Práctica 1
Vector

Estructuras de datos

1. Meta

Que el alumno domine el manejo de información almacenada arreglos.

2. Objetivos

Al finalizar la práctica el alumno será capaz de:

- Transferir información entre arreglos cuando la capacidad de un arreglo ya no es adecuada.
- Diferenciar entre el tipo de dato abstracto **Vector** y su implementación.

3. Antecedentes

Un arreglo en la computadora se caracteriza por:

1. Almacenar información en una región contigua de memoria.
2. Tener un tamaño fijo.

Ambas características se derivan del sistema físico en el cual se almacena la información y sus limitaciones. Por el contrario, un **tipo de dato abstracto** es una entidad matemática y debe ser independiente de el medio en que se almacene. Para ilustrar mejor este concepto, se pide al alumno programar una clase `Vector` que obedezca a la definición del tipo de dato abstracto que se incluye a continuación, utilizando arreglos y aquellas técnicas requeridas para ajustar las diferencias de comportamiento entre ambas entidades.

Los métodos de manipulación que no devuelven ningún valor no pueden ser definidos estrictamente como funciones, por ello a menudo se refiere a ellos como **subrutinas**. Para resaltar este hecho se utiliza el símbolo $\xrightarrow{?}$ al indicar el valor de regreso.

Definición 1: Vector

Un **Vector** es una estructura de datos tal que:

1. Puede almacenar n elementos de tipo T.
2. A cada elemento almacenado le corresponde un **índice** i con $i \in [0, n-1]$. Denotaremos esto como $V[i] \to e$.
3. Para cada índice hay un único elemento asociado.
4. La capacidad máxima n puede ser incrementada o disminuida.

Nombre: $Vector$.
Valores: $\mathbb{N}$, T, con $null \in T$.
Operaciones: Sea inc una constante con $inc \in \mathbb{N}, inc > 0$ y $this$ el vector sobre el cual se está operando.

Constructores :

Vector(): $\emptyset \to Vector$

Precondiciones: $\emptyset$

Postcondiciones :

- Un $Vector$ es creado con $n = inc$.
- A los índices $[0, n-1]$ se les asigna $null$.

Métodos de acceso :

lee(this, i) $\to$ **e:** $Vector \times \mathbb{N} \to T$

Precondiciones :

- $i \in \mathbb{N}, i \in [0, n-1]$

Postcondiciones :

- $e \in T$, e es el elemento almacenado en $Vector$ asociado al índice i.

leeCapacidad(this): $Vector \to \mathbb{N}$

Precondiciones: $\emptyset$

Postcondiciones: Devuelve n

Métodos de manipulación :

asigna(this, i, e): $Vector \times \mathbb{N} \times T \xrightarrow{?} \emptyset$

Precondiciones :

- $i \in \mathbb{N}, i \in [0, n-1]$
- $e \in T$

Postcondiciones :

- El elemento e queda almacenado en el vector, asociado al índice i [a].

asignaCapacidad(this, n'): $Vector \times \mathbb{N} \xrightarrow{?} \emptyset$

Precondiciones: $n' \in \mathbb{N}, n' > 0$

Postcondiciones :

- A n se le asigna el valor n'.
- Si $n' < n$ los elementos almacenados en $[n', n-1]$ son eliminados.
- Si $n' > n$ a los índices $[n, n'-1]$ se les asigna $null$.

aseguraCapacidad(this, n'): $Vector \times \mathbb{N} \xrightarrow{?} \emptyset$

Precondiciones: $n' \in \mathbb{N}, n' > 0$

Postcondiciones :

- Si $n' < n$ no pasa nada.
- Si $n' < inc$ no pasa nada.
- Si $n' > n$: sea $nn = 2^i inc$ tal que $nn > n'$, a n se le asigna el valor de nn.

Esta definición puede ser traducida a una implementación concreta en cualquier lenguaje de programación, en particular, a `Java`. Dado que `Java` es un lenguaje orientado a objetos, se busca que la definición del tipo abstracto de datos se vea reflejada en la interfaz pública de

[a] Dado que el elemento asociado al índice es único, cualquier elemento que hubiera estado asociado a i deja de estarlo.

