2 | Vector

Meta

Que el alumno domine el manejo de información almacenada arreglos.

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.

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 del 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 para la implementación subyacente 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 2.1: Vector

Un *Vector* es una estructura tal que:

1. Puede almacenar n elementos de tipo T.
2. A cada elemento almacenado e 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) → 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. Nota: dado que el elemento asociado al índice es único, cualquier elemento que hubiera estado asociado a i deja de estarlo.

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'.

- Los elementos con índices i para $\mathrm{i} < \mathrm{n}'$ continúan almacenados en sus posiciones originales.
- Si $\mathrm{n}' < \mathrm{n}$ los elementos almacenados en $[\mathrm{n}', \mathrm{n}-1]$ son eliminados.
- Si $\mathrm{n}' > \mathrm{n}$ a los índices $[\mathrm{n}, \mathrm{n}'-1]$ se les asigna null.

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

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

Postcondiciones:

- Si $\mathrm{n}' < \mathrm{n}$ no pasa nada.
- Si $\mathrm{n}' > \mathrm{n}$: sea $\mathrm{nn} = 2^{\mathrm{i}}\mathrm{inc}$ tal que $\mathrm{nn} > \mathrm{n}'$, a n se le asigna el valor de nn.
  El significado de esta fórmula surge de la heurística siguiente: habrá menos cambios de tamaño si, cada vez que se requiere más espacio, se duplica el tamaño actual. Esta fórmula surge entonces de duplicar virtualmente el tamaño del arreglo tantas veces como sea necesario hasta que el índice n' quepa en el arreglo. Este método garantiza que hay al menos n' lugares en this.
- Los elementos con índices i para $\mathrm{i} < \mathrm{n}'$ continúan almacenados en sus posiciones originales.
- A los índices $[\mathrm{n}, \mathrm{nn}-1]$ se les asigna null.

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 la clase que le corresponde, mientras que los detalles de implementación se vuelven privados. El esqueleto de la clase contenido en el archivo `src/ed/lineales/Vector.java` corresponde a esta definición. Observa 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.

Actividad 2.1

Abre el archivo `Vector.java` dentro del directorio `src/ed/lineales`. Observarás que ya contiene el esqueleto de una clase: su declaración, un par de atributos y las declaraciones de varios métodos, aunque sólo el constructor está implementado. Observa el uso de `Arrays.copyOf()` para crear el arreglo de tipo genérico. Este es el único caso donde podrás utilizar la clase `Arrays` **en todo el curso**, por motivos didácticos todas las demás funciones con arreglos las debes programar a mano. Lee con cuidado la documentación de esta clase porque tu trabajo será completar su código.

También observa el estándar de la documentación, pues así tendrás que entregar tu código documentado en las próximas prácticas.

Abre también el archivo `UsoVector`, es un ejemplo de cómo se deben poder utilizar los objetos de tipo `Vector` una vez que el código esté completo.

Actividad 2.2

Revisa la documentación de la clase `Vector` de `Java`.

Entregable: ¿Cuáles serían los métodos equivalentes a los definidos aquí? ¿En qué difieren?

Compilación y prueba

Deberás trabajar el código en algún editor de texto o código como `Emacs`, `Kate` o similares. No es recomendable utilizar IDEs por el momento. El paquete para esta práctica incluye un archivo `build.xml` con las instrucciones necesarias para compilarlo usando `ant`.

Actividad 2.3

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 como el siguiente pues el código no está completo.

```
[javac] /media/blackzafiro/74e557c6-b247-4ff8-bf25-9e3d48232f48/
   ↪ home/blackzafiro/Documentos/Demos/ed-vector-demo/src/ed/
   ↪ lineales/Vector.java:42: error: missing return statement
    [javac]     }
    [javac]     ^
```

Estos reportes te ayudarán a completar la actividad siguiente. Estos errores aparecen porque las firmas de los métodos que no tienen tipo de regreso `void`, le indican al compilador que debería haber una instruccción `return` devolviendo un valor de ese tipo, pero la instrucción aún no está ahí.

Actividad 2.4

Consigue que la clase compile. Agrega los enunciados `return` que hagan falta en los métodos cuyo tipo de regreso **no sea** `void`, 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 2.5

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 (una pasa sólo por coincidencia). 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 ver a `UsoVector` en acción puedes ejecutar:

```
1 $ ant run
```

Cuando tu código esté completo y funcione se imprimirá el contenido del vector ejemplo en cada posición. Por ahora, si bien se ejecuta de principio a fin, no imprimirá nada hasta que implementes los métodos de la clase `Vector`.

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 2.6

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

```
$ ant docs
```

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

Actividad 2.7

Para remover todos los archivos que fueron generados utiliza:

```
$ ant clean
```

Asegúrate de ejecutar esta tarea antes de entregar tu práctica. Incluso remueve los archivos de respaldo de editores como `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.

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.

2. En el método `asignaCapacidad` debes copiar los elementos del arreglo original a uno nuevo más grande cuando necesites un `buffer` más grande. Por intereses académicos, es necesario que realices esta tarea con un ciclo, sin ayuda del API de `Java`; como es necesario crear un arreglo de tipo genérico y no es posible hacer esto directamente, utiliza `java.util.Arrays.copyOf`, pero debes utilizar al atributo `muestra` para que el arreglo esté vacío y tú lo tengas que llenar. TIP: recuerda utilizar una *variable local* para referirte al arreglo recién creado, al final actualiza el *atributo del objeto* `buffer` para apuntar al arreglo nuevo, el otro se lo puede llevar el recolector de basura.

3. Para el método `aseguraCapacidad` se puede calcular una fórmula que te permite cumplir con la condición indicada en el tamaño ($2^{\mathfrak{i}}\mathfrak{inc} > \mathfrak{n}'$). Sea la nueva longitud $\mathfrak{nn}$ el resultado de duplicar el tamaño del arreglo hasta que sea mayor a $\mathfrak{n}'$, entonces necesitamos calcular la potencia $\mathfrak{i}$ para la cual se cumple la condición:

$$
\begin{aligned}
nn = 2^i inc &> n' \\
\log_2(2^i inc) &> \log_2(n') \\
\log_2(2^i) + \log_2(inc) &> \log_2(n') \\
i &> \log_2(n') - \log_2(inc) \\
i &= \lceil \log_2(n') - \log_2(inc) \rceil
\end{aligned}
$$

Para calcularla en `Java` puedes utilizar las funciones `Math.log`, `Math.pow` y `Math.ceil` para realizar el cálculo, utiliza castings a `int` cuando sea necesario. Obseva que esta fórmula permite calcular el tamaño deseado en tiempo constante, si duplicaras la longitud en un ciclo hasta cumplir la condición, la complejidad sería $\log_2(n')$.

Preguntas

1. Explica ¿cuál es el peor caso en tiempo de ejecución para la operación `aseguraCapacidad`?

2. ¿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?