Estrategias de Programación y Estructuras de Datos

Grado en Ingeniería Informática

Grado en Tecnologías de la Información

Práctica curso 2021-2022

Enunciado y orientaciones

1. Presentación del problema

Los arrays son estructuras de datos indexadas, es decir: permiten el acceso a los datos a través de un índice numérico entero que se mueve entre un primer y un último valor (los cuales dependen del lenguaje de implementación). En este sentido se comportan de manera similar a las listas indexadas, con la salvedad de que para insertar un elemento en una determinada posición, es necesario desplazar, previamente, todos los elementos con un índice igual o superior al índice siguiente.

Normalmente, los arrays se implementan reservando una zona consecutiva de memoria con suficiente capacidad para almacenar todos los elementos del array comprendidos entre su primer y último índice. De esta forma, el cálculo de la posición de memoria donde se encuentra un elemento cualquiera del array se puede realizar en tiempo constante en función de su índice, lo que nos proporciona una estructura de datos en la que el acceso a los elementos se realiza en tiempo constante.

Sin embargo, no todos los lenguajes de programación permiten la reserva de una zona consecutiva de memoria que pueda ser accedida y/o modificada, por lo que esta estructura de datos no es generalizable a cualquier lenguaje.

Utilizar arrays es muy útil cuando tenemos datos que pueden ser indexados, ya que normalmente podremos acceder a ellos en tiempo constante. Sin embargo, hay ocasiones en las que utilizar un array supone un gasto excesivo (y tal vez inasumible) de memoria. Por ejemplo, si los índices que realmente utilizamos en el array son pocos (con respecto al total de índices disponibles) y están muy separados, toda la memoria reservada para los índices no utilizados estaría desaprovechada.

En estos casos se utilizan estructuras denominadas arrays dispersos (en inglés: "*sparse arrays*"), que nos permiten almacenar y recuperar elementos mediante su índice de la misma forma que haríamos en un array, pero que no reservan memoria para todos los elementos que puedan ser indexados. En estas estructuras el acceso a los elementos no se puede realizar en tiempo constante, a cambio de no desperdiciar toda la memoria de los índices no utilizados del array.

En esta práctica consideraremos dos implementaciones alternativas de arrays dispersos en la que almacenaremos los elementos del array mediante pares `<índice,valor>`. La primera implementación utilizará una secuencia para almacenar esos pares, mientras que la segunda, empleará un árbol binario en el cual el lugar donde almacenar un elemento se calculará mediante la secuencia de dígitos de la representación binaria de su índice. Una vez implementadas ambas opciones, se realizará el cálculo teórico de su coste, comparándolo con un estudio empírico del coste.

2. Enunciado de la práctica

La práctica consiste en:

(i) Razonar e implementar dos soluciones alternativas para el tipo de datos abstracto array disperso, una basada en secuencias y otra basada en árboles binarios. La implementación utilizará los tipos de datos programados por el Equipo Docente.
(ii) Implementar un programa que aplique ambas implementaciones.
(iii) Ejecutar el programa anterior sobre juegos de prueba empleando ambas implementaciones y realizar estudios comparativos del tiempo de ejecución de cada una de ellas.

2.1. Array Disperso: Tipo de Datos Abstracto

El tipo de datos abstracto array disperso que vamos a implementar funciona como un array en el que el acceso a sus elementos se realiza mediante un índice. Las diferencias con un array son dos:

1. No tienen ninguna restricción sobre el número máximo de elementos a almacenar. Esto significa que se podrán añadir cuantos elementos sean necesarios mediante nuevos índices.
2. Si un índice no está utilizado, no se reserva memoria para ningún elemento bajo ese índice. Esto significa que la estructura de datos deberá liberar la memoria reservada para un elemento cuando dicho elemento sea borrado.

Además, un array disperso puede ser considerado como una secuencia de elementos cuyo orden viene dado por el orden de sus índices. De esta manera, situaremos las implementaciones de arrays dispersos dentro de las secuencias en nuestro esquema de tipos abstractos de datos.

