencia a medida que el tamaño del problema aumenta. Por ejemplo, cuando el tamaño del problema aumenta de 95 a 16306 líneas, el tiempo de ejecución del árbol general pasa de 52 ms a 132 ms, mientras que el tiempo de ejecución de la secuencia pasa de 51 ms a 1558 ms.

Esta observación coincide con la expectativa basada en el coste asintótico temporal, lo que sugiere que nuestras estimaciones son consistentes con los costes reales observados en las pruebas.

Por lo tanto, podemos concluir que el coste calculado concuerda con el coste real en función de los datos proporcionados, y la implementación del árbol general es más eficiente que la implementación de la secuencia en términos de tiempo de ejecución.

Representaciones estudio empírico.





2. Diseño

El paquete es.uned.lsi.eped.pract2022_2023 contiene una clase llamada StockSequence y StockTree.

StockSequence

La clase StockSequence es una implementación de una secuencia de pares de stock con métodos para actualizar, buscar y listar el inventario. Utiliza el algoritmo de ordenación por mezcla para ordenar la lista de pares de stock en orden alfabético por el nombre del producto e implementa la interfaz StockIF.

La clase StockSequence tiene un atributo protected SequenceIF<StockPair> stock, que representa la secuencia de pares de stock.

El constructor de la clase StockSequence inicializa la secuencia de pares de stock como una nueva lista vacía.

La clase StockSequence implementa los siguientes métodos de la interfaz StockIF:

retrieveStock(String p): busca un par de stock que corresponde al producto con el nombre p. Devuelve el número de unidades disponibles para ese producto o -1 si no se encuentra el par de stock.

updateStock(String p, int u): actualiza la cantidad de unidades para un producto específico. Si el producto ya existe en la secuencia de pares de stock, se actualiza su número de unidades. Si el producto no existe en la secuencia, se crea un nuevo par de stock con el producto y el número de unidades especificados.

listStock(String prefix): devuelve una secuencia ordenada de pares de stock cuyo producto comienza con el prefijo especificado. La secuencia se ordena alfabéticamente por el nombre del producto.

La clase StockSequence también contiene tres métodos auxiliares:

mergeSort(List<StockPair> stockList, StockPairComparator comparator): ordena una lista de pares de stock utilizando el algoritmo de ordenación por mezcla.

merge(List<StockPair> left, List<StockPair> right, StockPairComparator comparator): combina dos listas ordenadas de pares de stock en una sola lista ordenada.

StockPairComparator.compare(StockPair a, StockPair b): compara dos pares de stock y devuelve un valor entero que indica si el primer par es menor, igual o mayor que el segundo par. Este método se utiliza para definir el orden de los elementos en la secuencia de pares de stock.

StockTree

La clase StockTree implementa una estructura de árbol para almacenar y gestionar pares de stock (nombre del producto y unidades disponibles). Esta clase implementa la interfaz StockIF.

protected GTreeIF<Node> stock: es un árbol genérico que contiene nodos Node.

El constructor StockTree(): inicializa el atributo stock con un nuevo árbol genérico vacío y establece su nodo raíz como un nodo interno con el carácter '#'.

Métodos implementados de la interfaz StockIF:

retrieveStock(String p): busca el par de stock del producto con el nombre p en el árbol. Devuelve el número de unidades disponibles para el producto o -1 si no se encuentra.

updateStock(String p, int u): actualiza la cantidad de unidades disponibles para un producto específico. Si el producto ya existe en el árbol, se actualiza su número de unidades. Si no existe, se crea un nuevo nodo con el producto y el número de unidades especificados.

listStock(String prefix): devuelve una lista de pares de stock cuyos productos comienzan con el prefijo especificado.

