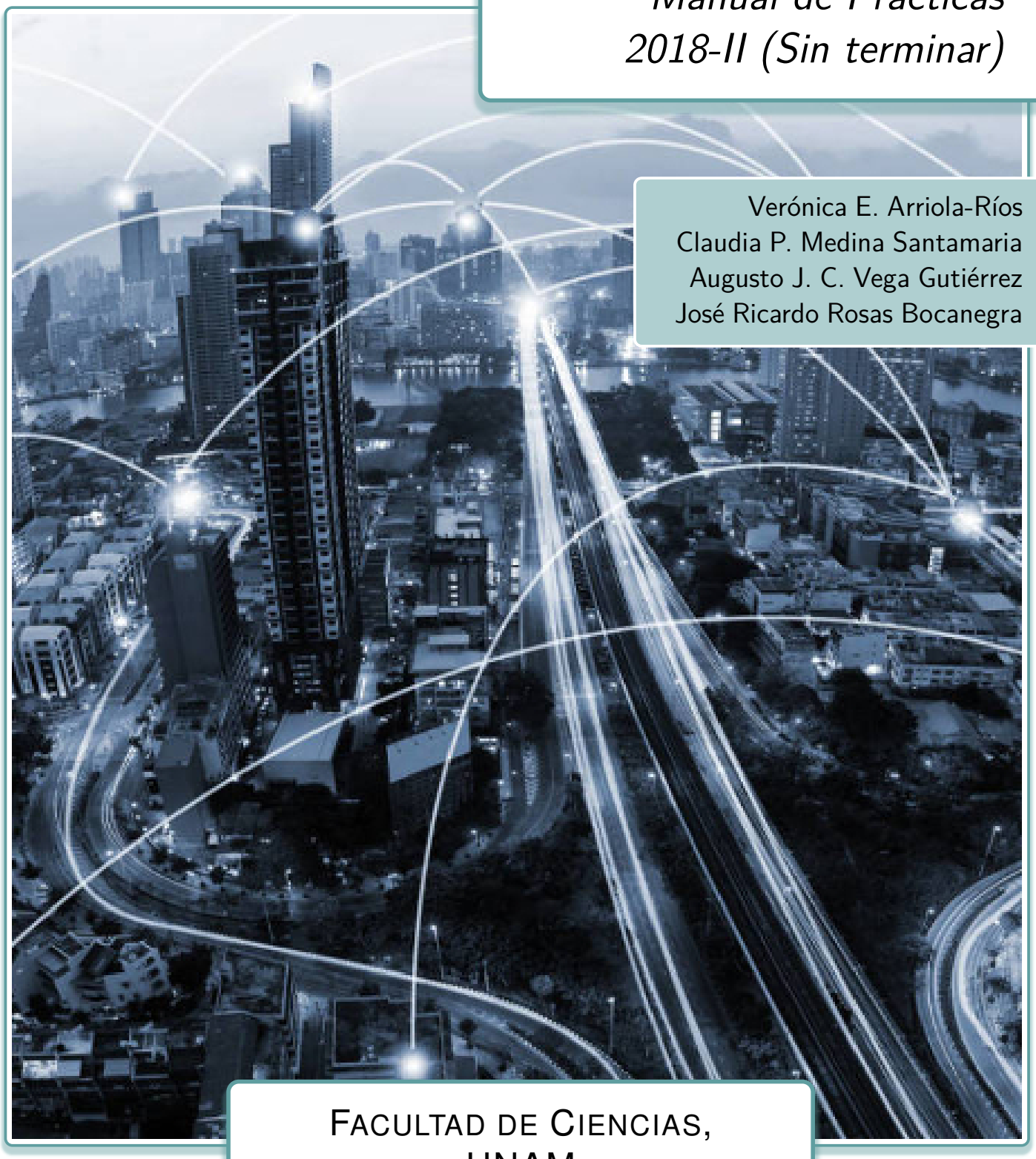
Estructuras de Datos

Manual de Prácticas
2018-II (Sin terminar)



Verónica E. Arriola-Ríos
Claudia P. Medina Santamaria
Augusto J. C. Vega Gutiérrez
José Ricardo Rosas Bocanegra

Facultad de Ciencias,
UNAM

Índice general

Parte I
Prácticas

1 | 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 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 1.1: 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. 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'.

- 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 $n' > n$: 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 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.

Actividad 1.1

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

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 1.2

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 1.3

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 1.4

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 1.5

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 1.6

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 `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.

   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 ($2^{\mathrm{inc}} > \mathrm{n}'$). Puedes utilizar las funciones `Math.log`, `Math.pow` y `Math.ceil` para realizar el cálculo, utiliza castings a `int` cuando sea necesario.

Preguntas

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

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

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?

2 | Complejidad

Meta

Que el alumno visualice el concepto de "función de complejidad computacional".

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Medir la complejidad en número de operaciones de un método de manera experimental.
- Comparar el desempeño entre las versiones iterativas y recursivas de un método.

Antecedentes

Sucesión de Fibonacci.

La sucesión de fibonacci fue descubierta por Fibonacci en relación a un problema de conejos. Supongamos que se tiene una pareja de conejos y cada mes esa pareja cría una nueva pareja. Después de dos meses, la nueva pareja se comporta de la misma manera. Entonces, el número de parejas nuevas nacidas a_n en el n-ésimo mes es $a_{n-1} + a_{n-2}$, ya que nace una pareja por cada pareja nacida en el mes anterior y cada pareja nacida hace dos meses cria una nueva pareja. Por convención consideremos $a_0 = 0$ y $a_1 = 1$.

Actividad 2.1

Define, con estos datos, la función de fibonacci.

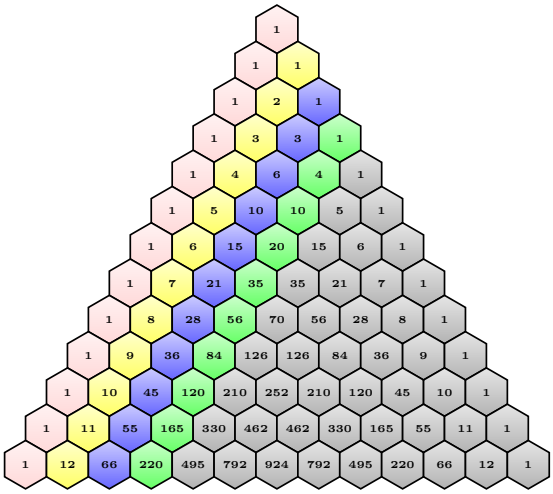


Figura 2.1 Triángulo de Pascal. Autor:M.H. Ahmadi

Triángulo de Pascal.

En la figura Figura 2.1 se muestran algunos términos del Triángulo de Pascal.