Por lo tanto, además de las operaciones propias de las secuencias (y las heredadas de las colecciones), este tipo abstracto de datos incluye las operaciones descritas en la siguiente interfaz:

```
/* Representa un array disperso en el que los elementos se
 * indexan bajo un indice entero y no se reserva memoria
 * para un elemento si no se ha usado su indice
 */
public interface SparseArrayIF<E> extends SequenceIF<E> {

    /* Indexa el elemento elem bajo el indice pos.
     * @PRE: pos >= 0
     * Si ya habia un elemento bajo el mismo indice, el nuevo
     * elemento substituye al anterior.
     */
    public void set(int pos,E elem);

    /* Devuelve el elemento indexado bajo el indice pos.
     * @PRE: pos >= 0
     * Si no existe un elemento indexado bajo el indice pos,
     * devuelve null.
     */
    public E get(int pos);

    /* Elimina el elemento indexado bajo el índice pos.
     * @PRE: pos >= 0
     * Elimina toda la memoria utilizada para almacenar el elemento
     * borrado.
     * Si no existe un elemento indexado bajo el indice pos,
     * esta operacion no realiza ninguna modificacion en la estructura.
     */
    public void delete(int pos);

    /* Devuelve un iterador de todos los indices utilizados
     * en el array disperso, por orden creciente de índice.
     */
    public IteratorIF<Integer> indexIterator();
}
```

Las implementaciones de `SparseArrayIF` utilizarán la siguiente clase para relacionar los índices con los elementos que indexan:

```java
/* Representa un par indexado */
public class IndexedPair<E> {

    private int index;
    private E value;

    /* Constructor */
     * @PRE: index >= 0
    public IndexedPair(int index,E value) {
        this.index = index;
        this.value = value;
    }

    /* Obtiene el indice */
    public int getIndex() {
        return this.index;
    }

    /* Obtiene el valor */
    public E getValue() {
        return this.value;
    }

    /* Modifica el valor */
    public void setValue(E value) {
        this.value = value;
    }
}
```

Preguntas teóricas de la sección 2.1

Antes de avanzar, reflexiona y responde a las siguientes preguntas:

1. ¿Qué debería devolver la operación `size()` de `CollectionIF` para un array disperso? ¿Qué operaciones de `SparseArrayIF` deben tenerse en cuenta para que el código de `size()` ya implementado en `Collection` sirva para un array disperso? Razona tus respuestas.
2. ¿A qué clase debería extender una implementación de `SparseArrayIF`: `Collection` o `Sequence`? Razona tu respuesta.

2.2. Implementación 1: *secuencia ordenada de pares indexados*

Esta implementación consiste en utilizar una secuencia ordenada de pares de la clase `IndexedPair` como estructura de datos de soporte para almacenar los elementos bajo sus correspondientes índices.

Preguntas teóricas de la sección 2.2

Antes de implementar esta solución, reflexiona y responde a las siguientes preguntas:

1. ¿Qué tipo de secuencia sería la adecuada para realizar esta implementación? ¿Por qué? ¿Qué consecuencias tendría el uso de otro tipo de secuencias?
2. ¿Cuál sería el orden adecuado para almacenar los pares en la secuencia? ¿Por qué? ¿Qué consecuencias tendría almacenarlos sin ningún tipo de orden?
3. ¿Afectaría el orden a la eficiencia de alguna operación de `Collection`? Razona tu respuesta.

