Algoritmia II

Guías de autoevaluación

Roberto Flórez Rueda

Rector de la Universidad de Antioquia
Alberto Uribe Correa

Vicerrector de Docencia
Óscar Sierra Rodríguez

Decano de la Facultad de Ingeniería
Elkin Libardo Ríos Ortiz

Vicedecano de la Facultad de Ingeniería
Carlos Alberto Palacio Tobón

Coordinador del Programa de Educación Ude@
Guillermo León Ospina Gómez

Autor
Roberto Flórez Rueda

Jefe del Departamento de Recursos de Apoyo e Informática (DRAI)
Juan Diego Vélez Serna

Coordinadora de Producción
Lyda Yaneth Contreras Olivares

Corrector de estilo / Asesor pedagógico
Daniel Aldana Estrada / Carlos Alberto Hurtado García

Diagramación y diseño
Juan Felipe Vargas Martínez

Impresión
Cátedra Litografía

Primera edición, julio de 2010

ISBN: 978-958-714-441-3

Impreso en Medellín (Colombia).

Presentación

La guía de autoevaluación configura un proceso que incluye verificación, diagnóstico, reflexión, corrección y realimentación como fases fundamentales que permiten identificar las fortalezas y debilidades de los estudiantes en cuanto al conocimiento adquirido, buscando mejorar sus niveles de desempeño para garantizar certeza y calidad en la aplicación de dichos conocimientos en situaciones problemáticas que emerjan en su entorno.

La guía de autoevaluación tiene como propósito llevar al estudiante a que practique, ponga a prueba y verifique por sí mismo el grado de apropiación que ha logrado sobre una temática específica, todo ello apoyado en una completa y clara realimentación que los docentes hacen de cada uno de los puntos que incluyen las actividades propuestas.

El diseño educativo de los cursos Ude@ incluye guías de autoevaluación que, a partir de la realimentación que hace el docente, ofrecen a los estudiantes indicadores sobre su avance en el aprendizaje a la vez que permiten apoyar el estudio independiente y la toma de decisiones sobre qué puntos deberá mejorar para lograr los objetivos.

Una guía de autoevaluación en Ude@ puede estar integrada por diferentes tipos de actividades: ejercicios para resolver a partir de desarrollo matemático, cuestionarios de relación de columnas, actividades de falso y verdadero, actividades de complementación, análisis de casos, historietas, tiras cómicas, entre otros, e incluye como proceso de realimentación no sólo las respuestas correctas a los ejercicios y cuestionarios planteados, sino las sustentaciones de por qué las demás opciones no eran correctas.

En síntesis, la guía de autoevaluación se plantea con el fin de que el estudiante tenga la oportunidad de verificar los conocimientos adquiridos durante cada semana; las actividades propuestas están relacionadas en su totalidad con la temática abordada en este lapso de tiempo y le ayudarán a recordar o a afianzar aquellos conceptos y/o procedimientos que le permitan acercarse con mayor propiedad a una prueba real de conocimientos.

La idea es que el estudiante desarrolle la autoevaluación las veces que desee hasta que logre obtener el nivel de acierto deseado. No obstante, y aunque la realimentación de cada actividad muestra las respuestas correctas para cada caso, se le sugiere al aprendiz abstenerse de revisarlas mientras no haya intentado —como mínimo una vez— resolver la actividad a partir de su propio conocimiento, ya sea eligiendo una respuesta correcta entre las opciones presentadas o desarrollando los ejercicios a lápiz para después verificar el respectivo procedimiento y resultado.

Autoevaluación 1

La presente autoevaluación está desarrollada con base en los ejercicios 1, 3 y 5 propuestos en el módulo 1 del texto guía.

1. Determine el contador de frecuencias y el orden de magnitud para el algoritmo 1 propuesto en el módulo 1 del texto guía.

```
1. void  p1(entero  n)
2.           s = 0
3.           for  (i = 1; i <= n; i++)  do
4.                for  (J = 1; J <= i; J++)  do
5.                     for  (k = 1; k <= J; k++)  do
6.                          s = s + 1
7.                     end(for)
8.                end(for)
9.           end(for)
10.  fin(p1)
```

Solución:

```
1. void  p1(entero  n) _____________________  1
2.           s = 0 _____________________________  1
3.           for  (i = 1; i <= n; i++)  do _____________  n + 1
4.                for  (J = 1; J <= i; J++)  do _______  n(n + 1)/2 + n
5.                     for  (k = 1; k <= J; k++)  do _  n(n + 1)(n + 2)/6 + n(n + 1)/2
6.                          s = s + 1 ________  n(n + 1)(n + 2)/6
7.                     end(for) ___________________  n(n + 1)(n + 2)/6
8.                end(for) ____________________  n(n + 1)/2
9.           end(for) ____________________________  n
10.  fin(p1) ____________________________________  1
```

$$\text{Contador de frecuencias} = \frac{}{n(n+1)(n+2)/2 + 3n(n+1)/2 + 3n + 4}$$

Explicación:

Las instrucciones 1, 2 y 10 se ejecutan, cada una de ellas, una sola vez. Esto es obvio ya que no se hallan dentro de ningún ciclo. La instrucción 3 plantea un ciclo con la i desde 1 hasta n, con incrementos de a 1. Esto lleva a que la instrucción 3 se ejecute una vez más, ya que es la instrucción donde se evalúa la condición de terminación del ciclo. Por consiguiente, la instrucción 3 se ejecutará **n + 1** veces y la instrucción 10 **n** veces. La

instrucción 4 plantea un ciclo donde la variable controladora del ciclo es la **j**. La **j** varía desde 1 hasta **i**, es decir, el número de veces que se ejecuten las instrucciones de este ciclo interno depende del valor de la **i**: cuando la **i** vale 1 las instrucciones de este ciclo interno se ejecutan una vez; cuando la **i** vale 2 las instrucciones se ejecutan dos veces; cuando la **i** vale 3 las instrucciones se ejecutan tres veces, y hasta este punto, el total de veces que se han ejecutado las instrucciones de este ciclo interno es 6 (1 + 2 + 3); cuando la **i** valga **n** el total de veces que se habrán ejecutado las instrucciones de este ciclo interno es 1 + 2 + 3 + 4 +…+ n, sumatoria que según vimos en el texto guía (numeral 1.5) es n*(n + 1)/2. Por lo tanto, la instrucción 8 se ejecuta n*(n + 1)/2 veces, mientras que la instrucción 4 se ejecutará este mismo número de veces más **n** ya que por cada vez que ejecute el ciclo externo (que se ejecuta **n** veces) la instrucción 4 se ejecuta una vez más. Para determinar cuántas veces se ejecuta el ciclo más interno (instrucciones 5 a 7) consideremos la siguiente tabla:

i	j	k	Total de veces que se ejecuta k para un valor de i	Total de veces acumulado de ejecución de k
1	1	1	1	1
2	1	1		
	2	1, 2		
			3	4
3	1	1		
	2	1, 2		
	3	1, 2, 3		
			6	10
4	1	1		
	2	1, 2		
	3	1, 2, 3		
	4	1, 2, 3, 4		
			10	20
. . .				

Cuando la **i** vale 1, el ciclo de la **j** varía desde 1 hasta 1; por lo tanto, el ciclo de la **k** se ejecuta sólo una vez (columna "total de veces que se ejecuta **k** para un valor de **i**").

Cuando la **i** vale 2, el ciclo de la **j** varía desde 1 hasta 2; por lo tanto, el ciclo de la **k** se ejecuta una vez cuando la **j** vale 1, y dos veces cuando la **j** vale 2, es decir, que en total se ejecuta tres veces (columna "total de veces que se ejecuta **k** para un valor de **i**").

Cuando la **i** vale 3, el ciclo de la **J** varía desde 1 hasta 3; por lo tanto, el ciclo de la **k** se ejecuta una vez cuando la **J** vale 1, dos veces cuando la **j** vale 2, y tres veces cuando la **j** vale 3. En consecuencia, el ciclo de la **k** se ejecuta seis veces (columna "total de veces que se ejecuta **k** para un valor de **i**").