la clase que le corresponde, mientras que los detalles de implementación se vuelven privados. El esqueleto que se muestra a continuación corresponde a esta definición. Obsérvese cómo las precondiciones y postcondiciones pasan a formar parte de la documentación de la clase, mientras que el dominio y el rango de las funciones están especificados en las firmas de los métodos. Igualmente, el argumento `this` es pasado implícitamente por `Java`, por lo que no es necesario escribirlo entre los argumentos de la función; otros lenguajes de programación, como `Python`, sí lo solicitan.

```
/*
 * Codigo utilizado para el curso de Estructuras de Datos.
 * Se permite consultarlo para fines didacticos en forma personal,
 * pero no esta permitido transferirlo tal cual a estudiantes actuales o potenciales.
 */
package estructuras.lineales;

public class Vector<T> {
    public static final int INC = 10;
    private Object[] buffer;

    /**
     * Constructor que crea un <code>Vector</code> con capacidad inicial INC.
     */
    public Vector() {
        buffer = new Object[INC];
    }

    /// METODOS DE ACCESO

    /**
     * Devuelve el elemento almacenado en la posicion <code>i</code>.
     * @param i el indice del objeto a recuperar.
     * @return el elemento almacenado en la posicion <code>i</code>.
     * @throws IndexOutOfBoundsException si
     *         <code>!(0 &lt;= i &lt; this.leeCapacidad()) </code>.
     */
    public T lee(int i) {

    }

    /**
     * Devuelve la capacidad actual de este <code>Vector</code>.
     * @return la capacidad actual del <code>Vector</code>.
     */
    public int leeCapacidad() {

    }

    /// METODOS DE MANIPULACIoN

    /**
     * Almacena el elemento <code>e</code> en la posicion <code>i</code>.
     * @param i el indice en el cual <code>e</code> sera almacenado.
     *          Debe cumplirse <code>0 &lt;= i &lt; this.leeCapacidad() </code>.
     * @param e el elemento a almacenar.
     * @throws IndexOutOfBoundsException si
     *         <code>!(0 &lt;= i &lt; this.leeCapacidad()) </code>.
     */
    public void asigna(int i, T e) {

    }

    /**
```

```
55        * Asigna la capacidad del <code>Vector</code>.
56        * Si <code>n &lt; this.leeCapacidad()</code> los elementos de
57        * <code>n</code> en adelante son descartados.
58        * Si <code>n &gt; this.leeCapacidad()</code> se agregan <code>null</code>
59        * en los espacios agregados.
60        * @param n la nueva capacidad del <code>Vector</code>, debe ser mayor que
61        *          cero.
62        * @throws IllegalSizeException si <code>n &lt; 1</code>.
63        */
64       public void asignaCapacidad(int n) {
65
66       }
67
68       /**
69        * Garantiza que el <code>Vector</code> cuente al menos con capacidad para
70        * almacenar <code>n</code> elementos.
71        * Si <code>n &gt; this.leeCapacidad()</code> la capacidad del
72        * <code>Vector</code> es incrementado de tal modo que el requerimiento
73        * sea satisfecho con cierta holgura.
74        * @param n capacidad minima que debe tener el <code>Vector</code>,
75        *          no puede ser menor a cero.
76        */
77       public void aseguraCapacidad(int n) {
78
79       }
80   }
```

Actividad 1

Revisa la documentación de la clase `Vector` de `Java`. ¿Cuáles serían los métodos equivalentes a los definidos aquí? ¿En qué difieren?

3.1. Compilando con ant

El código en este curso será editado en `Emacs` y será compilado con `ant`. El paquete para esta primera práctica incluye un archivo `ant` con las instrucciones necesarias.

Actividad 2

Abre una consola y cambia el directorio de trabajo al directorio que contiene a `src`. Intenta compilar el código utilizando el comando:

```
1    $ ant compile
```

Aparecerán varios errores pues el código no está completo.

Actividad 3

Consigue que la clase compile. Agrega los enunciados `return` que hagan falta, aunque sólo devuelvan `null` ó `0`. Tu clase no ejecutará nada útil, pero será sintácticamente correcta. Por ejemplo, puedes hacer esto con el método `lee`:

```
1    public T lee(int i) {
2      return null;
3    }
```

Al invocar `ant compile` ya no deberá haber errores y el directorio `build` habrá sido creado. Dentro de `build` se encuentran los archivos `.class`.

Actividad 4

Intenta compilar el código utilizando el comando:

```
1    $ ant
```

Esto intentará generar una distribución de tu código, pero para ello es necesario que pase todas las pruebas de `JUnit`, así que de momento te indicará que éstas fallaron. Para ejecutar únicamente las pruebas puedes llamar:

```
1    $ ant test
```

Esta tarea genera reportes en el directorio `reportes` donde puedes revisar los detalles sobre la ejecución de las pruebas, particularmente cuáles fallaron.

