Estrategias de Programación y Estructuras de Datos

Grado en Tecnologías de la Información

Práctica 2022 – 2023

Juan R. Barranco Esteban

Preguntas teóricas de la sección 2.1 – Stock: Tipo de datos abstracto

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.

Si bien la intuición dice que lo que más sentido tiene es guardar el listado de productos en una lista, lo cierto es que el programa lo único que hace con ella es obtener un iterador y a partir de este generar una cadena, con lo cual a nivel funcional no tenemos ninguna limitación.

Por otro lado, se ha tomado la decisión de, en ambas implementaciones, mantener en todo momento los elementos ordenados alfabéticamente de forma que al devolver el listado en *listStock* no haya que hacer operaciones de ordenación. Otra operación que hay que realizar en este método es filtrar los resultados según el prefijo recibido por parámetro.

Por tanto, partiendo de esas premisas, necesitaremos generar una secuencia en la que, mientras se va leyendo de la estructura original en que se guarda el stock, se pueda ir añadiendo a la de respuesta cada elemento al final. El tipo de secuencia que nos permite esto de forma más eficiente es la cola, y es la que utilizaremos.

Una pila no nos sería útil porque los elementos se insertan por la cima (principio), con lo cual nos invertiría el orden de los elementos, y utilizar una lista incurriría en costes al generarla (cada vez que queramos añadir un elemento al final hay que recorrerla entera).

Preguntas teóricas de la sección 2.2 – Implementación 1: secuencia ordenada de pares indexados

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

Partiendo de la base de que se ha tomado la decisión de que los elementos se guarden ya ordenados, necesitamos poder insertarlos en posiciones específicas (en el lugar que les corresponde según el orden alfabético). La única secuencia que nos permite hacer esto es una lista.

Utilizar una cola o una pila nos limitaría en esta decisión de diseño, obligándonos a guardar los elementos sin orden y teniendo que ordenarlos al devolver el listado en el método *listStock*, lo cual, como se explicará en el siguiente apartado, se ha estudiado que es menos conveniente.

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

La mejor forma de guardar los elementos de la secuencia es por orden alfabético, ya que es este el que se pide al devolver el listado en *listStock*.

En cualquier caso, hay que realizar esta ordenación ya sea en el guardado o en *listStock*, por lo que en primera instancia puede parecer que es indiferente hacerlo en uno u otro paso. Sin embargo, si se hiciera en *listStock*, habría que ordenar en <u>todas</u> las ocasiones en que se llame al

método, incluso aunque la lista no haya cambiado entre una y otra. Haciendo la ordenación en el guardado, sólo se hace cuando la lista cambia, con lo que a la larga este enfoque es más eficiente.

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

Sí, como se ha explicado antes, si no se guarda la lista ya ordenada, hay que hacer la ordenación en *listStock*.

En lineas generales, el algoritmo para esta ordenación consistiría en construir una segunda lista auxiliar, que sería la que devolveríamos ordenada, recorrer cada uno de los elementos de la original, y para cada uno de ellos, recorrer la lista auxiliar para buscar la posición en que insertar el elemento. La inserción también implica un recorrido de la lista, por tanto, el orden de complejidad del algoritmo sería de $O(n^2)$.

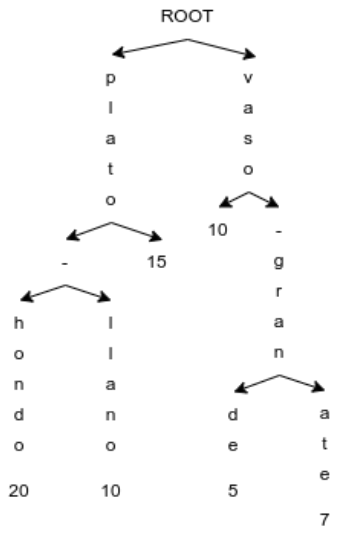
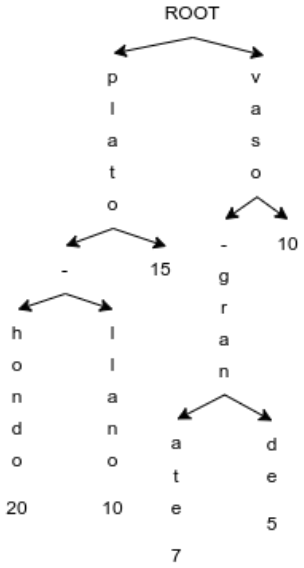
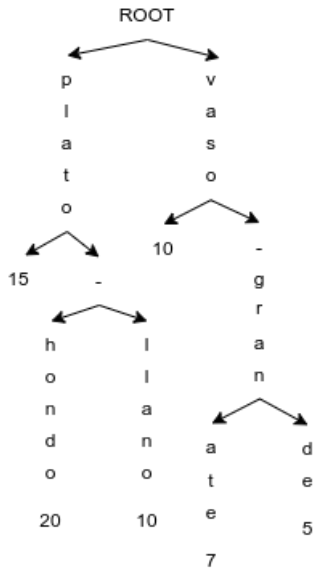
Preguntas teóricas de la sección 2.3 – Implementación 2: árbol general