Cuando la **i** vale 4, el ciclo de la **j** varía desde 1 hasta 4; por lo tanto, el ciclo de la **k** se ejecuta 1 vez cuando la **j** vale 1, dos veces cuando la **J** vale 2, tres veces cuando la **j** vale 3, y cuatro veces cuando la **j** vale 4. En consecuencia, el ciclo de la **k** se ejecuta seis veces (columna "total de veces que se ejecuta **k** para un valor de **i**").

Ahora, si queremos determinar cuántas veces se ha ejecutado en total el ciclo de la **k**, miremos la "columna total de veces acumulado de ejecución de **k**": es la sumatoria de 1 + 3 + 6 + 10 + …, la cual, expresada matemáticamente, es:

$$S = \sum_{i=1}^{i=n} (i * (i + 1)/2).$$

que, al resolverla, da la expresión mostrada en las líneas 6 y 7 del algoritmo.

En la línea 5 se suma n*(n + 1)/2 puesto que este es el número de veces que se ejecuta el ciclo de la **j**.

Procesando el contador de frecuencias obtenido tenemos que el orden de magnitud de este algoritmo es cúbico: $O(n^3)$.

3. Determine el contador de frecuencias y el orden de magnitud para el algoritmo 3 propuesto en el módulo 1 del texto guía.

```
1.   void  p3(vector V, entero n, real x)
2.         i = 1
3.         j = n
4.         do
5.                   k = (i + j)/2
6.                   if V[k] <= x then
7.                             i = k + 1
8.                   else
9.                             j = k – 1
10.                            end(if)
11.                  while (i <= j)
12.   fin(void)
```

Solución:

```
1. void  p3(vector V, entero n, real x) _____ 1
2.           i = 1 _____ 1
3.           j = n _____ 1
4.           do _____ log2n
5.                   k = (i + j)/2 _____ log2n
6.                   if V[k] <= x then _____ log2n
7.                           i = k + 1 _____ log2n
8.                   else _____ log2n
9.                           j = k – 1 _____ log2n
10.                  end(if) _____ log2n
11.          while (i <= j) _____ log2n + 1
12. fin(void) _____ 1
```

Contador de frecuencias = $8 \log_2 n + 5$

Explicación:

Es obvio que cada una de las instrucciones 1, 2, 3 y 12 se ejecuta sólo una vez. Analicemos el ciclo (instrucciones 4 a 11): este ciclo se ejecuta hasta que la **i** sea mayor que la **j** (instrucción 11). El valor inicial de la **i** es 1, y el valor inicial de la **j** es **n**. En cada iteración se calcula la mitad entre **i** y **j** (instrucción 5), y dependiendo del resultado de la comparación de la instrucción 6 el valor de **i** será la mitad más 1 o el valor de la **j** será la mitad menos 1, es decir, como resultado de cada comparación se elimina la mitad de los datos sobre los cuales actúa el ciclo. Dicho de otra forma, en cada iteración se divide la muestra de datos entre 2, lo cual arroja como resultado que el número de veces que se ejecutan las instrucciones del ciclo es logarítmico en base 2 (algoritmos 5 y 6 del texto guía). Por consiguiente, el número de veces que se ejecutan las instrucciones 4 a 10 es $\log_2 n$ y la instrucción 11 se ejecuta una vez más, ya que es la instrucción que evalúa la condición. El orden de magnitud de este algoritmo es logarítmico en base 2: $O(\log_2 n)$.

5. Determine el contador de frecuencias y el orden de magnitud para el algoritmo 5 propuesto en el módulo 1 del texto guía.

```
1.   void  p5()
2.           read(n)
3.           s = 0
4.           i = 2
5.           while  (i <= n)  do
6.                   s = s + 1
7.                   i = i * i
8.           end(while)
9.           write(n, s)
10.   fin(p5)
```

Solución:

```
1.  void  p5() ______________________  1
2.          read(n) _________________  1
3.          s = 0 ___________________  1
4.          i = 2 ___________________  1
5.          while  (i <= n)  do ______  log2(log2n) + 1
6.                  s = s + 1 _______  log2(log2n)
7.                  i = i * i _______  log2(log2n)
8.          end(while) ______________  log2(log2n)
9.          write(n, s) _____________  1
10.  fin(p5) ________________________  1
                                       _________________
        Contador de frecuencias  =  4 log2(log2n) + 7
```

Explicación:

Es obvio que cada una de las instrucciones 1, 2, 3, 4, 9 y 10 se ejecuta sólo una vez. Analicemos el ciclo (instrucciones 5 a 8): cada vez que entra al ciclo la variable **i** se eleva al cuadrado (instrucción 7). En nuestro ejemplo, el valor inicial de la **i** es 2, al ejecutar el ciclo la primera vez el valor de la **i** queda en cuatro, al ejecutar el ciclo la segunda vez el valor de la **i** será 16, al ejecutar el ciclo la tercera vez el valor de la **i** será 216, y así sucesivamente.

Al expresarlo de una forma matemática, el valor de la **i** será:

$$\left(2^{2^k}\right)$$

siendo **k** el número de veces que se ejecuta el ciclo. Para que se ejecute el ciclo, el resultado de esta operación deberá ser menor o igual que **n**. Por consiguiente, deberemos despejar **k**. Para despejarla, tenemos:

$$
\begin{aligned}
\left(2^{2^k} <= n\right) \\
\log_2\left(2^{2^k}\right) &<= \log_2 n \\
2^k \log_2(2) &<= \log_2 n \\
2^k &<= \log_2 n \\
\log_2(2^k) &<= \log_2(\log_2 n) \\
k \log_2(2) &<= \log_2(\log_2 n) \\
k &<= \log_2(\log_2 n)
\end{aligned}
$$

Por consiguiente, el número de veces que se ejecuta el ciclo es $\log_2(\log_2 n)$, tal como se muestra en las instrucciones 5, 6, 7 y 8 del algoritmo. La instrucción 5 se ejecuta una vez más puesto que la pregunta también se ejecuta cuando evalúa la condición con la que se sale del ciclo.

Ahora, tengamos en cuenta lo siguiente: si en la instrucción 4 la variable **i** se hubiera inicializado en 3, la forma como se incrementa la variable controladora del ciclo, es decir la **i**, viene dada por la fórmula:

$$(3^{2^k})$$

Por consiguiente, para despejar **k** tendremos:

$$\begin{aligned}
(3^{2^k} &<= n) \\
\log_3(3^{2^k}) &<= \log_3 n \\
2^k \log_3(2) &<= \log_3 n \\
2^k &<= \log_3 n \\
\log_2(2^k) &<= \log_2(\log_3 n) \\
k \log_2(2) &<= \log_2(\log_3 n) \\
k &<= \log_2(\log_3 n)
\end{aligned}$$

lo cual significa que el número de veces que se ejecutan las instrucciones del ciclo es $\log_2(\log_3 n)$.

En general, para un algoritmo que tenga la forma del que estamos analizando, si el valor inicial de la **i** es **x**, el número de veces que se ejecutan las instrucciones de dicho ciclo es $\log_2(\log_x n)$.

Autoevaluación 2

La presente autoevaluación está desarrollada con base en los ejercicios 1, 3 y 5 propuestos en el módulo 2 del texto guía.

1. Elabore un algoritmo que modifique el ordenamiento por inserción efectuando el proceso de búsqueda, utilizando la búsqueda binaria.