Matemáticamente, podemos definir el elemento $Pascal_{ij}$ que corresponde al elemento en la fila i, columna j de la siguiente manera:

$$Pascal_{ij} = \begin{cases} 1 & si\ j = 0\ ó\ j = i \\ Pascal_{(i-1)(j-1)} + Pascal_{(i-1)(j)} & \text{En cualquier otro caso.} \end{cases} \tag{2.1}$$

Desarrollo

La práctica consiste en implementar métodos que calculen el triángulo de pascal y el n-ésimo número de fibonacci, al tiempo que estiman el número de operaciones realizadas. Esto se realizará en forma recursiva e iterativa. Se deberá implementar la interfaz `IComplejidad` en una clase llamada `Complejidad`. Se entregan pruebas unitarias para ayudar a verificar que estas funciones estén bien implementadas. Adicionalmente deberán llevar la cuenta del número de operaciones estimadas en un atributo de la clase para generar un reporte ilustrado sobre el número de operaciones que realiza cada método.

GNUPLOT

Gnuplot es una herramienta interactiva que permite generar gráficas a partir de archivos de datos planos. Para esta práctica, los datos deben ser guardados en un archivo de este tipo y graficados con gnuplot. Supongamos que el archivo donde se guardan es llamado **datos.dat**.

Por ejemplo, para el método de Fibonacci el archivo **datos.dat** tendría algo semejante al siguiente contenido:

Listing 2.1: data/FibonacciIt.dat

```
0       1
1       1
2       2
3       3
4       4
5       5
```

donde la primer columna es el valor del argumento y la segunda, el número de operaciones.

Para el método de Pascal el archivo **datos.dat** tendría algo semejante al siguiente contenido:

Listing 2.2: data/PascalRec.dat

```
0       0       2

1       0       2
1       1       2

2       0       2
2       1       4
2       2       2

3       0       2
3       1       6
3       2       6
3       3       2
```

donde la primer columna es el valor del renglón, la segunda es la columna, y la tercera el número de operaciones. Observa que cada vez que cambies de renglón, debes dejar una linea en blanco para indicarle a `gnuplot` cuándo cambia el valor en el eje x.

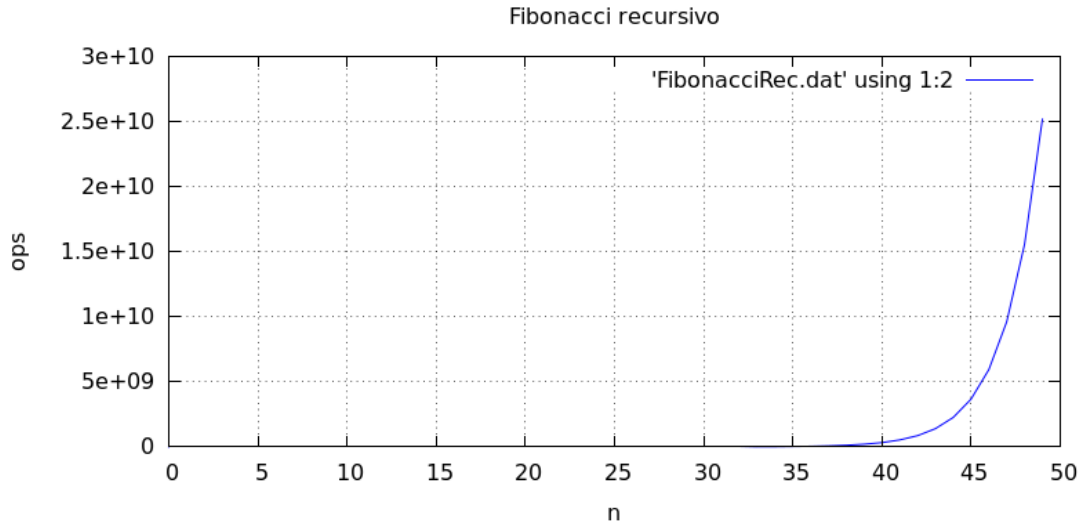


Figura 2.2 Complejidad en tiempo al calcular el n-ésimo coeficiente de la serie de Fibonacci en forma recursiva.

Gráficas en 2D

Al iniciar el programa `gnuplot` aparecerá un promt y se puede iniciar la sesión de trabajo. A continuación se muestra cómo crear una gráfica 2D. Deberás obtener algo como la Figura 2.2.

```
gnuplot> set title "Mi gráfica"      //Título para la gráfica
gnuplot> set xlabel "Eje X: n"       //Título para el eje X
gnuplot> set ylabel "Eje Y: ops"     //Título para el eje Y
gnuplot> set grid "front";           // Decoración
gnuplot> plot "datos.dat" using 1:2 with lines lc rgb 'blue' //
    graficamos los datos
gnuplot> set terminal pngcairo size 800,400  //algunas caracterí
    sticas de la imagen que se guardará
gnuplot> set output 'fib.png'        //nombre de la imagen que se
    guardará
gnuplot> replot                      //lo graficamos para que se
    guarde en la imagen
```

Gráficas en 3D

A continuación se muestra cómo crear una gráfica 3D. Observa que, en este caso, el archivo de datos requiere tres columnas. Deberás obtener algo como la Figura 2.3.

```
gnuplot> set title "Mi gráfica"
gnuplot> set xlabel "Eje X"
```

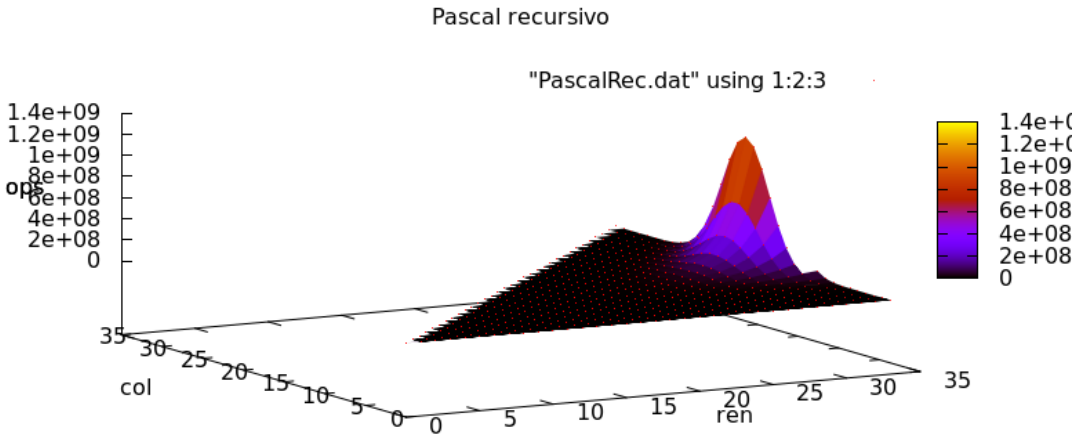


Figura 2.3 Complejidad en tiempo al calcular el coeficiente del triángulo de Pascal para el renglón y columna dados.

```
gnuplot> set ylabel "Eje␣Y"
gnuplot> set zlabel "Eje␣Z"
gnuplot> set pm3d
gnuplot> splot "datos.dat" using 1:2:3 with dots
gnuplot> set terminal pngcairo size 800,400
gnuplot> set output 'pascal.png'
gnuplot> replot
```

Ejercicios

1. Crea la clase `Complejidad`, que implemente `IComplejidad`. Agrega las firmas de los métodos requeridos y asegúrate de que compile, aunque aún no realize los cálculos.