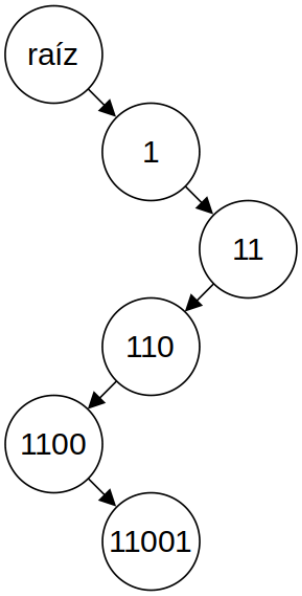
2.3. Implementación 2: *árbol binario*

Para esta implementación, utilizaremos un árbol binario en el que el nodo que almacena el elemento indexado bajo el índice x se localiza siguiendo la secuencia de dígitos de la representación binaria de x. La forma de localizar el nodo adecuado es la siguiente:

1. Comenzamos el recorrido en el nodo raíz del árbol binario.
2. Se recorre la secuencia de dígitos de la representación binaria de la posición buscada, desde el dígito más significativo al menos significativo.
   a. Si el dígito es un 0, se desciende por el hijo izquierdo del nodo actual.
   b. Si el dígito es un 1, se desciende por el hijo derecho del nodo actual.
3. El nodo buscado es el nodo donde termina el camino al haber recorrido todos los dígitos de la secuencia.

Por ejemplo, si buscamos la posición 25, los pasos descritos anteriormente serían los siguientes, que se corresponden con el recorrido gráfico que se puede ver a la derecha:

1. Comenzamos el recorrido en el nodo raíz del árbol binario.
2. Recorremos la secuencia de dígitos de la representación binaria de 25, que es 11001:
   a. Tenemos un 1, descendemos por el hijo derecho.
   b. Tenemos un 1, descendemos por el hijo derecho.
   c. Tenemos un 0, descendemos por el hijo izquierdo.
   d. Tenemos un 0, descendemos por el hijo izquierdo.
   e. Tenemos un 1, descendemos por el hijo derecho.
3. Hemos llegado al nodo buscado.



En esta implementación, la operación `set` será la encargada de añadir tantos nodos como sean necesarios para completar el recorrido desde la raíz al nodo que contendrá el elemento a insertar. De igual manera, la operación `delete` se encargará de borrar todos los nodos que ya no sean necesarios tras eliminar el nodo que contenía el elemento borrado.

Al igual que en la implementación anterior, cada nodo que contenga un elemento deberá almacenar un objeto de la clase `IndexedPair`.

Preguntas teóricas de la sección 2.3

Nuevamente, antes de implementar esta solución, reflexiona y responde a las siguientes preguntas:

1. Utilizando papel y lápiz inserta elementos en los índices 0,1,2,3… En vista del árbol obtenido, ¿cómo podrías recorrer el árbol para encontrar los elementos por orden creciente de índice? Razona tu respuesta.
2. Dado que podemos localizar el nodo que contiene el par indexado con el índice x mediante la búsqueda descrita anteriormente, ¿es necesario almacenar un par indexado en los nodos o bastaría con almacenar únicamente el elemento a indexar? Razona tu respuesta.

3. Diseño de la práctica

A continuación, veremos el diseño de clases de la práctica para su implementación. Para las clases ya programadas se explica su funcionamiento. Para las clases que deban ser completadas por los estudiantes se listan las operaciones que contienen y cuál será su comportamiento esperado.

3.1. SparseArrayIF.java

Se trata de la interfaz presentada en la sección anterior que especifica las operaciones de los arrays dispersos y **no debe modificarse**.

3.2. IndexedPair.java

Esta clase, también presentada en la sección anterior, representa pares indexados `<índice,valor>`, siendo el índice un entero mayor o igual que cero. **No debe modificarse**.

3.3. SparseArraySequence.java

Esta clase implementará los arrays dispersos tal y como se ha descrito en el apartado 2.2, empleando una secuencia de objetos de la clase `IndexedPair`. Junto a este enunciado se adjunta el esqueleto de esta clase.

3.4. SparseArrayBtree.java

Esta clase implementará los arrays dispersos tal y como se ha descrito en el apartado 2.3, empleando un árbol binario en el que los nodos que contienen información almacenan un objeto de la clase `IndexedPair`. Se incluye la implementación de una operación que devuelve la secuencia de dígitos de la representación binaria de un número en forma de pila (una secuencia) de booleanos, representando **`false`** un 0 y **`true`** un 1. Junto a este enunciado se adjunta el esqueleto de esta clase.