   Solución:

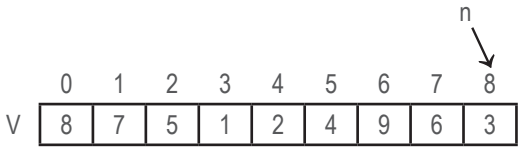
   El procedimiento de inserción modificado puede ser implementado como sigue:

```
1.  void  insercionModificado()
2.            entero  i, j, k, d
3.            for  (i = 2; i <= n; i++)  do
4.                        d = V[i]
5.                        k = posicion(1, i – 1, d)
6.                        for  (j = i – 1; j >= k; j--)  do
7.                                    V[j+1] = V[j]
8.                        end(for)
9.                        V[k] = d
10.           end(for)
11.  fin(insercionModificado)
```

   Nos apoyamos en un método que calcula la posición en donde hemos de insertar un nuevo elemento dentro de un subvector previamente ordenado, utilizando búsqueda binaria. El subvector en el cual se efectúa la búsqueda es desde la posición 1 hasta la posición i – 1. Dicho método lo presentamos a continuación:

```
1.  entero  posicion(entero  primero, entero ultimo, entero  d)
2.            entero  mitad
3.            while  (primero <= ultimo)  do
4.                        mitad = (primero + ultimo)/2
5.                        if  (d == V[mitad]  then  return  mitad
6.                        if  (d < V[mitad]  then
7.                                    ultimo = mitad – 1
8.                        else
9.                                    primero = mitad + 1
10.                       end(if)
11.           end(while)
12.           return  mitad
13.  fin(posicion)
```

3. Haga seguimiento detallado, paso por paso, al proceso de ordenamiento del vector mostrado a continuación usando el método de burbuja:

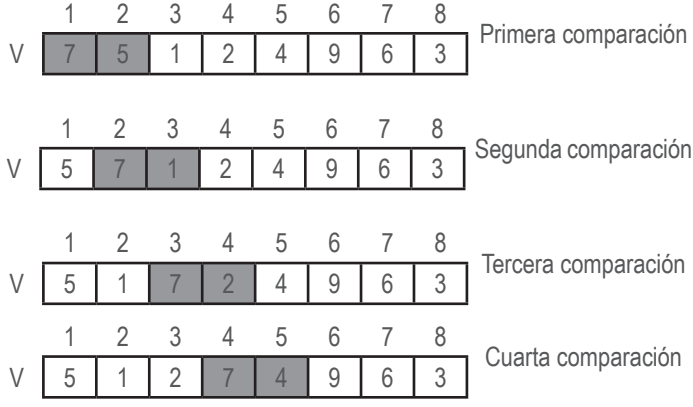


Solución:

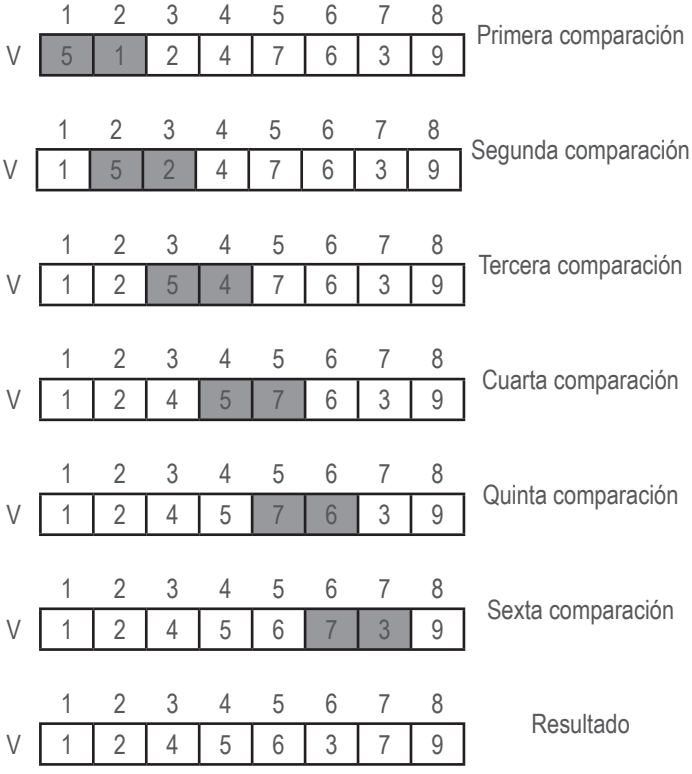
Primero recordemos nuestro algoritmo de ordenamiento ascendente por el método burbuja:

```
1.  void burbuja()
2.          entero  i, j
3.          for  (i = 1; i < n; i++)  do
4.                  for  (j =1; j <= n – i; j++)  do
5.                          if  (V[j] > V[j + 1])  then
6.                                  intercambia(j, j + 1)
7.                          end(if)
8.                  end(for)
9.          end(for)
10. end(burbuja)
```

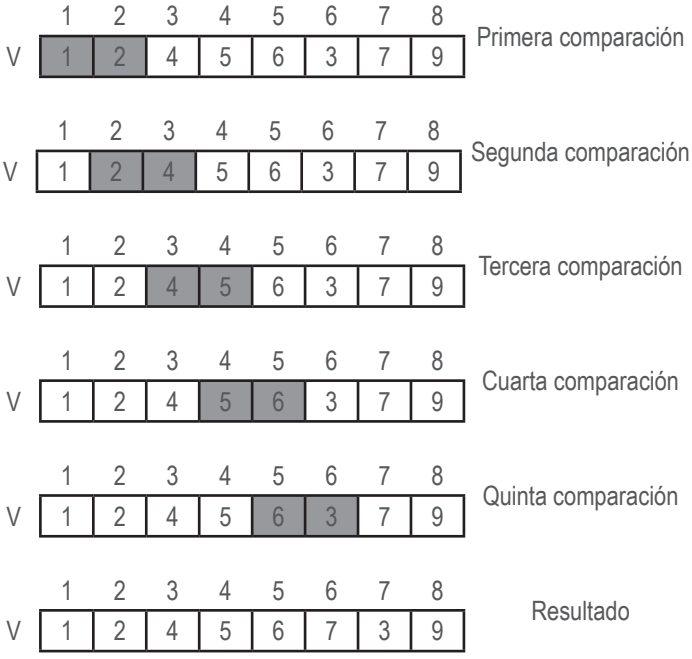
Tenga en cuenta que el método consiste en efectuar **n – 1** pasadas, las cuales se cuentan con el ciclo definido en la instrucción 3. En cada pasada se comparan elementos consecutivos: si V[j] > V[j + 1] se intercambian dichos datos y se continúa con la siguiente pareja; de lo contrario, simplemente se continúa con la siguiente pareja. El resultado de una pasada es que el mayor dato queda ubicado en la última posición del vector. Para nuestro ejemplo las comparaciones y las acciones que se efectúan en la primera pasada son:



En la segunda pasada se hacen sólo seis comparaciones ya que el dato de la posición 8 está en su sitio. Las comparaciones paso a paso de la segunda pasada son:

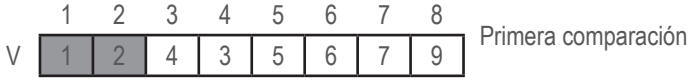


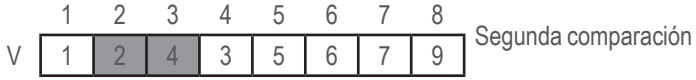
Como podrá observar, en las posiciones 7 y 8 ya están los dos últimos datos; por lo tanto, en la siguiente pasada estos datos no entran en las comparaciones. La tercera pasada es:

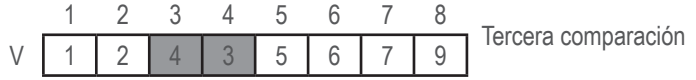


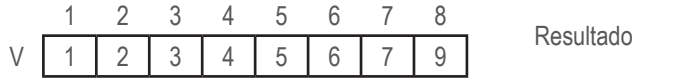
En esta pasada ya están ordenados los datos de las tres últimas posiciones. La cuarta pasada es:

Después de esta pasada ya han quedado ordenados ascendentemente los datos desde la posición 5 hasta la 8. La siguiente pasada es:

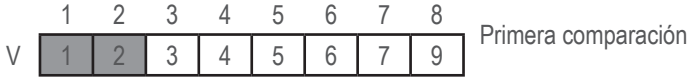


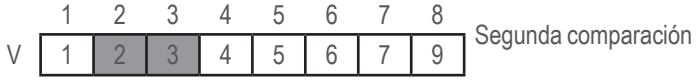


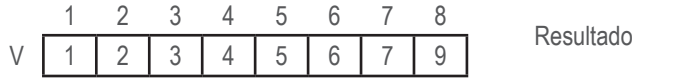




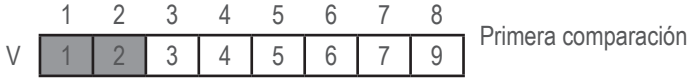
En este punto los datos del vector ya han quedado ordenados; sin embargo, este algoritmo no detecta esta situación, y por lo tanto continúa haciendo pasadas. La siguiente pasada es:

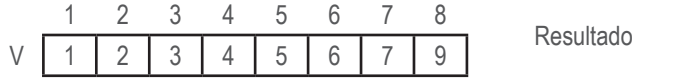






Si el método utilizado fuera el de burbuja mejorado, en este punto termina el proceso porque detectaría que no hubo que hacer ningún intercambio en las comparaciones que se hicieron. Pero como estamos haciéndolo con el método burbuja simple, el algoritmo continúa con las comparaciones, así éstas sean inoficiosas. La siguiente pasada es:

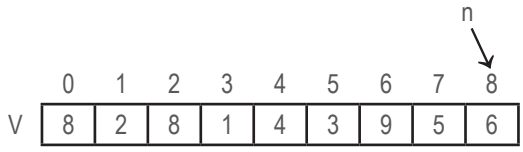




Efectuó la última pasada y termina el proceso.

Fíjese que no hemos tenido en cuenta el dato de la posición 0 en el proceso de ordenamiento, ya que dicha posición la utilizamos para guardar el número de datos en el vector.

5. Haga seguimiento detallado, paso por paso, al proceso de ordenamiento del vector mostrado a continuación usando el método de inserción.

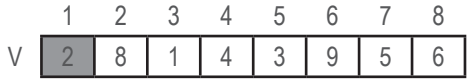


Solución:

Recordemos primero nuestro algoritmo de ordenamiento ascendente por el método inserción:

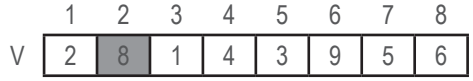
```
1.  void  inserción()
2.            entero  i, j, d
3.            for  (i = 2; i <= n; i++) do
4.                        d = V[i]
5.                        j = i – 1
6.                        while ((j > 0) and (d < V[j])) do
7.                                    V[j + 1] = V[j]
8.                                    j = j – 1
9.                        end(while)
10.                       V[j + 1] = d
11.           end(for)
12.  end(inserción)
```

Recuerde que el método de inserción comienza en la posición 2 del vector. Compara el dato de la posición 2 con el de la posición 1. Si el de la posición 2 es menor que el dato de la posición 1, intercambia dichos datos. Luego continúa el proceso con el dato de la posición 3, comparando dicho dato con todos los datos desde la posición 2 hasta llegar a la posición 1 (llamamos **j** la variable con la cual efectuamos ese recorrido) o hasta encontrar un dato que sea menor que el de la posición 3. A medida que se van haciendo las comparaciones vamos desplazando el dato de la posición **j** una posición hacia la derecha si se cumple que el dato de la posición **j** es mayor que el de la posición con el cual se está trabajando (el dato de la posición **i**). Este proceso se desarrolla con el ciclo de las instrucciones 6 a 9. Al terminar este ciclo significa que se ha hallado la posición en la cual debe quedar el dato que se hallaba en la posición **i**: la posición **j + 1**. En la instrucción 10 colocamos el dato que estaba en la posición **i** en la posición **j + 1**. Gráficamente el proceso es: cuando **i** vale 2, entonces d = V[2] = 8.

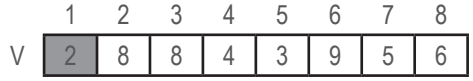


Se compara el dato **d** con el dato de la posición 1 (la variable **j** comienza en 1 puesto que **i** es 2). Como el dato de la posición **j** (j = 1) es menor que **d** se termina el ciclo de las instrucciones 6 a 9 y se procede a almacenar la **d** en la posición **j + 1**, es decir, en la misma posición 2.

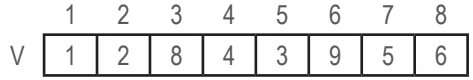
Cuando la **i** valga 3 tenemos d = V[3] = 1. El proceso de comparaciones es:



Compara 1 con el dato de la posición 2

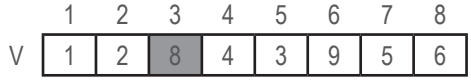


Compara 1 con el dato de la posición 1

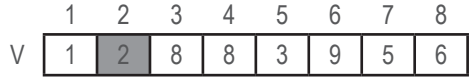


Almacena el 1 en la posición 1

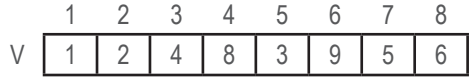
Cuando la **i** valga 4 tenemos d = V[4] = 4. El proceso de comparaciones es:



Compara 4 con el dato de la posición 3



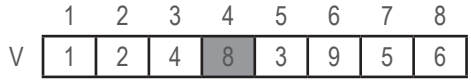
Compara 4 con el dato de la posición 2



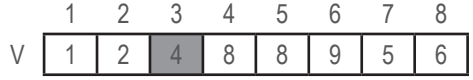
Almacena el 4 en la posición 3

Fíjese que los primeros cuatro datos del vector ya están ordenados en forma ascendente.

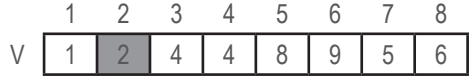
Cuando la **i** valga 5 tenemos d = V[5] = 3. El proceso de comparaciones es:



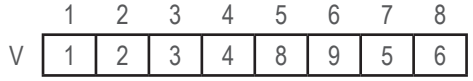
Compara 3 con el dato de la posición 4



Compara 3 con el dato de la posición 3



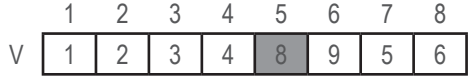
Almacena el 3 con el dato de la posición 2



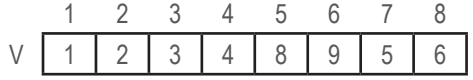
Almacena el 3 en la posición 3

En este paso ya tenemos ordenados los datos desde la posición 1 hasta la 5.

Consideremos ahora cuando la **i** vale 6: d = V[6] = 9. El proceso de comparaciones es:

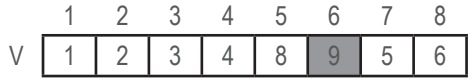


Almacena el 9 con el dato de la posición 5

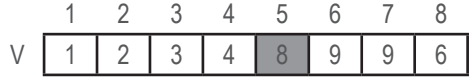


Almacena el 9 en la posición 6

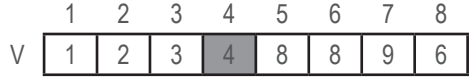
Cuando la **i** valga 7 tenemos d = V[7] = 5 y el proceso de comparaciones es:



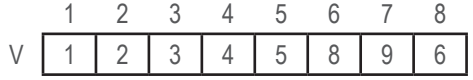
Compara 5 con el dato de la posición 6



Compara 5 con el dato de la posición 5

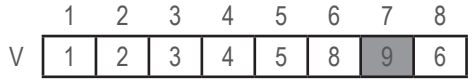


Almacena el 5 con el dato de la posición 4

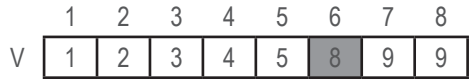


Almacena el 5 en la posición 5

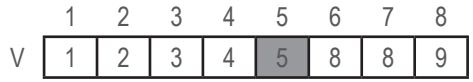
Sólo falta por ordenar el dato de la posición 8. Cuando **i** sea 8: d = V[8] = 6 y el proceso de comparaciones es:



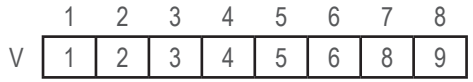
Compara 6 con el dato de la posición 7



Compara 6 con el dato de la posición 6



Almacena el 6 con el dato de la posición 5



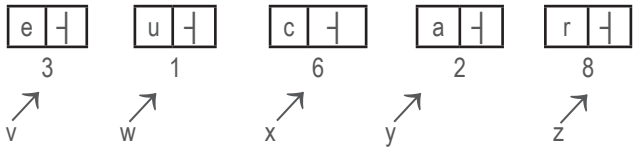
Almacena el 6 en la posición 6

Con esto hemos ordenado ascendentemente los datos del vector.