2. Programa los métodos indicados en la interfaz. Las pruebas unitarias te ayudarán a verificar tus implementaciones de Fibonacci y Pascal. Compila tu código utilizando el comando `ant` en el directorio donde se encuentra el archivo `build.xml`, si compila correctamente las pruebas se ejecutarán automáticamente.

   En particular, nota que los métodos estáticos se implementan en la interfaz, sólo son auxiliares para escribir los datos en el archivo. El archivo se debe abrir para agregar (modo *append*), de modo que los datos se acumulen entre llamadas sucesivas al método, revisa la documentación de `PrintStream` y `FileOutputStream`, te ayudarán mucho en esta parte.

3. Agrégale un atributo a la clase para que cuente lo siguiente:

Iterativos El número de veces que se ejecuta el ciclo más anidado. Observa que puedes inicializar el valor del atributo auxiliar al inicio del método y después incrementarlo en el interior del ciclo más anidado.

Recursivos El número de veces que se manda llamar la función. Aquí utilizarás una técnica un poco más avanzada que sirve para optimizar varias cosas. Necesitarás crear una función auxiliar (*privada*) que reciba los mismos parámetros. En la función original revisarás que se cumplan las precondiciones de los datos e inicializarás la variable que cuenta el número de llamadas recursivas. La función auxiliar es la que realmente realizará la recursión. Ya no revises aquí las precondiciones, pues ya sólo depende de ti garantizar que no la vas a llamar con parámetros inválidos. Incrementa aquí el valor del atributo contador, deberá incrementarse una vez por cada vez en que mandes llamar esta función.

A continuación se ilustra la idea utilizando la función factorial: (OJO: tu código no es igual, sólo se ilustra el principio).

```
1  /** Ejemplo de cómo contar el número de llamadas a la
2   * implementación recursiva de la función factorial. */
3  public class ComplejidadFactorial {
4
5    /* Número de operaciones realizadas en la última
6     * llamada a la función. */
7    private long contador;
8
9    /** Valor del contador de operaciones después de la ú
       ltima
10    *  llamada a un método. */
11   public long leeContador() {
12     return contador;
13   }
14
15   /** n! */
16   public int factorial(int n) {
17     contador = 1;
18     if (n < 0) throw new IndexOutOfBoundsException();
19     if (n == 0) return 1;
20     return factorialAux(n);
21   }
22
23   private int factorialAux(int n) {
24     operaciones++;
25     if (n == 1) return 1;
26     else return factorialAux(n - 1);
27   }
28
29   /** Imprime en pantalla el número de llamadas a la funci
       ón para
30    *  varios parámetros. */
31   public static void main(String[] args) {
```

```
32      ComplejidadFactorial c = new ComplejidadFactorial();
33      for(int n = 0; n < 50; n++) {
34        int f = c.factorial(n);
35        System.out.format("Para␣n=%d␣se␣realizaron␣%d␣
              operaciones",
36                          n, c.leeContador());
37      }
38    }
39  }
```

4. Crea un método `main` en una clase code `UsoComplejidad` que mande llamar los métodos programados para diferentes valores de sus parámetros y que guarde los resultados en archivos de texto. Podrás ejecutarlo con el comando `ant run`.

5. Para el método de fibonacci, genera las gráficas n(entrada) vs número de operaciones y haz un análisis de lo que sucede. ¿Cuál es el orden de complejidad? Justifica.

6. Para el método recursivo del cálculo del triángulo de Pascal, genera una gráfica en 3-D en donde el parámetro renglón se encontrará en el eje X, el parámetro columna se encontrará en el eje Y, el número de operaciones en el eje Z y haz un análisis de lo que sucede. ¿Cuál es el orden de complejidad? Justifica.

7. Entrega tus resultados en un reporte en un archivo `.pdf`, junto con tu código limpio y empaquetado.

Preguntas

1. ¿Cuál es el máximo valor de n que pudiste calcular para el factorial sin que se alentara tu computadora? (Puede variar un poco de computadora a computadora (± 3), así que no te esfuerces en encontrar un valor específico).

2. ¿Cuál es el máximo valor de ren que pudiste calcular para el triángulo de Pascal sin que se alentara tu computadora?

3. Justifica a partir del código ¿cuál es el orden de complejidad para cada uno de los métodos que programaste?

4. Escribe un reporte con tus gráficas generadas y las respuestas a las preguntas anteriores.

3 | Polinomio de direccionamiento

META

Que el alumno domine el manejo de información almacenada en arreglos multidimensionales.

OBJETIVOS

Al finalizar la práctica el alumno será capaz de:

- Almacenar un arreglo de n dimensiones en uno de una sola dimensión.

ANTECEDENTES

Vectores de Iliffe

Los arreglos multidimensionales son aquellos que tienen más de una dimensión, los más comunes son de dos dimensiones, conocidos como matrices. Este tipo de arreglos en Java se ven como arreglos de arreglos, a los cuales se llama *vectores de Iliffe*.

Existen dos momentos fundamentales en la creación de arreglos:

1. Declaración: en esta parte no se reserva memoria, solo se crea una referencia.

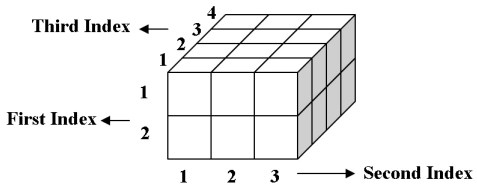
```
1 int [][] arreglo;
2 float[][][] b;
```

2. Reservación de la memoria y la especificación del número de filas y columnas.

```
1 //matriz con 10 filas y 5 columnas
```

```
arreglo = new int[10][5];

//cubo con 13 filas, 25 columnas y 4 planos
b = new float [13][25][4];
```



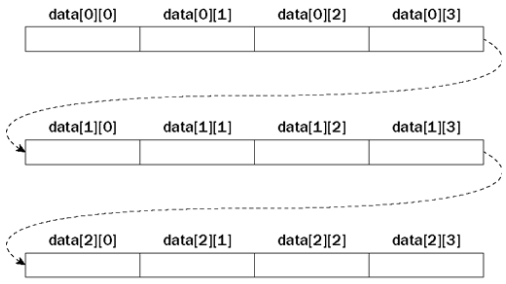
Three-dimensional array with twenty four elements

Nota: Se puede declarar una dimensión primero, pero siempre debe de ser en orden, por ejemplo `arreglo = new int[][10];` es erróneo pues las filas quedan indeterminadas. Esta libertad permite crear arreglos de forma irregular, como:

```
int[][][] arreglo = new int[3][][];
arreglo[0] = new int[2][];
arreglo[0][1] = {1,2,3};
arreglo[2] = new int[1][2];
// Se ve como:
// {{{1,2,3}, null}, null, {0,0}}
```

Polinomio de direccionamiento

Dado que la memoria de la computadora es esencialmente lineal es natural pensar en almacenar los elementos de un arreglo multidmensional en un arreglo de unidimensional. Por ejemplo, consideren un arreglo de tres dimensiones:



The array elements are stored in contiguous locations in memory.