3.5. Main.java

Esta clase contiene el programa principal y se da completamente terminada por parte del Equipo Docente y **no debe modificarse**. El funcionamiento es el siguiente:

1. El programa requiere tres argumentos:
    a. El primero es una cadena de caracteres que puede ser `SEQUENCE` o `BTREE`, que indica cuál de las dos implementaciones propuestas de los arrays dispersos se va a emplear.

    b. El segundo es la ruta del fichero de entrada del programa. Un fichero de entrada será un fichero de texto en el que cada línea contiene una de las siguientes operaciones de los arrays dispersos sobre elementos de tipo `String` que no contienen espacios:

        - `set indice elemento`
        - **`get indice`**
        - `delete indice`
        - **`indexiterator`**
        - **`iterator`**
        - **`size`**
        - **`isempty`**
        - **`contains elemento`**
        - `clear`

        Las operaciones marcadas en negrita devuelven un resultado, lo que se reflejará de forma adecuada en la salida del programa.

    c. El tercero es la ruta del fichero de salida generado por el programa. La salida

generada se explica en el punto 3.

2. Se comprueba que se han proporcionado adecuadamente los tres parámetros. En caso contrario, se lanza un mensaje de error y termina el programa. En el caso del primer argumento, se comprueba que la cadena de caracteres es `SEQUENCE` o `BTREE` y se crea un array disperso de elementos de tipo `String` vacío con la implementación correspondiente. Para el segundo argumento, se comprueba que existe el fichero de entrada y puede leerse. Por último, para el tercer argumento se comprueba que existe la carpeta donde se quiere escribir el fichero de salida y que dicho fichero puede ser escrito (o sobreescrito si ya existía).

3. El programa lee línea a línea el fichero de entrada, de manera que se aplican en orden las operaciones correspondientes sobre el array disperso. Tras ejecutar cada operación, se escribe en el fichero de salida la línea leída y en el caso de que la operación ejecutada devuelva un resultado, a continuación escribirá la cadena "`:  `" y el resultado de la operación siguiendo este formato:

    - **`get indice`**: escribe el elemento indexado bajo ese índice, o la cadena "`null`" si dicho elemento no existe.
    - **`indexiterator`**: escribe la secuencia de índices utilizados en el array disperso.
    - **`iterator`**: escribe la secuencia de elementos contenida en el array disperso.
    - **`size`**: escribe el número de elementos contenidos en el array disperso.
    - **`isempty`**: escribirá "`true`" si la estructura está vacía y "`false`" en caso contrario.
    - **`contains elemento`**: escribirá "`true`" si el array disperso contiene el elemento indicado y "`false`" en caso contrario.

4. El programa termina mostrando el tiempo de ejecución en milisegundos del paso anterior.

4. Implementación.

Se deberá realizar un programa en Java llamado **`eped2022.jar`** que contenga todas las clases anteriormente descritas completamente programadas. Todas ellas se implementarán en un único paquete llamado:

`es.uned.lsi.eped.pract2021_2022`

Para la implementación de las estructuras de datos de soporte **se deberán utilizar las interfaces y las implementaciones proporcionadas por el Equipo Docente** de la asignatura.

A través del Curso Virtual, el Equipo Docente proporcionará el código de todas las clases (total o parcialmente) implementado según la descripción que se ha hecho en este documento.

Esto significa que la lectura de los parámetros de entrada, lectura de los ficheros y salida del programa ya está programada por el Equipo Docente y los estudiantes no tienen que realizar ni modificar nada sobre estos temas.