Autoevaluación 3

La presente autoevaluación está desarrollada con base en el ejercicio 1 propuesto en el módulo 3 del texto guía.

1. Dados los siguientes nodos:



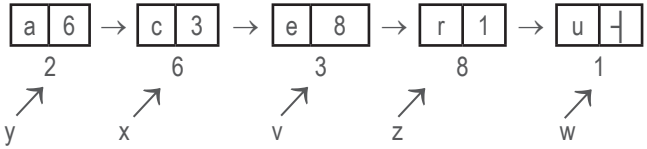
a. Escriba las instrucciones para conectarlos, de tal manera que al recorrerlos queden escritos ascendentemente.
b. Redibuje la lista de tal forma que los datos se vean ordenados ascendentemente.
c. Con la lista obtenida después de ejecutar el literal b, se requiere insertar un nuevo nodo con la letra "f", de tal manera que al recorrer la lista los datos sigan estando ordenados ascendentemente. Escriba las instrucciones necesarias para hacer esta tarea.
d. Con la lista obtenida en el literal c asigne null a todas las variables, excepto a la **y**. Ahora, escriba las instrucciones necesarias para llegar al nodo que tiene el dato "u" e imprimir el número del nodo en el cual la encontró.
e. Con la lista obtenida en el literal c y teniendo la lista con todas las variables en null, excepto la **y**, escriba las instrucciones necesarias para desconectar de la lista el nodo que tiene el dato "r".

Solución:

a. El primer nodo debe ser el que contiene el dato "a", es decir, el nodo 2, el cual se identifica con la variable **y**. Su campo de liga debe quedar apuntando hacia el nodo 6, el cual contiene el dato "c" y se identifica con la variable **x**; por lo tanto, para conectar **y** con **x** se debe modificar el campo de liga del nodo **y**: y.asignaLiga(x). A continuación de la "c" sigue la "e", la cual se halla almacenada en el nodo 3, que se identifica con la variable **v**; por consiguiente, para conectar **x** con **v** se debe modificar el campo de liga de nodo **x**: x.asignaLiga(v). Luego de la "e" sigue la "r", la cual se halla almacenada en el nodo 8, que identifica con la variable **z**; por lo tanto, para conectar **v** con **z** se debe modificar el campo de liga de nodo **v**: v.asignaLiga(z). Por último está la "u", la cual se halla almacenada en el nodo 1, que identifica con la variable **w**; en consecuencia, para conectar **z** con **w** se debe modificar el campo de liga de nodo **z**: z.asignaLiga(w). Por consiguiente, el conjunto de instrucciones para conectar los nodos dados, de tal manera que al recorrerlos estén en forma ascendente, son:

```
y.asignaLiga(x)
x.asignaLiga(v)
v.asignaLiga(z)
z.asignaLiga(w)
```

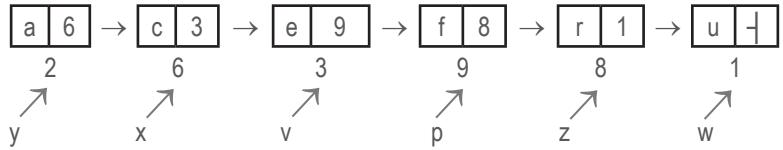
b.



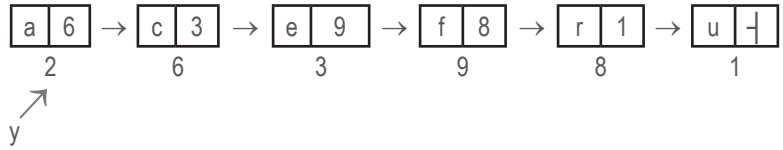
c. Para que los datos continúen ordenados la letra "f" debe quedar entre la "e" y la "r", es decir, a continuación del nodo 3, el cual se identifica con la variable **v**. Lo que se debe hacer es conseguir un nuevo nodo, almacenar en dicho nodo la letra "f" y conectarlo a continuación del nodo **v**. Para hacer esta conexión basta con asignar al campo de liga del nuevo nodo, el nodo identificado con la variable **z** y modificar el campo de liga del nodo **v** con la dirección del nuevo nodo. Digamos, por ejemplo, que vamos a identificar el nuevo nodo con la variable **p**. Las instrucciones para hacer la inserción son:

```
p = new  nodoSimple("f")
p.asignaLiga(z)
v.asignaLiga(p)
```

La lista queda:



d. Al asignar null a todas las variables, excepto a la **y**, la lista queda:



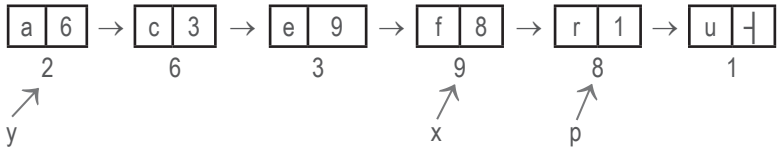
Al no tener sino la variable **y**, debemos recorrer la lista hasta encontrar un nodo cuyo dato sea la "u". Para ello utilizamos una variable auxiliar, la cual llamaremos **p**. Esta variable la inicializamos con el valor de **y** y nos movemos sobre la lista hasta encontrar un dato que sea la "u". Las instrucciones para esta tarea son:

```
p = y
while (p.retornaDato() != "u") do
        p = p.retornaLiga()
end(while)
write(p.retornaDato())
```

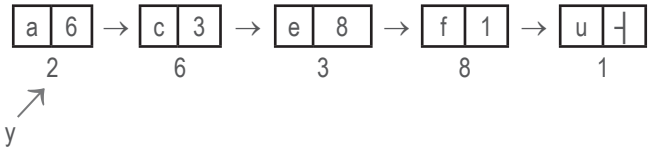

e. Para desconectar el nodo que tiene el dato "r" debemos recorrer la lista hasta encontrar el nodo que tiene dicho dato. Con un conjunto de instrucciones como el mostrado en la solución del literal d se encuentra el nodo que tiene el dato "r". Ahora, para desconectar dicho nodo debemos conocer cuál es el nodo anterior al que contiene a "r". Para lograr esto, basta con modificar un poco el conjunto de instrucciones que dan solución al literal d. Trabajaremos una variable adicional que siempre esté indicando hacia el nodo anterior a **p**, de tal manera que cuando se encuentre el nodo que tiene la "r" dicha variable esté señalando cuál es su anterior. Llamemos a esta variable **x**. El conjunto de instrucciones queda:

```
p = y
x = null
while (p.retornaDato() != "r") do
        x = p
        p = p.retornaLiga()
end(while)
x.asignaLiga(p.retornaLiga())
```

f. Al terminar de ejecutar el ciclo la situación es:



La instrucción con la que se desconecta el nodo **p** es la instrucción siguiente al end(while). La lista queda:



Autoevaluación 4

La presente autoevaluación está desarrollada con base en los ejercicios 1, 3, 5, 7, 9, 11, 13 y 15 propuestos en el módulo 4 del texto guía.

1. Elabore un algoritmo que lea un entero **n** y que construya una lista ligada, de a dígito por nodo.

   Solución:

```
1.  LSL  construyeListaDigitoPorNodo(entero  n)
2.              entero  res
3.              LSL  a = new  LSL()
4.              while  (n > 0)  do
5.                          res = n % 10
6.                          a.insertar(res, null)
7.                          n = n / 10
8.              end(while)
9.              return  a
10.  fin(construyeListaDigitoPorNodo)
```

   Definimos un método que retorne un objeto de la clase LSL. Este método lo llamamos construyeListaDigitoPorNodo. Tiene un parámetro **n** (parámetro por valor), el cual es el entero a procesar. Para descomponer el número **n** en sus dígitos usamos las operaciones módulo y división. Con la operación módulo obtenemos un dígito correspondiente a **n**, y con la división eliminamos ese dígito del número **n**. Cada vez que se extraiga un dígito lo agregamos a la lista que estamos construyendo (la lista **a**). Dicho dígito lo insertamos al principio de la lista **a** usando el método insertar definido para objetos de la clase LSL (numeral 4.3 del texto guía).