Generalicemos el ejemplo anterior a uno de n-dimensiones, donde el tamaño de cada dimensión i esta dada por δ_i, entonces tenemos un arreglo n-D: $\delta_0 \times \delta_1 \times \delta_2 \times ... \times \delta_{n-1}$.

Entonces la posición del elemento $a[i_0][i_1][...][i_{n-1}]$ esta dada por:

$$p(i_0, i_1, ..., i_{n-1}) = \sum_{j=0}^{n-1} f_j i_j \tag{3.1}$$

donde

$$f_j = \begin{cases} 1 & \text{si } j{=}n-1 \\ \prod_{k=j+1}^{n-1} \delta_k & \text{si } 0 \leqslant j < n-1 \end{cases} \tag{3.2}$$

Desarrollo

La práctica consiste en implementar los métodos definidos en la interfaz `IArreglo`, la cual convierte un arreglo de enteros de n-dimensiones en uno de una dimensión. El constructor de la clase que implemente la interfaz deberá tener como parámetro un arreglo de `int`s que representen las dimensiones del arreglo. Por ejemplo para crear un arreglo tridimensional ($3 \times 10 \times 5$). Observa que entonces el número de dimensiones en la matriz estará dada por la longitud de este primer arreglo. Invocamos el constructor de la siguiente forma:

```
1  Arreglo a = new Arreglo(new int [] {3,10,5});
```

Observa que todas las dimensiones deben ser mayores que cero.

Preguntas

1. Explica la estructura de tu código, explica en más detalle tu implementación del método `obtenerIndice`.
2. ¿Cuál es el orden de complejidad de cada método?

4 | Colección abstracta

Meta

Que el alumno aplique la reutilización de código mediante el mecanismo de herencia e interfaces propuesto por el paradigma orientado a objetos en `Java`.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Escribir código general en una clase padre, sin conocer detalles sobre la implementación de las clases descendientes.
- Utilizar la definición, mediante una interfaz, de un tipo de dato abstracto, para programar funciones generales, sin conocer detalles sobre la implementación de los objetos que utiliza.

Antecedentes

Para reducir la cantidad de trabajo en las prácticas siguientes, se utilizarán las ventajas del paradigma orientado a objetos. Concretamente, se programará una biblioteca con varias estructuras de datos y las operaciones comunes a todas ellas serán implementadas en una clase padre. En esta práctica se trata de completar tantos métodos como sea posible programar eficientemente, aún sin haber programado ninguna de esas estructuras.

Para adquirir algo de habilidad creando código profesional, este paquete trabajará cumpliendo un conjunto de especificaciones dictadas por la API[1] de `Java`.

[1] Interfaz de programación de aplicaciones.

Actividad 4.1

Para familiarizarte con el ambiente de trabajo, revisa la documentación de las clases `Collection<E>` e `Iterator<E>` de `Java`.

Todas las estructuras que programaremos implementarán `Collection<E>`. No te preocupes, no duplicaremos la labor de las estructuras que ya vienen programadas en `Java`. La mayoría de nuestras estructuras ofrecerán características distintas a las versiones de la distribución oficial. Más aún, por motivos didácticos y ligeramente nacionalistas, nuestras clases tendrán nombres en español[2].

Actividad 4.2

Revisa la documentación del paquete `java.util`. ¿Qué estructuras de datos encuentras incluidas?

La primer clase a programar de nuestro paquete se llama `ColeccionAbstracta<E>` e implementa la intefaz `Collection<E>`. Todas nuestras estructuras heredarán de ella, por lo que el trabajo de esta práctica nos ahorrará mucho código en las venideras. Como `ColeccionAbstracta<E>` no sabe aún cómo serán guardados los datos, no podrá implementar todos los métodos de `Collection<E>`; de ahí que será de tipo `abstracto`. Para poder trabajar, hará uso del único conocimiento que tiene de estructuras tipo `Collection<E>`: que todas ellas implementan el método `iterator()`, que devuelve un método de tipo `Iterator<E>`.

Dado que el iterador recorre la estructura (sea cual sea ésta), otorgando acceso a cada uno de sus elementos una única vez, es posible implementar varios de los métodos de `Collection<E>` haciendo uso de este objeto.

Desarrollo

1. Crea una clase llamada `ColeccionAbstracta<E>` que implemente la interfaz `Collection<E>`, dentro del paquete `estructuras`.
2. Implementa únicamente los métodos listados a continuación. Una sugerencia es que añadas todas las firmas de los métodos y hagas que devuelvan `0` o `null` para verificar que tu clase compile. Observa que la clase `Conjunto<E>` fue provista como ejemplo de clase hija. Una vez que agregues los métodos, `Conjunto<E>` deberá compilar sin problemas, esto será necesario para que las pruebas unitarias funcionen.
   - `public boolean contains(Object o)`

[2]Aunque los métodos seguirán teniendo nombres en inglés, pues así lo requieren las interfaces.

- `public Object[] toArray()`
- `public <T> T[] toArray(T[] a)`
- `public boolean containsAll(Collection<?> c)`
- `public boolean addAll(Collection<? extends E> c)`
- `public boolean remove(Object o)`
- `public boolean removeAll(Collection<?> c)`
- `public boolean retainAll(Collection<?> c)`
- `public void clear()`

3. Adicionalmente, sobreescribe el método `toString()` de la superclase `Object` para que la colección devuelva una cadena con todos los elementos almacenados en ella. Te será muy útil en el futuro para depurar tus colecciones mientras las programas.

Preguntas

1. ¿Qué estructuras de datos incluye la API de `Java` dentro del paquete que importas, `java.util`?

2. ¿Cuál crees que es el objetivo de la interfaz `Collection`? ¿Por qué no hacer que cada estructura defina sus propios métodos?

3. ¿Qué métodos permite la interfaz `Collection` que su funcionalidad sea opcional? ¿Qué deben hacer estos métodos opcionales si no se implementa su funcionalidad? ¿Por qué crees que son opcionales?

5 | Pila con referencias

Meta

Que el alumno domine el manejo de información almacenada en una *Pila*.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Implementar el tipo de dato abstracto *Pila* utilizando nodos y referencias.

Antecedentes

Una `Pila` es una estructura de datos caracterizada por:

1. El último elemento que entra a la `Pila` es el primer elemento que sale.
2. Tiene un tamaño dinámico.

A continuación se define el tipo de dato abstracto `Pila`.

Definición 5.1: Pila

Una *Pila* es una estructura de datos tal que:

1. Tiene un número variable de elementos de tipo T.
2. Cuando se agrega un elemento, éste se coloca en el tope de la `Pila`.
3. Sólo se puede extraer al elemento en el tope de la `Pila`.

Nombre: $\mathtt{Pila}$.
Valores: $\mathbb{N}$, T, con $\mathtt{null} \in$ T.

Figura 5.1 Representación en memoria de una pila, utilizando nodos y referencias.

Operaciones: sea `this` la pila sobre la cual se está operando.

Constructores :

Pila(): $\emptyset \to \text{Pila}$

Precondiciones: $\emptyset$

Postcondiciones :

- Una Pila vacía.