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.

Vamos a insertar los siguientes productos:

plato 15
vaso-granate 7

Considerando las siguientes formas de colocar los hijos de un nodo:
1. Sin ordenar (cada nodo al final de la lista en el momento de insertarlo)
2. Lista ordenada alfabéticamente de NodeInner, y NodeInfo si lo hubiera al final
3. NodeInfo si lo hubiera al principio, y después lista ordenada alfabéticamente de NodeInner

Forma 1	Forma 2	Forma 3
		

Dado que *listStock* debe devolver la lista de productos ordenada alfabéticamente, podemos descartar de entrada la forma 1, que en principio sería la que haría que la inserción de nuevos productos fuera la más sencilla de todas, pero que complicaría bastante el algoritmo de ordenación al devolver el resultado.

Tanto la forma 2 como la forma 3 permiten obtener la lista ordenada de forma más sencilla, se necesitaría para ello un algoritmo parecido al recorrido en preorden, pero en el que cada salida para cada hoja contenga la ruta completa recorrida desde la raíz.

La diferencia entre ambas formas radica en que, por ejemplo, “plato” va en orden alfabético antes que “plato-hondo”. Por tanto, en un listado, “plato, 15” debe ir antes que “plato-hondo, 20”, con lo que es conveniente que los NodeInfo, de haberlos, sean siempre el primer elemento de la lista de hijos de su padre, de modo que el producto al que hacen referencia unas unidades se añada al listado antes que un producto con un nombre con el que comparte raíz pero es más largo.

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.

Aparte de la mencionada en el apartado anterior, otra razón que hace conveniente que NodeInfo esté en la primera posición es que, puesto que los hijos de un nodo se dan como una lista (y no otro tipo de secuencia, sea una cola o una pila), si el NodeInfo estuviera al final, cuando se necesitara actualizarlo se tendría que recorrer la lista entera hasta llegar a la última posición, haciendo el algoritmo más ineficiente. Colocándolo al principio, el acceso es inmediato.

Esta consideración es irrelevante para nodos NodeInner, puesto que, estén al principio o al final, siempre los vamos a tener que recorrer para buscar uno de ellos o encontrar la posición en que insertar uno nuevo. No obstante, sí es de importancia que estén ordenados alfabéticamente puesto que de esta forma podemos encontrar esa posición de forma unívoca y salir del bucle en el momento en que hemos llegado a ella.

Por tanto, podemos concluir que la mejor forma de ordenar la lista de hijos de un nodo es, primero el NodeInfo si lo hubiera, y después los NodeInner ordenados alfabéticamente.

Preguntas teóricas de la sección 6 – Estudio del coste

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.

StockSequence

- **updateStock**

```
38  @Override
39  public void updateStock(String p, int u) {
40
41      IteratorIF<StockPair> iterator = this.stock.iterator();
42      boolean found = false;
43      boolean inserted = false;
44      int pos = 1;
45      while (iterator.hasNext() && !found && !inserted) {
46          StockPair element = iterator.getNext();
47          if (element.getProducto().equals(p)) {
48              element.setUnidades(u);
49              found = true;
50          } else if (element.getProducto().compareTo(p) > 0) {
51              StockPair newElement = new StockPair(p, u);
52              this.stock.insert(pos, newElement);
53              inserted = true;
54          }
55          pos++;
56      }
57
58      if (!found && !inserted) {
59          StockPair newElement = new StockPair(p, u);
60          this.stock.insert(this.stock.size() + 1, newElement);
61          inserted = true;
62      }
63  }
64
```

En el caso de *updateStock* de *StockSequence*, el tamaño del problema es el nº de productos en el stock, es decir, el coste es mayor cuanto mayor sea el número de productos que existan ya guardados en el stock, siendo el caso peor el añadir un producto que deba ir al final de la lista.