5. Ejecución y juegos de prueba.

Para la ejecución del programa se deberá abrir una consola y ejecutar:

```
java –jar eped2022.jar <implementacion> <entrada> <salida>
```

siendo:

- **<implementacion>** SEQUENCE o BTREE según el tipo de implementación a utilizar
- **<entrada>** nombre del fichero de entrada con las operaciones a realizar
- **<salida>** nombre del fichero de salida

El Equipo Docente proporcionará, a través del curso virtual, unos juegos de prueba para que los estudiantes puedan comprobar el correcto funcionamiento del programa. Si se supera el juego de pruebas privado de los tutores (que será diferente del proporcionado a los estudiantes), la práctica tendrá una calificación mínima de 4 puntos, a falta de la evaluación del estudio empírico del coste y de las preguntas teóricas.

6. Estudio empírico del coste.

Queremos estudiar empíricamente el tiempo de ejecución de cada implementación dependiendo del tamaño del problema.

Para realizar esta tarea, el estudiante tiene libertad para generar su propio juego de pruebas, explicando el diseño de forma razonada en función de su uso para medir tiempos de ejecución y reportando sus resultados experimentales.

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 `set` y `get` 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?

7. Documentación y plazos de entrega.

La práctica supone un 20% de la calificación de la asignatura, y es necesario aprobarla para superar la asignatura. Además, será necesario obtener, al menos, un 4 sobre 10 en el examen presencial para que la calificación de la práctica sea tenida en cuenta de cara a la calificación final de la asignatura.

Los Centros Asociados organizarán sesiones de prácticas en las que los tutores monitorizarán y orientarán a los estudiantes en su realización de la práctica. La asistencia a estas sesiones no es obligatoria, pero se recomienda encarecidamente a los estudiantes asistir a ellas. La interacción con el tutor en esas sesiones presenciales puede influir en la calificación de la práctica. Estas sesiones son organizadas por los Centros Asociados teniendo en cuenta sus recursos y el número de estudiantes matriculados, por lo que en cada Centro las fechas serán diferentes. Los estudiantes deberán, por tanto, dirigirse a su tutor para conocer las fechas de celebración de estas sesiones.

De igual modo, el plazo y forma de entrega son establecidos por los tutores de forma independiente en cada Centro Asociado, por lo que deberán ser consultados también con ellos.

La documentación que debe entregar cada estudiante consiste en:

- Memoria de práctica, en la que se deberán responder a las preguntas teóricas y se incluirá el

diseño y resultados del estudio empírico de costes.

- Implementación en Java de la práctica, de la cual se deberá aportar **únicamente el código fuente de las clases `SparseArraySequence.java` y `SparseArrayBTree.java`**.
- Juego de pruebas diseñado por el estudiante para el estudio empírico de costes.

Adicionalmente, los estudiantes deberán entregar **obligatoriamente** una copia de su práctica al Equipo Docente a través del portlet que se habilitará en el Curso Virtual.

8. Calificación.

La calificación de la práctica será otorgada por el tutor, en función de las respuestas a las preguntas teóricas, la superación del juego de pruebas público y del juego de pruebas privado de los tutores y el diseño y resultados del estudio empírico de costes. El tutor también podrá tener en cuenta el grado de participación activa en las sesiones de tutorización relativas a la práctica. La puntuación se distribuye de la siguiente manera:

- 4 puntos en función del código de la práctica: corrección, presentación, documentación. **Para aprobar es imprescindible que el código supere el juego de pruebas privado de los tutores**.
- 3 puntos en función del estudio empírico del coste. Se valorará el diseño del juego de pruebas y la presentación y análisis de los resultados empíricos.
- 3 puntos en función de las respuestas a las preguntas teóricas. Se valorará la corrección de las respuestas, su justificación y el nivel de detalle.
- Hasta un punto adicional en función de la participación e interacción en las tutorías del centro orientadas a la práctica.

La práctica es un trabajo individual de cada estudiante. **Tanto el código como las respuestas a las preguntas teóricas y el diseño del juego de pruebas individual para el estudio empírico de costes deben ser originales del estudiante**. Si se detectan casos de plagio en cualquiera de estos aspectos se suspenderá a los estudiantes involucrados y podrán ser objeto de sanciones por parte de la universidad.