Métodos de acceso :

mira(this) $\to$ **e:** $\text{Pila} \times \mathbb{N} \to \text{T}$

Precondiciones: $\emptyset$

Postcondiciones :

- $e \in \text{T}$, e es el elemento almacenado en el tope de la Pila.

Métodos de manipulación :

expulsa(this) $\to$ **e:** $\text{Pila} \to \text{T}$

Precondiciones :$\emptyset$

Postcondiciones :

- Elimina y devuelve el elemento e que se encuentra en el tope de la Pila.

empuja(this, e): $\text{Pila} \times \text{T} \xrightarrow{?} \emptyset$

Precondiciones: $\emptyset$

Postcondiciones :

- El elemento e es asignado al tope de la Pila.

Actividad 5.1

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

Como se ilustra en la Figura 5.1, en esta implementación los datos se guardan dentro de objetos llamados *nodos*. Cada nodo contiene dos piezas de información:

- El dato[1] que guarda y
- la dirección del nodo con el siguiente dato.

Una clase, a la cual nosotros llamaremos `PilaLigada<E>`, tiene un atributo escencial:

[1] O la dirección del dato, si se trata de un objeto.

- La dirección del primer nodo, es decir, del nodo con el último dato que fue agregado a la pila.

Cada vez que se quiera empujar un dato a la pila, se creará un nodo nuevo para guardar ese dato. El nuevo nodo también almacenará la dirección del nodo que solía estar a la *cabeza* de la estructura, y la variable `cabeza` ahora tendrá la dirección de este nuevo nodo. Imaginemos que el nuevo dato acaba de *sumergir* un poco más a los datos anteriores (los empujó más lejos). Esos datos no volverán a ser visibles, hasta que el dato en la cabeza haya sido expulsado.

Para expulsar un dato se realiza el procedimiento inverso: la cabeza volverá a guardar la dirección del nodo siguiente y se devolverá el valor que estaba guardado en el nodo *de hasta arriba*, a la vez que se descarta el nodo que contenía al dato. Cuidado: al realizar estas operaciones en código es importante cuidar el orden en que se realizan, para no perder datos o direcciones en el proceso. A menudo requerirás del uso de variables temporales, para almacenar un dato que usarás depués. Pero recuerda: las variables temporales deben desaparecer cuando se termina la ejecución de un método, es decir, deben ser variables locales. Asegúrate de guardar todo lo que deba permanecer en la pila en atributos de objetos, ya sea en la `PilaLigada<E>` o algún `Nodo` adecuado.

Desarrollo

Se implementará el TDA `Pila` utilizando nodos y referencias. Para esto se deberá implementar la intefaz `IPila<E>` y extender `ColeccionAbstracta<E>`. Asegúrate de que tu implementación cumpla con las condiciones indicadas en la documentación de la intefaz. Por razones didácticas, no se permite el uso de ninguna clase que se encuentre en el api de Java, por ejemplo clases como `Vector<E>`, `LinkedList<E>` o cualquiera otra estructura del paquete `java.util`.

1. Programa la clase `Nodo`..

   Puedes crear esta clase dentro del paquete `ed.estructuras.lineales`. Esto te permitirá reutilizarlo cuando programes la siguiente estructura: la cola. Si eliges esta opción, dale acceso de paquete (es decir, la declaración de la clase omite el acceso e inicia con `class Nodo<E>...` en lugar de `public class Nodo...`). Esto es para no confundir este nodo con otros nodos que utilizarán futuras estructuras y que tienen características diferentes. Otra opción es programarlo como una clase estática interna de `PilaLigada<E>`, pero en ese caso, sólo la pila podrá usarlo.

2. Programa la clase `PilaLigada<E>`.

   - Implementa la interfaz `IPila<E>` y
   - extiende `ColeccionAbstracta<E>`.

Inicia con los métodos básicos.

3. Luego agrega el iterador que requiere `Collection<E>`. Puedes programar al iterador como una clase interna, en este caso debes implementar la interfaz `Iterator<E>` pero no es necesario que tu clase declare una nueva variable de tipo, la `<E>`. Esto se vería así:

```
import java.util.Iterator;
...
public class PilaLigada<E> ... {
  private class Iterador implements Iterator<E> {
    ...
  }
}
```

Aunque en una pila sólo se pueden agregar y remover elementos en un extremo, necesitaremos un iterador que permitar ver todos los elementos en la pila, desde el último insertado hasta el primero. Para esta práctica sólo programarás un constructor, que inicialice el iterador en el tope de la pila, y los métodos `next` y `hasNext` del iterador.

Preguntas

1. Explica, para esta implementación, cómo funciona el método `empuja`.
2. ¿Cuál es la complejidad, en el peor caso, de los métodos `mira`, `expulsa` y `empuja`?

6 | Pila en arreglo

Meta

Que el alumno domine el manejo de información almacenada en una *Pila*.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Implementar el tipo de dato abstracto *Pila* utilizando un arreglo.
- Diferenciar entre distintos tipos de implementación para el tipo de dato abstracto *Pila*.

Antecedentes

Una `Pila` es una estructura de datos caracterizada por:

1. El último elemento que entra a la `Pila` es el primer elemento que sale.
2. Tiene un tamaño dinámico.

La definición del tipo de dato abstracto `Pila` utilizada en la práctica anterior es igualmente válida para la nueva implementación. La interfaz `IPila` que se debe implementar también es la misma. La diferencia radica en la forma en que serán almacenados los datos.

Desarrollo

Se implementará el TDA `Pila` utilizando arreglos. Para esto se utilizará la clase abstracta `ColeccionAbstracta` de la práctica anterior, que implementa algunos

métodos de la interfaz `Collection`.

Además de extender la clase abstracta, se debe implementar la interfaz `IPila`. Esto se deberá hacer en una clase `PilaArreglo`. Por razones didácticas, no se permite el uso de ninguna clase que se encuentre en el API de Java, por ejemplo clases como `Vector`, `ArrayList` o cualquiera que haga el manejo de arreglos dinámicos.

Preguntas

1. Explica, de acuerdo a cada una de tus implementaciones, cómo funciona el método `empuja`.
2. ¿Cuál es la complejidad, en el peor caso, de los métodos `mira`, `expulsa` y `empuja`?
3. ¿En qué escenarios conviene más usar una `PilaArreglo<E>`? ¿Cuándo es mejor una `PilaLigada<E>`? Justifica tus respuestas.

7 | Cola con referencias

Meta

Que el alumno domine el manejo de información almacenada en una *Cola*.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Implementar el tipo de dato abstracto *Cola* utilizando nodos y referencias.

Antecedentes

Una `Cola` es una estructura de datos caracterizada por:

1. Ser una estructura de tipo `FIFO`, esto es que el primer elemento que entra es el primero que sale.
2. Tener un tamaño dinámico.
3. Ser lineal.

A continuación se define el tipo de dato abstracto `Cola`.

Definición 7.1: Cola

Una *Cola* es una estructura de datos tal que:

1. Tiene un número variable de elementos de tipo T.
2. Mantiene el orden de los datos ingresados, permitiendo únicamente el acceso al primero y el último.
3. Cuando se agrega un elemento, éste se coloca al final de la `Cola`.