Para cuando termines esta práctica `ant` habrá creado el directorio `dist/lib`. Éste contendrá al archivo `Estructuras-<timestamp>.jar`. Si fueras a distribuir tu código, éste es el archivo que querrías entregar. Para los fines de este curso, más bien querremos el código fuente.

Actividad 5

Cuando comentas tu código siguiendo el formato de `javadoc` es posible generar automáticamente la documentación de tus clases en formato `html`. Ejecuta la tarea:

```
1    $ ant docs
```

Esto creará el directorio `docs`, con la documentación.

Actividad 6

Para remover todos los archivos que fueron generados utiliza:

```
1    $ ant clean
```

Asegúrate de ejecutar esta tarea antes de entregar tu práctica. Incluso remueve los archivos de respaldo de `emacs`. Ojo, no remueve los que llevan `#`. Puedes remover estos a mano o intenta modificar el archivo `build.xml` para que también los elimine, guíate por lo que ya está escrito.

---

4. DESARROLLO

Agrega el código necesario para que los métodos funcionen según indica la documentación. Cada vez que programes alguno asegúrate de que pase sus pruebas correspondientes de `JUnit`.

1. Obseva que los métodos `lee` y `asigna` deben ser programados para pasar cualquier prueba, pues son los métodos de acceso a la información, sin los cuales no es posible probar a los demás. Inicia con éstos.

   Para el método `lee` observa que tu clase promete entregar un objeto de tipo `T`, pero el arreglo interno (el atributo `buffer`) contiene `Object`. Desafortunadamente el sistema de tipos de `Java` no permite crear arreglos de un tipo genérico. Por ello será necesario utilizar un casting de la forma siguiente:

   ```
   T e = (T)(buffer[i]);
   ```

   Asegúrate de únicamente realizar este casting cuando estés seguro de que el objeto es de tipo `T`, de lo contrario `Java` te lo creerá, compilará correctamente, ejecutará el código y **cualquier cosa puede pasar**, desde excepciones tipo `ClassCastException` y errores extraños hasta que nunca se de cuenta.

2. En el método `asignaCapacidad` debes copiar los elementos del arreglo original cuando cambies el `buffer`. Por intereses académicos, es necesario que realices esta tarea con un ciclo, sin ayuda del API de `Java`. TIP: recuerda utilizar una **variable local** para referirte al arreglo recién creado, al final actualiza la **variable de clase**.

3. Para el método `aseguraCapacidad` calcula una fórmula que te permita cumplir con la condición indicada en el tamaño ($2^i inc > n'$). Puedes utilizar las funciones `Math.log`, `Math.pow` y `Math.ceil` para realizar el cálculo, utiliza castings a `int` cuando sea necesario.

5. PREGUNTAS

1. ¿Cuál fue la fórmula que utilizaste para calcular nn en el caso en que es necesario redimensionar el arreglo?

2. ¿Cuál es el peor caso en tiempo de ejecución para la operación `aseguraCapacidad`? Explique.

3. ¿Qué problema se presenta si, después de haber incrementado el tamaño del arreglo en varias ocasiones, el usuario remueve la mayoría de los elementos del `Vector`, quedando un gran espacio vacío al final? ¿Cómo lo resolverías?

6. FORMA DE ENTREGA

- Asegúrense de borrar todos los archivos generados, incluyendo archivos de respaldo usando `ant clean`.

- Copien el directorio `Practica1` dentro de un directorio llamado `<apellido1_apellido2>` donde `apellido1` es el primer apellido de un miembro del equipo y `apellido2` es el primer apellido del otro miembro. Por ejemplo: `marquez_vazquez`.
- Borren el directorio libs en la copia.
- Agreguen un archivo `reporte.pdf` dentro del directorio `Practica1` de la copia, con el nombre completo de los integrantes del equipo, las repuestas a las preguntas de la sección anterior y cualquier comentario que quieran hacer sobre la práctica.
- Compriman el directorio `<apellido1_apellido2>` creando `<apellido1_apellido2>.tar.gz`. Por ejemplo: `marquez_vazquez.tar.gz`.
- Suban este archivo en la sección correspondiente del aula virtual. Basta una entrega por equipo.

7. Forma de evaluación

Para su calificación final se tomarán en cuenta los aspectos siguientes:

70 % Calificación generada por las pruebas automáticas. Usaremos nuestros archivos, por lo que si realizan modificaciones a sus copias, éstas no tendrán efecto al momento de calificar.

15 % Documentación completa y adecuada. Entrega en el formato requerido, sin archivos `.class`, respaldos, bibliotecas de `JUnit` u otros no requeridos.

15 % Revisión manual del código, para verificar que se cumpla con las especificaciones, que no se haya copiado, etcétera.