En el análisis de costes vemos lo siguiente:

- Lineas 41 a 44: todas las operaciones son de orden $O(1)$.
- Bucle *while* (l. 45 a 56): el número de iteraciones es n en el caso peor. Todas las operaciones internas son de orden $O(1)$, excepto *List.insert* (l. 52) que por debajo conlleva un recorrido de la lista hasta la posición *pos,* cuyo coste es de orden $O(n)$. Por tanto, el coste del bucle completo es de orden $O(n^2)$.
- Lineas 58 a 62: coste $O(n)$ debido al insert de l. 60 (resto de operaciones $O(1)$)

El orden de complejidad del método es el mayor de los referidos, esto es, $O(n^2)$.

- **retrieveStock:**

```
17    @Override
18    public int retrieveStock(String p) {
19        StockPair stockElement = null;
20
21        IteratorIF<StockPair> iterator = this.stock.iterator();
22        boolean found = false;
23        while (iterator.hasNext() && !found) {
24            StockPair element = iterator.getNext();
25            if (element.getProducto().equals(p)) {
26                stockElement = element;
27                found = true;
28            }
29        }
30
31        if(stockElement == null) {
32            return -1;
33        }
34
35        return stockElement.getUnidades();
36    }
37
```

En cuanto a *retrieveStock*, el tamaño del problema es de igual modo el nº de productos en el stock, es decir, siendo el caso peor la consulta de un producto que esté al final de la lista.

Análisis de costes:

- L. 19 a 22 y 31 a 35: todas las operaciones son de orden $O(1)$.
- Bucle *while* (l. 23 a 29): Todas las operaciones internas son de $O(1)$. El número de iteraciones en el caso peor es n. Por tanto, el bucle es de orden $O(n)$.

El método pertenece en consecuencia al $O(n)$.

StockTree

- **updateStock**

```
48    @Override
49    public void updateStock(String p, int u) {
50        // Obtenemos cola auxiliar con los nodos
51        // preparados para el árbol
52        Queue<Node> queue = this.getAuxQueue(p);
53
54        // leemos o insertamos recursivamente en el árbol los nodos de la cola
55        GTreeIF<Node> tree = this.readPath(this.stock, queue, true);
56
57        NodeInfo node = new NodeInfo(u);
58        this.updateInfoChild(tree, node);
59    }
60
```

En el caso de *StockTree*, para *updateStock*, con el fin de estimar cuál es el tamaño del problema hay que analizar los submétodos que se usan. En particular tendremos que mirar *readPath*, que se encarga de buscar el recorrido que sigue en el árbol una determinada secuencia de caracteres, haciéndolo de forma recursiva.

```
80
81    private GTreeIF<Node> readPath(GTreeIF<Node> parentTree, Queue<Node> queue, boolean insertIfNotFound) {
82        GTreeIF<Node> targetTree;
83
84        Node node = queue.getFirst();
85        queue.dequeue();
86
87        GTreeIF<Node> child = insertIfNotFound ? this.getOrCreateChild(parentTree, node) : this.getChild(parentTree, node);
88
89        if (child == null || queue.isEmpty()) {
90            targetTree = child;
91        } else {
92            targetTree = this.readPath(child, queue, insertIfNotFound);
93        }
94
95        return targetTree;
96    }
97
```

Vemos que este método a su vez delega el comportamiento a seguir, en el momento en que no se encuentre un nodo, y dependiendo del valor recibido en el parámetro *insertIfNotFound*, en los submétodos *getOrCreateChild* o *getChild*. En este caso seguirá *getOrCreateChild*, ya que si un nodo no es encontrado entonces lo creamos.

Estos métodos tienen un bucle *while,* pero el número máximo de iteraciones se corresponde al número máximo de hijos que un nodo puede tener, o lo que es lo mismo, el número total de caracteres distintos que puede haber en un nodo. Este es un número fijo, por lo que el orden de complejidad asociado es $O(1)$.