4. Cuando se elimina un elemento, éste se saca del inicio de la `Cola`.

Nombre: $\mathcal{V}ector$.
Valores: $\mathbb{N}$, T, con $\mathcal{n}ull \in$ T.
Operaciones:

Constructores :

Cola() : $\emptyset \rightarrow \mathcal{C}ola$
Precondiciones : $\emptyset$
Postcondiciones : Una $\mathcal{C}ola$ vacía.

Métodos de acceso :

mira(this) $\rightarrow$ **e:** $\mathcal{C}ola \rightarrow$ T
Precondiciones : $\emptyset$
Postcondiciones :
- $e \in$ T, e es el elemento almacenado al inicio de la $\mathcal{C}ola$.

Métodos de manipulación :

forma(this, e) : $\mathcal{C}ola \times$ T $\xrightarrow{?} \emptyset$
Precondiciones : $\emptyset$
Postcondiciones :
- El elemento e es asignado al final de la $\mathcal{C}ola$.

atiende(this) $\rightarrow$ **e:** $\mathcal{C}ola \rightarrow$ T
Precondiciones :$\emptyset$
Postcondiciones :
- Elimina y devuelve el elemento e que se encuentra al inicio de la $\mathcal{C}ola$.

Aunque esta es la definición teórica de la estructura y todas las colas tienen métodos que se comportan de esta manera, para esta práctica no utilizaremos los nombres de los métodos que corresponden a la definición. La API de `Java` ya incluye una interfaz para la estructura de datos `Cola` y esa es la que utilizaremos.

Actividad 7.1

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

Desarrollo

Se harán dos implementaciones para el TDA *Cola*: una con referencias y otra con arreglos. En esta práctica se realizará la implementación con referencias. De nuevo se

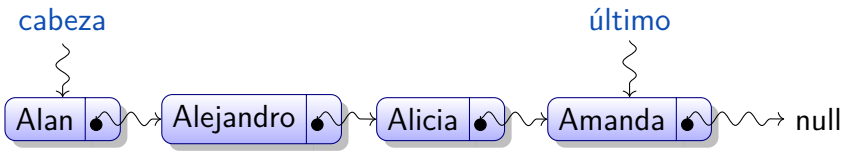


Figura 7.1 Representación en memoria de una cola, utilizando nodos y referencias.

hará una extención de la clase `ColeccionAbstracta`.

La implementación es análoga a la utilizada para la pila con referencias, con la diferencia de que necesitarás un atributo más en la clase cola: una referencia al último elemento de la estructura, pues ahí se formarán los nodos con los elementos entrantes (Figura 7.1).

1. Programa la clase `Nodo`. Si en la práctica sobre pilas creaste esta clase dentro del paquete `estructuras.lineales`, puedes utilizar la que ya tienes. Si elegiste programarlo como una clase estática interna de `PilaLigada`, puedes copiar y pegar el código dentro de tu clase `ColaLigada` y funcionará igual.

2. Programa la clase `ColaLigada`. No olvides implementar la interfaz `Queue<E>`. Inicia con los métodos básicos y luego agrega el iterador que requiere `Collection<E>`. Ojo, para esta implementación no hay límites en la capacidad de la cola, por lo que nunca se lanzan excepciones con los métodos `add`, `remove` y `element`.

Preguntas

1. Explica cómo funcionan los métodos `offer` y `poll`.

2. ¿Cuál es la complejidad de los métodos `peek`, `offer` y `poll`? Justifica tus respuestas.

8 | Cola en arreglo

Meta

Que el alumno domine el manejo de información almacenada en una *Cola*.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Implementar el tipo de dato abstracto *Cola* utilizando un arrelgo.
- Diferenciar entre distintos tipos de implementación para el tipo de dato abstracto *Cola*.

Antecedentes

Una `Cola` es una estructura de datos caracterizada por:

1. Ser una estructura de tipo `FIFO`, esto es que el primer elemento que entra es el primero que sale.
2. Tener un tamaño dinámico.
3. Ser lineal.

La definición del tipo de dato abstracto `Cola` utilizada en la práctica anterior es igualmente válida para la nueva implementación. La interfaz `Cola` que se debe implementar, también es la misma. La diferencia radica en la forma en que serán almacenados los datos. Para programar una cola en un arreglo, es necesario utilizar algunos trucos:

1. Se debe tener dos enteros indicando las posiciones del primer elemento (cabeza) y último elemento en la cola.

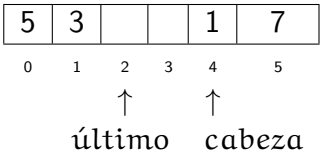


Figura 8.1 Cola en un arreglo, cuando ya se han eliminado elementos de la cabeza y se han formado elementos más allá de la longitud del buffer.

2. Los elementos se colocan a la derecha de la cabeza, módulo la longitud del arreglo, siempre que haya espacios disponibles. Si no hay espacios, se debe cambiar el arreglo por uno más grande.

3. Los elementos se remueven de la posición de la cabeza y el inidicador de esta se recorre a la casilla siguiente, a la derecha, módulo la longitud del arreglo. Un ejemplo se muestra en la Figura 8.1.

Desarrollo

En esta práctica se implementará la Cola utilizando arreglos. De nuevo se hará una extención de la clase `ColeccionAbstracta`. Por razones didácticas, no se permite el uso de ninguna clase que se encuentre en el API de Java, por ejemplo clases como `Vector`, `ArrayList` o cualquiera que haga el manejo de arreglos dinámicos.

1. Crea un constructor que reciba como parámetro un arreglo de tamaño cero, del mismo tipo que la clase. Utilizarás este arreglo para crear el buffer en forma genérica. También debe recibir un tamaño inicial para el buffer de tipo entero. El encabezado de tu constructor quedará:

```
1 public ColaArreglo(E[] a, int tamInicial);
```

2. Crea otro constructor que sólo reciba el arreglo. Asignarás un valor inicial para el buffer con un tamaño por defecto que tú puedes elegir. Puedes llamar a tu otro constructor para no repetir el trabajo:

```
1 public ColaArreglo(E[] a) {
2   this(a, DEFAULT_INITIAL_SIZE);
3 }
```

3. Programa el método para agregar un elemento, en forma semejante a como lo hiciste con la pila. Ojo: en este caso las dimensiones del arreglo no cambian si se

llega al final, es posible que haya espacios vacíos al inicio del arreglo y deberás reutilizarlos antes que cambiar el tamaño del arreglo. Así puedes ahorrar tiempo, al no copiar a todos al nuevo arreglo. Verifica que funcione.

4. Programa el método para sacar un elemento. Deberás ir recorriendo la cabeza según sea necesario, dejando y hueco a su izquierda. Verifica que funcione.

5. Continúa con los otros métodos.

Preguntas

1. ¿Qué método utilizas para detectar cuando la cola está vacía?

2. ¿Qué fórmula utilizas para detectar cuando el buffer de la cola está lleno?