3. Se tiene una lista simplemente ligada, con un dato numérico en cada nodo. Elabore un algoritmo que determine e imprima el promedio de datos de la lista.

   Solución:

```
1.  void  promedio()
2.              entero  cont  = 0
3.               float  prom, s = 0
4.              nodoSimple  p
5.              p = primerNodo()
6.              while  !finDeRecorrido(p)  do
```

```
7.                  s  = s + p.retornaDato()
8.                  cont = cont + 1
9.                  p = p.retornaLiga()
10.         end(while)
11.         prom = s / cont
12.         write(prom)
13.  fin(promedio)
```

Definimos un método que pertenezca a la clase LSL. A dicho método lo llamamos promedio, no tiene parámetros y simplemente va a ejecutar una tarea con los datos del objeto de la clase LSL que invoca el método. Definimos variables locales: **cont**, para contar cuántos nodos tiene la lista; **s**, para sumar los datos de los nodos que hay en la lista; **prom**, para obtener el promedio; y **p,** para recorrer la lista. La variable **p**, la inicializamos utilizando el método primerNodo() definido para objetos de la clase LSL (instrucción 5), luego recorremos la lista con un ciclo (instrucción 6) acumulando el dato de cada nodo, contando los nodos de la lista (instrucciones 7 y 8) y avanzando en la lista (instrucción 9). Para detectar cuándo se terminó de recorrer la lista se utiliza el método finDeRecorrido(p) definido para objetos de la clase LSL: cuando **p** sea nulo este método retorna verdadero; por lo tanto, el resultado de evaluar la condición de la instrucción 6 es falso y termina el ciclo. Al salir del ciclo calcula el promedio (instrucción 11) y lo imprime en la instrucción 12.

5. Elabore un algoritmo que retorne el nodo que contiene el menor dato en una lista simplemente ligada.

Solución:

```
1.  nodoSimple  nodoConMenorDato(nodoSimple  antMenor)
2.          nodoSimple  y, p, menor
3.          menor = primerNodo()
4.          antMenor = anterior(menor)
5.          y = menor
6.          p = y.retornaLiga()
7.          while  (!finDeRecorrido(p))  do
8.                      if  (p.retornaDato()  <  menor.retornaDato())  then
9.                                  antMenor = y
10.                                 menor = p
11.                     end(if)
12.                     y = p
13.                     p = p.retornaLiga()
14.         end(while)
15.         return  menor
16.  fin(nodoConMenorDato)
```

Nuestro algoritmo retorna el nodo en el cual se halla el menor dato (a dicho nodo lo llamamos **menor**), además en el parámetro **antMenor** retorna el nodo anterior al menor. Nuestro algoritmo lo definimos así con el fin de que si el usuario desea manipular el nodo que contiene el dato menor, de una vez conozca el nodo anterior ya que se requiere en la manipulación de listas simplemente ligadas. Esto hace más eficientes los procesos, puesto que no tendrá que efectuar el método para buscar el anterior, el cual tiene orden de magnitud lineal.

En la instrucción 2 se definen las variables de trabajo: **p**, para recorrer la lista e ir haciendo las comparaciones de los datos; **y**, variable que apunta siempre hacia el nodo anterior a **p**; y **menor**, que es la variable donde se conserva la dirección del nodo que contiene el menor dato.

En la instrucción 3 se inicializa **menor** con el primer nodo, es decir, comenzamos suponiendo que el menor dato se halla en el primer nodo. En la instrucción 4 se inicializa el anterior al menor. La variable **y** se inicializa con **menor**, ya que la variable con la cual se recorre la lista y se efectúan las comparaciones, la variable **p**, se inicializa con el siguiente al menor (instrucción 6). En el ciclo (instrucciones 7 a 14) se compara el dato de **p** con el dato de **menor** (instrucción 8): si el dato del nodo **p** es **menor** que el dato del nodo menor se actualizan las variables **menor** y **antMenor** (instrucciones 9 y 10). Cualquiera que hubiera sido la situación se avanza en la lista con las variables **p** y **y** (instrucciones 12 y 13). Al terminar de ejecutar el ciclo se retorna **menor**, y en el parámetro **antMenor** regresa el nodo anterior al menor. Recuerde que el fin de recorrido de la lista se controla con el método finDeRecorrido(p), el cual retorna verdadero si **p** es igual a null, o falso de lo contrario.

7. Elabore un algoritmo que sume los datos impares en una lista simplemente ligada.

Solución:

```
1.  entero sumaDatosImpares()
2.          entero  s
3.          nodosimple  p
4.          s = 0
5.          while  (!finDeRecorrido(p))  do
6.                  if  (p.retornaDato() % 2  !=  0)  then
7.                          s = s + p.retornaDato()
8.                  end(if)
9.                  p = p.retornaLiga()
10.         end(while)
11.         return  s
12. fin(sumaDatosImpares)
```

Es un algoritmo realmente muy sencillo. Basta con definir una variable local en la cual se lleve el acumulado de los datos impares de la lista. A dicha variable la llamamos **s**.

Además, se utiliza la variable **p** para recorrer la lista. Dentro del ciclo, instrucciones 5 a 10, simplemente se determina si el dato del nodo **p** es impar o no. Para ello se utiliza la operación módulo (%): si el residuo de dividir el dato del nodo **p** por 2 es diferente de cero, significa que el dato del nodo **p** es impar; por lo tanto, se incluye en el acumulador de dichos datos (la variable **s**). Cualquiera que hubiera sido el resultado se avanza en la lista con la variable **p** (instrucción 9). Al terminar de ejecutar el ciclo simplemente se retorna el contenido de la variable **s**.

9. Se tienen dos listas simplemente ligadas A y B, cada una de ellas con datos numéricos en cada nodo. Los datos en cada lista son únicos, pero datos de A pueden estar en B. Elabore un algoritmo que construya una tercera lista ligada con los datos de A que están en B.

Solución:

```
1.  LSL  datosComunes(LSL  b)
2.          nodoSimple  p
3.          LSL  c = new  LSL()
4.          p = primerNodo()
5.          while  (!finDeRecorrido(p))  do
8.                  if  (b.buscarDato(p.retornaDato(), y)  !=  null)    then
9.                          c.insertar(p.retornaDato(), null)
10.                 end(if)
11.                 p = p.retornaLiga()
12.         end(while)
13.         return  c
14.  fin(datosComunes)
```

Definimos un método que retorne un objeto de la clase LSL. A dicho método lo llamamos datosComunes. Este método pertenecerá a la clase LSL; por consiguiente, deberá ser invocado por un objeto de esa clase, y tiene como parámetro un objeto de la clase LSL, el cual llamamos **b**. Nuestro método simplemente consistirá en buscar los datos que hay en la lista que invocó el método, en la lista enviada como parámetro (la lista **b**). Si lo encuentra, agregaremos en una nueva lista el dato hallado. Por simplicidad el dato se inserta al principio de la lista que tiene los datos comunes.