A continuación, vemos que, como dijimos anteriormente, *readPath* es recursiva. Esta recursividad se hace sobre la longitud de la cola de caracteres para la cual se quiere encontrar el recorrido en el árbol. Es decir, el tamaño del problema n es la longitud de los identificadores de productos.

El número de recursiones que se hacen es 1, y la complejidad disminuye por sustracción de 1.
El coste del caso base es asimismo 1, y el de las operaciones no recursivas es el de las lineas 82 a 87, que son de orden $O(1)$.

El cálculo de la complejidad de *readPath* será entonces, por tanto:

$$O(n \cdot c_{nr}(n) + cb(n)) = O(n)$$

Volviendo a *updateStock*, podemos comprobar que el resto de operaciones que se hacen en ella son de $O(1)$, por lo que su complejidad es $O(n)$, con tamaño del problema la longitud de los identificadores de productos, y caso peor añadir un producto con un nombre de longitud manifiestamente larga.

- **retrieveStock**

```
20
21    @Override
22    public int retrieveStock(String p) {
23
24        int units = 0;
25
26        // Obtenemos cola auxiliar con los nodos
27        // a leer en el árbol
28        Queue<Node> queue = this.getAuxQueue(p);
29
30        // leemos recursivamente en el árbol los nodos de la cola
31        GTreeIF<Node> tree = this.readPath(this.stock, queue, false);
32
33        if (tree == null) {
34            units = -1;
35        } else {
36            GTreeIF<Node> firstChild = tree.getChildren().get(1);
37            if (firstChild == null || firstChild.getRoot().getNodeType() != Node.NodeType.INFO) {
38                units = -1;
39            } else {
40                NodeInfo node = (NodeInfo) firstChild.getRoot();
41                units = node.getUnidades();
42            }
43        }
44
45        return units;
46    }
47
```

El análisis sobre *retrieveStock* es muy similar al de *updateStock* con la salvedad de que, en lugarde *getOrCreateChild*, se hace uso de *getChild*. El orden de este método es igualmente $O(1)$, por lo que, siguiendo el mismo cuso de deducción que seguimos en *updateStock*, el orden de complejidad de *retrieveStock* es el mismo: $O(n)$.

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

Dado que hemos determinado que los costes dependen de distintos parámetros en las dos implementaciones, haremos pruebas distintas para cada una de ellas.

Haremos las pruebas utilizando ficheros de carga que utilicen únicamente el comando *compra*, ya que este hace una llamada a *retrieveStock* y una a *updateStock* por cada linea.

StockSequence

Como comentábamos, aquí el tamaño del problema depende de la cantidad de productos y el caso peor es añadir un producto al final de la lista según orden alfabético.

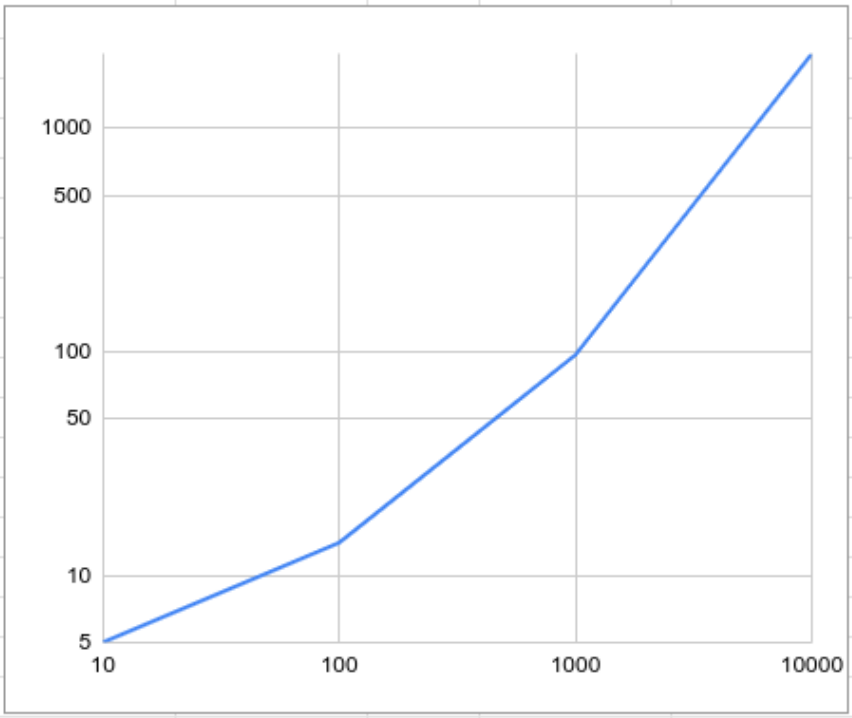
Haremos cuatro pruebas con 10, 100, 1000 y 10000 lineas, todas de productos distintos, y ordenados de tal forma que cada uno de ellos se tenga que insertar siempre al final (ordenados alfabéticamente en el fichero de carga).