3. ¿Cuál es la complejidad para el mejor y peor caso de los métodos `mira`, `forma` y `atiende`? Justifica.

9 | Lista doblemente ligada

Meta

Que el alumno domine el manejo de información almacenada en una *Lista*.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Visualizar cómo se almacena una lista en la memoria de la computadora mediante el uso de nodos con referencias a su elemento anterior y su elemento siguiente.
- Programar dicha representación en un lenguaje orientado a objetos.

Antecedentes

Definición 9.1

Una lista es:

$$\text{Lista} = \begin{cases} \text{Lista vacía} \\ \text{Dato seguido de otra lista} \end{cases}$$

Alternativamente:

Definición 9.2

Una **lista** es una secuencia de cero a más elementos **de un tipo determina-**

do (que por lo general se denominará tipo-elemento). Se representa como una sucesión de elementos separados por comas:

$$a_0, a_1, ..., a_{n-1}$$

donde $n \geqslant 0$ y cada a_i es del tipo **tipo-elemento**.

- Al número n de elementos se le llama *longitud* de la lista.
- a_0 es el *primer elemento* y a_{n-1} es el *último elemento*.
- Si $n = 0$, se tiene una **lista vacía**, es decir, que no tiene elementos. Aho, Hopcroft y Ullman 1983, pp. 427

En este caso utilizaremos como definición del tipo de datos abstracto, la interfaz definida por Oracle:

http://docs.oracle.com/javase/8/docs/api/java/util/List.html.

Actividad 9.1

Lee la definición de la interfaz `List<E>`. ¿Te queda claro lo qué debe hacer cada método? Si no, pregunta a tu ayudante.

Actividad 9.2

Elige los métodos que concideres más importantes y dibuja cómo te imaginas que se ve la lista antes de mandar llamar un método y qué le sucede cuando éste es invocado.

Una **lista doblemente ligada** es una implementación de la estructura de datos lista, que se caracteriza por:

1. Guardar los datos de la lista dentro de nodos que hacen referencia al nodo con el dato anterior y al nodo con el siguiente dato.
2. Tener una referencia al primer y último nodo.
3. Tener un tamaño dinámico, pues el número de datos que se puede almacenar está limitado únicamente por la memoria de la computadora y el tamaño de la lista se incrementa y decrementa conforme se insertan o eliminan datos de ella.
4. Es fácil recorrerla de inicio a fin o de fin a inicio.

Desarrollo

Para implementar el TDA *Lista* se deberá extender la clase:

`ColeccionAbstracta<E>`

programada anteriormente e implementar la interfaz `List<E>`. Esto se deberá hacer en una clase llamada

`ListaDoblementeLigada<E>`.

1. Dentro del paquete correspondiente, programa `ListaDoblementeLigada<E>` según lo indicado.
2. Programa los métodos faltantes. Sólamente `sublist()` es opcional, los demás son obligatorios.

Preguntas

1. Explica la diferencia conceptual entre los tipos `Nodo<E>` y `E`.
2. ¿Por qué `ListIterator` sólo permite remover, agregar o cambiar datos después de llamar `previous` o `next`?
3. Si mantenemos los elementos ordenados alfabéticamente, por ejemplo, ¿cuándo sería más eficiente agregar un elemento desde el inicio o el final de la lista?
4. En qué casos sería más eficiente obtener un elemento desde el inicio de la lista o desde el final de la lista.

PARTE II
APLICACIONES

10 | Aplicación de Pilas: Backtracking

BACKTRACKING

Una aplicación de las pilas es el algoritmo conocido como *backtracking*. Se utiliza para buscar soluciones a problemas en forma sistemática. En esta sección se utilzará una pila para resolver el problema de las *n-reinas* utilizando backtracking (Main 2003).

Problema de las n-reinas

Dado un tablero de ajedrez de $n \times n$ casillas, se desea colocar n reinas sin que se coman las unas a las otras. [1] En la Figura 10.1 se muestra la solución para un tablero 5×5.

Actividad 10.1

Busca a mano una solución para el problema de 8×8. Sí hay solución.

Ejercicio

Harás un programa que, dado el número n de reinas, determine si existe una solución para el problema en un tablero $n \times n$. En caso de existir imprimirá la columna y renglón donde se debe colocar cada reina, de lo contrario imprimirá un aviso indicando que no existe solución para ese tamaño del tablero.

El algoritmo funciona de la siguiente manera: para que las reinas no se coman, deben estar en renglones distintos. Por lo tanto, ya sabes que debes colocar una reina por renglón y falta averiguar en qué columna. Irás probando a colocar las reinas una por una de izquierda a derecha incrementando renglón por renglón. A continuación se incluye el pseudocódigo correspondiente. Tu labor es implementarlo en `Java` utilizando

[1] Recuerda que una reina en el ajedrez se puede comer a las piezas que se encuentran en la misma columna, mismo renglón o sobre cualquiera de las dos diagonales.

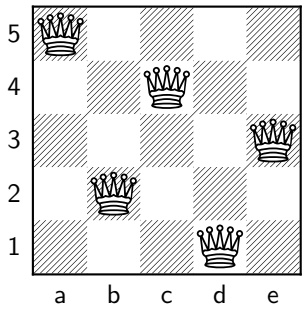


Figura 10.1 Solución al problema para 5-reinas.

la pila que acabas de programar, con su iterador. Nota que los renglones se cuentan a partir de 1 y las columnas se indican con letras, puedes representar internamente a las columnas con números, si así lo prefieres, pero el programa debe imprimir los resultados usando la notación con caracteres.

Algoritmo 1 Backtracking N-Reinas.

```
1:  function ResuelveNReinas(n)
2:      pila ←′ a′                                  ▷ Iniciamos con (1, a)
3:      while pila ≠ ∅ do
4:          if La última reina agregada es comida por alguna de las anteriores then
5:              while pila ≠ ∅ y mira(pila) = n do   ▷ Se acabaron las opciones
6:                  expulsa(pila)                    ▷ Regresa un renglón
7:              if pila ≠ ∅ then
8:                  mira(pila) ← mira(pila) + 1      ▷ Prueba la siguiente columna
9:          else if tamaño(pila) = n then
10:             return pila                          ▷ Hay n reinas en su lugar
11:         else
12:             pila ←′ a′                           ▷ Avanza un renglón
13:     if pila = ∅ then return Fallo
14:     elsereturn pila
```

1. Agrega una clase en el paquete `ed.aplicaciones`. En esta clase implementarás el algoritmo y deberás incluir un método `main` para ejecutarlo. Eres libre para diseñar tu clase, pero procura seguir las buenas prácticas de orientación a objetos.

2. Modifica el archivo `build.xml` para que al escribir `ant run` se ejecute tu programa.

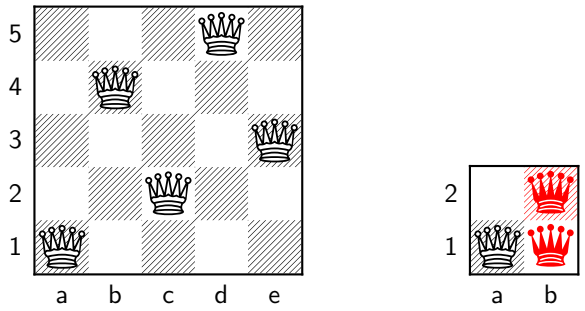


Figura 10.2 Otra solución al problema para 5-reinas, obtenida con *backtracking*. En el tablero 2×2 no hay solución.