En la instrucción 2 definimos la variable **p** con la cual se recorrerá la lista que invocó el método. En la instrucción 3 se define y se inicializa la lista que se va a crear. En la instrucción 4 se ubica la variable **p** en el primer nodo de la lista que invocó el método. Fíjese que el llamado al método primerNodo() no está precedido de ningún objeto; por consiguiente, actúa sobre el objeto que invocó el método datosComunes. En el ciclo, instrucciones 5 a 10, se busca el dato del nodo **p** en la lista busando el método buscarDato(). Recuerde que si dicho método retorna null significa que no encontró el dato, pero si el valor retornado es diferente de null quiere decir que el dato se halla en la lista **b**. Fíjese que es el objeto **b** el que invoca el método buscarDato(). En caso de que se haya encontrado el dato en la lista **b** procedemos a incluirlo en la nueva lista (la lista **c**). Para ello, el objeto

c invoca el método insertar y le envía como parámetros el dato a insertar y null. El hecho de que le envíe null como segundo parámetro, le está indicando que el dato de **p** se debe insertar al principio de la lista. Cualquiera que hubiera sido la situación se avanza en la lista que invocó el método con la instrucción 11. Al terminar de recorrer la lista que invocó el método, es decir, salirse del ciclo de las instrucciones 5 a 12, simplemente se retorna la lista construida (instrucción 13).

11. Se tiene una lista simplemente ligada en la cual cada registro tiene un dato numérico. Los datos de la lista no están ordenados bajo ningún criterio. Elabore un algoritmo que determine los registros en los cuales se hallan el mayor y el menor dato y luego intercambie dichos registros. Considere todas las posibilidades.

Solución:

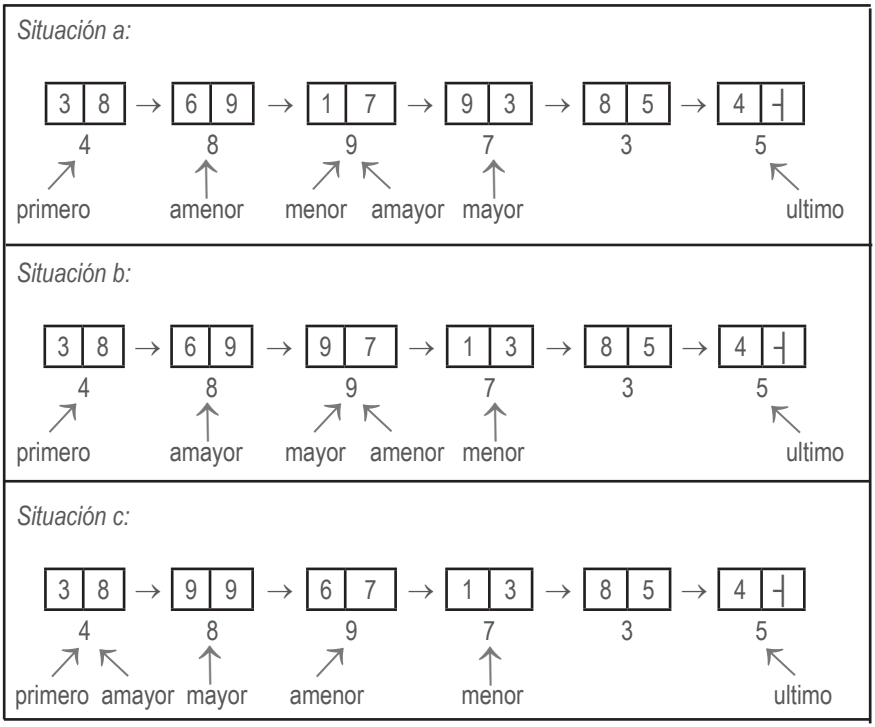
```
1.  void  intercambiaMayorYMenor()
2.          nodoSimple  menor, antMenor, mayor, antMayor
3.          menor = nodoConMenorDato(amenor)
4.          mayor = nodoConMayorDato(amayor)
5.              if  (menor == mayor)  then  return
6.                      desconectar(menor, amenor)
7.                      if  (menor  ==  amayor)  then
8.                              conectar(menor, mayor)
9.                      else
10.                     if  (mayor == amenor)  then
11.                             conectar(menor, amayor)
12.                     else
13.                             desconectar(mayor, amayor)
14.                             conectar(menor, amayor)
15.                             conectar(mayor, amenor)
16.                     end(if)
17.             end(if)
18.  fin(intercambiaMayorYMenor)
```

Esta solución muestra la importancia de utilizar métodos previamente definidos. Hay que buscar el nodo que tiene el menor dato y el nodo que tiene el mayor dato y luego intercambiarlos. Para buscar el nodo que tiene el menor dato usamos el método nodoConMenorDato(), instrucción 3, desarrollado en el ejercicio 5, y para determinar el nodo que tiene el mayor dato usamos el método nodoConMayorDato(), instrucción 4, el cual es análogo al método nodoConMenorDato(). Fíjese que ambos métodos retornan el anterior al menor y el anterior al mayor, respectivamente.

El intercambio de dichos nodos consiste en desconectar el menor y conectarlo a continuación del anterior al mayor, y desconectar el mayor y conectarlo a continuación del anterior al menor. Fíjese en la importancia que tiene el hecho de que los métodos que determinan los nodos que tienen el menor y el mayor dato retornen su respectivo anterior.

En la instrucción 5 se controlan algunas situaciones especiales: que la lista esté vacía, que la lista sólo tenga un nodo y que todos los datos de la lista sean iguales. En caso de que se presente alguna de esas situaciones no hay que efectuar ningún intercambio; por lo tanto, el método termina (instrucción return).

Cuando se determina que los nodos que tienen el menor y el mayor dato son diferentes se pueden presentar diversas situaciones, las cuales es necesario identificar con el fin de que nuestro algoritmo funcione correctamente. Dichas situaciones son: que el menor y el mayor estén consecutivos y que el menor esté antes del mayor (situación **a** de la siguiente figura); que el menor y el mayor estén consecutivos y que el menor esté después del mayor (situación **b**); y que no estén consecutivos (situación **c**).



En la instrucción 6 se desconecta el menor.

Si la situación es la **a** (instrucción 7) lo único que hay que hacer es conectar el menor a continuación del mayor (instrucción 8).

Si la situación es la **b** (instrucción 10) lo único que hay que hacer es conectar el menor a continuación del anterior al mayor (instrucción 11).

Si la situación es la **c** se procede a desconectar el mayor (instrucción 13) y a conectar el menor y el mayor a continuación del anterior al mayor y el anterior al menor, respectivamente (instrucciones 14 y 15).

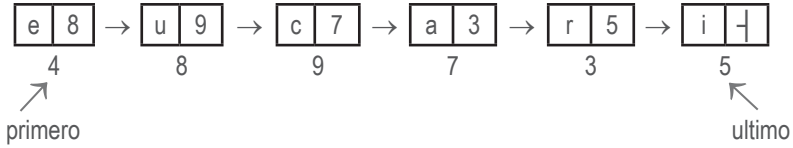
Fíjese que no nos preocupamos por el hecho de que el mayor o el menor pueda ser el primero o el último nodo. Los métodos que desarrollamos para conectar y desconectar en los numerales 4.3 y 4.4 del texto guía cubren todas estas situaciones.

13. Elabore un algoritmo que intercambie los registros de una lista simplemente ligada así: el primero con el último, el segundo con el penúltimo, el tercero con el antepenúltimo, y así sucesivamente.