La clase StockSequence también contiene cinco métodos auxiliares:

retrieveStockRecursively(GTreeIF<Node> tree, String p, int index): es un método recursivo que busca un producto en el árbol tree con nombre p y devuelve el número de unidades disponibles. El parámetro index indica la posición del carácter en el nombre del producto que se está buscando en el árbol en ese momento.

addChildInOrder(GTreeIF<Node> tree, GTreeIF<Node> newNode): este método añade un nuevo nodo newNode en el árbol tree como hijo del nodo actual, en orden alfabético según la letra del nodo interno. Para hacer esto, itera sobre los hijos del nodo actual y encuentra la posición correcta para insertar el nuevo nodo.

findChild(GTreeIF<Node> tree, char letter): busca en los hijos del nodo actual en el árbol tree un nodo interno que tenga el carácter letter. Si encuentra un nodo con la letra dada, lo devuelve; en caso contrario, devuelve null.

findChild(GTreeIF<Node> tree): busca en los hijos del nodo actual en el árbol tree un nodo de información (que contiene las unidades de stock). Si encuentra un nodo de información, lo devuelve; en caso contrario, devuelve null.

listStockRecursively(GTreeIF<Node> tree, String prefix, List<StockPair> stockList, String nodeString): es un método recursivo que genera una lista de pares de stock que comienzan con el prefijo especificado. El método recorre el árbol tree y va construyendo la cadena de caracteres nodeString con las letras de los nodos internos. Cuando encuentra un nodo de información, añade el par de stock a la lista stockList si el producto comienza con el prefijo dado.

3. Resultados del estudio empírico de costes

A continuación, se presentan los resultados del estudio empírico de costes realizado sobre el programa. El estudio se centró en medir el tiempo de ejecución del programa en diferente volumen de datos y en los dos tipos de estructuras de datos utilizadas en el código.

Se realizaron un total de seis pruebas, tres utilizando una estructura de datos de tipo secuencia y tres utilizando una estructura de datos de tipo árbol general.

Los resultados muestran que el tiempo de ejecución del programa varía significativamente dependiendo del volumen de datos. En general, se observó que el tiempo de ejecución aumenta a medida que aumenta el tamaño de la entrada y que el uso de una estructura de datos de tipo árbol general puede ser más eficiente en algunas situaciones específicas. La algoritmia utilizada probablemente sea más eficiente en un volumen de intermedio.

En resumen, los resultados del estudio empírico de costes muestran la importancia de elegir la estructura de datos adecuada y optimizar el código para lograr un mejor rendimiento del programa.

```
Presione una tecla para continuar . . .
Ejecutando el programa con secuencia (prueba 1)
[##################################################] 100.0%
48 ms
Ejecucion sin errores


Presione una tecla para continuar . . .
Comprobando bateria de pruebas para secuencia (prueba 1)
Los dos ficheros son iguales. ¡¡Prueba superada!!



Presione una tecla para continuar . . .
Ejecutando el programa con arbol general (prueba 1)
[##################################################] 100.0%
48 ms
Ejecucion sin errores


Presione una tecla para continuar . . .
Comprobando bateria de pruebas para arbol general (prueba 1)
Los dos ficheros son iguales. ¡¡Prueba superada!!



Presione una tecla para continuar . . .
Ejecutando el programa con secuencia (prueba 2)
[##################################################] 100.0%
89 ms
Ejecucion sin errores


Presione una tecla para continuar . . .
Comprobando bateria de pruebas para secuencia (prueba 2)
Los dos ficheros son iguales. ¡¡Prueba superada!!



Presione una tecla para continuar . . .
Ejecutando el programa con arbol general (prueba 2)
[##################################################] 100.0%
83 ms
Ejecucion sin errores
```

```
Presione una tecla para continuar . . .
Comprobando bateria de pruebas para arbol general (prueba 2)
Los dos ficheros son iguales. ¡¡Prueba superada!!



Presione una tecla para continuar . . .
Ejecutando el programa con secuencia (prueba 3)
[##################################################] 100.0%
135629 ms
Ejecucion sin errores


Presione una tecla para continuar . . .
Comprobando bateria de pruebas para secuencia (prueba 3)
Los dos ficheros son iguales. ¡¡Prueba superada!!



Presione una tecla para continuar . . .
Ejecutando el programa con arbol general (prueba 3)
[##################################################] 100.0%
93439 ms
Ejecucion sin errores


Presione una tecla para continuar . . .
Comprobando bateria de pruebas para arbol general (prueba 3)
Los dos ficheros son iguales. ¡¡Prueba superada!!
```