3. Prueba tu código para varios valores de n y verifica que las soluciones encontradas sean válidas, añade algunos ejemplos a tu reporte.

Este es un ejemplo de solución, que corresponde a la Figura 10.2:

Listing 10.1: Solución 5x5

```
Tablero 5x5
Renglón 5, columna e
Renglón 4, columna c
Renglón 3, columna f
Renglón 2, columna d
Renglón 1, columna b

Tablero 2x2
No hay solución
```

Un tablero 2×2 es un ejemplo de n para la cual no existe solución.

11 | Aplicación de pilas y colas: Calculadora

Calculadora

Una calculadora científica o un intérprete matemático de comandos debe ser capaz de evaluar expresiones utilizando precedencia de operadores. Es decir, las multiplicaciones y divisiones se evalúan antes que las sumas y restas. Para resaltar la diferencia veamos el ejemplo siguiente.

Ejemplo 11.1 *Sin precedencia de operadores, la expresión matemática siguiente se evalúa:*

$$2 + 5 \times -2 = 10 \times -2 = -20 \tag{11.1}$$

con precedencia esto es:

$$2 + 5 \times -2 = 2 - 10 = -8 \tag{11.2}$$

Para obtener un resultado equivalente a la Ecuación 11.1, pero con precedencia, debemos utilizar paréntesis:

$$(2 + 5) \times -2 = 10 \times -2 = -20 \tag{11.3}$$

Esta es una característica de la notación que utilizamos para las operaciones binarias, donde el operador se escribe entre los dos operandos. A esta notación se le llama *infija*.

Notaciones Prefija y Postfija

Existen notaciones alternativas que no requiren de la especificación del orden de precedencia. Se trata de las notaciones *prefija o polaca* y *posfija o polaca inversa*. En estas notaciones el operador se coloca a la izquierda o derecha de los operandos, respectivamente.

Ejemplo 11.2 *La Ecuación 11.1 en notación prefija se ve:*

$$\times \quad + \quad 2 \quad 5 \quad -2 \tag{11.4}$$

Mientras que la Ecuación 11.2 se convierte en:

$$+ \quad 2 \quad \times \quad 5 \quad -2 \tag{11.5}$$

Evaluación

Evaluar una expresión prefija o postfija es más sencillo que evaluar una expresión infija. Sólo se require de una pila como estructura auxiliar. El algoritmo se muestra en pseudocódigo en Apéndice 2.

Algoritmo 2 Evalua expresión prefija.

```
1:  function EvaluaPrefija(expresión)
2:      pila ← ∅
3:      for token ∈ expresióndederechaaizquierda do
4:          if token es un número then
5:              pila ← token
6:          else if token es un operador then
7:              operador_1 ← pila.expulsa()
8:              operador_2 ← pila.expulsa()
9:              pila.empuja(evalua(operador_1 tokenoperador_2))
10:     end for
11:     return pila.expulsa()
```

El algoritmo para evaluar notación posfija es semejante, pero expresión se revisa de izquierda a derecha y se debe tener cuidado de extraer los dos operandos en el orden adecuado.

Conversión

Para evaluar una expresión infija, podemos convertirla a notación prefija o postfija. Para realizar la conversión se usan una pila y una cola. El algoritmo se muestra en pseudocódigo en Apéndice 3.

Algoritmo 3 Convierte expresión infija a postfija.

```
1:  function ConvierteAPosfija(infija)
2:      pila ← ∅
3:      cola ← ∅
4:      for token ∈ infija do
5:          if token es un número then
6:              cola.forma(token)
7:          else if token es un operador then
8:              while precedencia(pila.peek()) >= precedencia(token) do
9:                  cola.forma(pila.expulsa())
10:             pila.empuja(token)
11:     end for
12:     while pila ≠ ∅ do
13:         cola.forma(pila.expulsa())
14:     return cola
```

Ejercicio

Harás un programa que evalúe expresiones introducidas por el usuario en las tres notaciones.

1. Revisa el código y la documentación de la clase `Fija`. Implementa los métodos. Observa que ya tiene un método `main` donde puedes probar tu código.
2. Revisa el código y la documentación de la clase `Infija`. Implementa los métodos. Aquí está el método `main` que te permite elegir entre cualquiera de las notaciones.

Este es un ejemplo de interacción con el usuario (no es necesario imprimir los tokens, pero aquí se utilizó para verificar lo que está haciendo el programa):

Listing 11.1: Infija

```
Calculadora en modo notación infija
4 + 5 - ( 2 * 5 )
Tokens: [4, +, 5, -, (, 2, *, 5, , )]
Sufija: [4, 5, +, 2, 5, *, -]
= -1.0
prefija
Cambiando a notación prefija
+ -4 - 8 2
[+, -4, -, 8, 2]
= 2.0
postfija
```

```
Cambiando a notación postfija
3 5 -7 * 3 -
[3, 5, -7, *, 3, -]
= -38.0
exit
```

12 | Aplicación de listas: Directorio

Directorio

Una aplicación inmediata de una lista es una lista de contactos o directorio telefónico. Sus principales usos son:

- Alamcenar *nombre*, *dirección*, *teléfono* y *correo electrónico* de tus contactos.
- Permitirte consultar los datos de algún contacto, dando su nombre o parte de él.
- Listar todo su contenido.
- Agregar contactos.
- Remover contactos.

Ejercicio

1. Crea una clase `Registro` con atributos tipo cadena para almacenar *nombre*, *dirección*, *teléfono* y *correo electrónico* de un contacto.
2. Agrega un método `boolean contains(String nom)` que devuelva `true` si el nombre contiene las subcadena `nom`.
3. Sobreescribe el método `toString()` para que el registro devuelva una cadena con sus datos, tal y como te gustaría que aparecieran en la consola.
4. Crea una clase `Directorio`, que tenga una lista de registros como atributo.
5. Agrega un método que liste (imprima en la consola) todos los registros cuyo nombre contenga una subcadena dada, junto con el índice de la posición que ocupan en la lista.

6. Agrega un método para guardar el contenido del directorio en un archivo de texto.
7. Agrega un método para cargar el directorio desde un archivo de texto.
8. Crea una interfaz de texto para el usuario con opciones para:
   a) Cargar un directorio desde un archivo.
   b) Guardar el directorio en un archivo.
   c) Listar el contenido del directorio.
   d) Agregar un registro nuevo.
   e) Buscar con subcadena.
   f) Borrar o modificar un registro dado su índice.

Bibliografía

Aho, Alfred V., John E. Hopcroft y Jeffrey D. Ullman (1983). *Data Structures and Algorithms*. Addison-Wesley.

Main, Michael (2003). *Data Structures & Other Objects Using Java*. 2nd. Pearson Education, Inc. 808 págs.