Los archivos con que se hacen estas pruebas son los adjuntos seq_10.txt, seq_100.txt, seq_1000.txt, seq_10000.txt.

Cabe esperar que la evolución de tiempos de ejecución entre cada una de las cuatro pruebas sea aproximadamente una función cuadrática, dado que este es el orden al que pertenece la llamada de mayor orden de las dos (*updateStock*).

En el siguiente gráfico se representan los resultados de estas pruebas:

SECUENCIA - NUMERO PRODUCTOS	
num	ms
10	5
100	14
1000	97
10000	2160



Se han ajustado las escalas de los ejes a escala logarítmica de forma que se pueda apreciar claramente cómo se incrementa la pendiente de la gráfica conforme se aumenta el número de productos. Esto concuerda con la hipótesis de complejidad de orden cuadrático.

StockTree

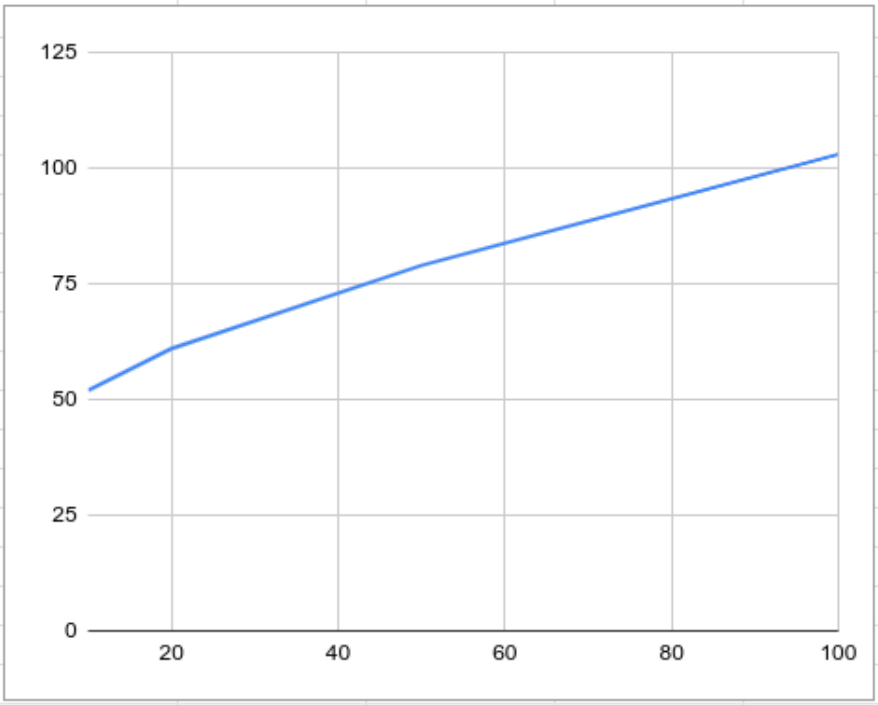
En *StockTree* el tamaño del problema depende de la longitud de los identificadores de producto. Haremos cuatro pruebas con 1000 lineas con productos, no necesariamente ordenados alfabéticamente, de 10, 20, 50 y 100 caracteres respectivamente.

Los archivos de carga se encuentran adjuntos como tree_10.txt, tree_20.txt, tree_50.txt, tree_100.txt.

De estas pruebas cabe esperar que la evolución de tiempos sea lineal (aproximadamente).

En el siguiente gráfico se representan los resultados:

ÁRBOL – LONGITUD PRODUCTOS	
lg	ms
10	52
20	61
50	79
100	103



Se puede ver como, efectivamente, el incremento es aproximadamente lineal.