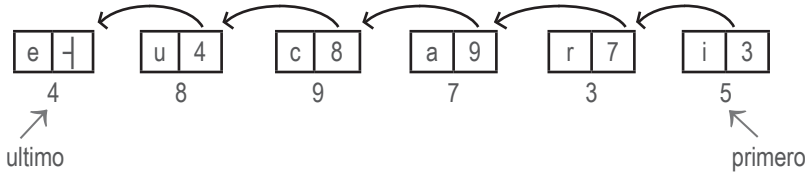
Solución:

```
1. void  intercambiaExtremos()
2.              reversaLista()
3.  fin(intercambia)
```

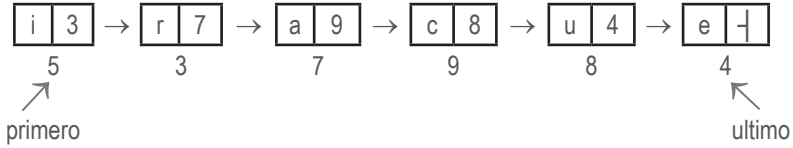
Esta es realmente una solución elegante y eficiente: intercambiar los nodos de una lista, tal como se plantea, es equivalente a reversar los apuntadores de una lista. Fíjese en las siguientes figuras:



Si reversamos los apuntadores nuestra lista queda así:

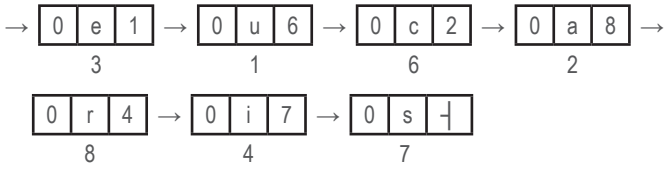


la cual, si dibujamos al derecho, queda así:

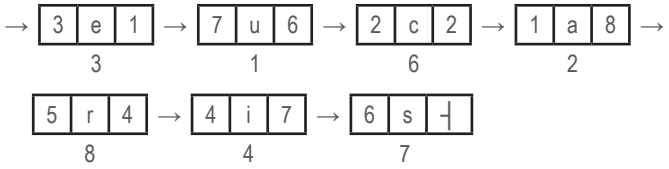


!Fíjese en lo importante que es hacer algunos análisis antes de comenzar a escribir instrucciones!

15. Se tiene una lista simplemente ligada en la cual cada nodo contiene tres campos llamados orden, dato y liga. El campo de orden inicialmente contiene un cero. Elabore un algoritmo que llene el campo de orden de tal manera que indique el lugar que ocupa en la lista, en caso de que estuvieran ordenados ascendentemente. Por ejemplo, si la lista inicial es:



Al ejecutar su algoritmo la lista debe quedar así:



Solución:

```
1.  void  actualizaOrden()
2.            entero  i, n
3.            nodoSimple  p
4.            n = longitud()
5.            i = 0
6.            while  (i < n)  do
7.                        p = primerNodo()
8.                        while  (p.retornaOrden() != 0)  do
9.                                    p = p.retornaLiga()
10.                       end(while)
11.                       menor = p
12.                       p = p.retornaLiga()
13.                       while  (!finDeRecorrido(p))  do
14.                                 if  (p.retornaOrden()  == 0)  then
15.                                       if  (p.retornaDato() < menor.retornaDato())  then
16.                                                 menor = p
17.                                       end(if)
18.                                 end(if)
19.                                 p = p.retornaLiga()
20.                       end(while)
```

```
21.                     i = i + 1
22.                     menor.asignaOrden(i)
23.             end(while)
24.   fin(actualizaOrden)
```

A nuestro método lo denominamos actualizaOrden y lo definimos perteneciente a la clase LSL, es decir, para ejecutarse debe invocarlo un objeto de esta clase.

Inicialmente el campo de orden de cada nodo tiene un cero. Para determinar el nodo que tiene el menor dato sólo se consideran los nodos que tienen el campo de orden en cero. Utilizamos una variable, la cual llamamos **i**, para asignar a cada nodo la posición que ocupa al tener los datos ordenados ascendentemente.

Primero determinamos el número de nodos en la lista utilizando la función longitud(). Este valor lo guardamos en la variable **n**, y nos servirá para determinar cuándo se ha terminado de actualizar el campo de orden de todos los nodos. La variable **i** se inicializa en cero.

Planteamos un ciclo general (instrucciones 6 a 23), con el cual controlamos el fin del proceso con la variable **i**. Cuando **i** sea igual a **n** significa que a cada nodo se le ha actualizado su campo de orden.

En las instrucciones 7 a 10 ubicamos el primer nodo que tenga su campo de orden en cero. A dicho nodo lo inicializamos como el menor, y luego con el ciclo de las instrucciones 13 a 20 detectamos el nodo que tenga el menor dato, entre todos aquellos que tengan el campo de orden en cero (instrucciones 14 a 18). Al terminar este ciclo incrementamos la variable **i** en 1 y le asignamos dicho valor al nodo que tenga el menor dato (instrucciones 21 y 22).

Autoevaluación 5

La presente autoevaluación está desarrollada con base en los ejercicios 1, 3 y 5 propuestos en el módulo 5 del texto guía.

1. Elabore un algoritmo que reste dos números de alta precisión y deje el resultado en un nuevo objeto.

   Solución:

```
1.  altaPrecision  resta(altaPrecision  b)
2.     altaPrecision  minuendo, c
3.     nodoSimple  p, q, r
4.     entero  s
5.     c = new altaPrecision()
6.     minuendo = copia()
7.     minuendo.reversaLista()
8.     p = minuendo.primerNodo()
9.     b.reversaLista()
10.    q = b.primerNodo()
11.    while  (no  b.finDeRecorrido(q))  do
12.          if  (p.retornaDato() < q.retornaDato())  then
13.               r = p.retornaLiga()
14.               while  (r.retornaDato() == 0)  do
15.                    r.asignaDato(9999)
16.                    r = r.retornaLiga()
17.               end(while)
18.               r.asignaDato(r.retornaDato() – 1)
19.               s = p.retornaDato() + 10000
20.               p.asignaDato(s)
21.          end(if)
22.          s = p.retornaDato()  –  q.retornaDato()
23.          c.insertar(s, null)
24.          p = p.retornaLiga()
25.          q = q.retornaLiga()
26.    end(while)
27.    while  (no  minuendo.finDeRecorrido(p))  do
28.          s = p.retornaDato()
29.          c.insertar(s, null)
30.          p = p.retornaLiga()
31.    end(while)
```

```
32.   b.reversaLista()
33.   c.normaliza()
34.   return  c
35. fin(resta)
```


Llamemos **A** la lista que invoca el método y **b** la lista enviada como parámetro. La lista **A** es el minuendo y la lista **b** el sustraendo. Nuestro algoritmo consiste en recorrer las dos listas simultáneamente, desde el último nodo hacia el primero, restándole el dato que hay en el nodo de la lista **b** al dato que hay en el nodo de la lista **A**. Utilizamos la variable **p** para recorrer la lista **A** y la variable **q** para recorrer la lista **b**. Al efectuar este proceso puede suceder que el nodo **p** tenga un dato menor que el nodo **q**. Cuando esto suceda le sumamos 10000 al dato del nodo **p** y le restamos 1 al dato del nodo anterior a **p**. Esta tarea la efectuamos en las instrucciones 12 a 21. Como podrá observar, la lista **A** puede ser modificada dentro de este algoritmo; por consiguiente, hacemos una copia de la lista **A** antes de efectuar el proceso de resta. Esta copia la hemos llamado **minuendo**, y la hacemos en la instrucción 6 con un método que llamamos **copia**.

En la instrucción 11 planteamos el ciclo para recorrer simultáneamente ambas listas. El ciclo termina cuando se termine de recorrer la lista **b**. Sin embargo, puede suceder que no haya terminado de recorrer la lista minuendo, y por lo tanto para cubrir esta posibilidad agregamos un ciclo en el cual terminemos de recorrer dicha lista llevando los datos de estos nodos a la lista resultado (instrucciones 27 a 30). Al efectuar el proceso de resta puede suceder que queden nodos a la izquierda con dato cero. Estos nodos afectan algoritmos como el de longitud(); por consiguiente, debemos eliminarlos. Esta tarea de eliminar ceros a la izquierda la efectuamos con un método que hemos llamado **normaliza**, el cual se invoca en la instrucción 33.

```
1.  altaPrecision  copia()
2.        nodosimple   p
3.        altaPrecision   c
4.        c = new  altaPrecision()
5.        p = primerNodo()
6.        while  (no  finDeRecorrido(p))   do
7.              c.insertar(p.retornaDato(), c.ultimoNodo())
8.              p = p.retornaLiga()
9.        end(while)
10.       return  c
11.  fin(copia)
```

Nuestro método copia es bastante simple. Basta con recorrer la lista que invoca el método (ciclo de las instrucciones 6 a 9) e ir construyendo una nueva lista, la cual llamamos **c**, e ir insertando nodos al final de ella a medida que se recorre la lista que invoca el método.

```
1.  void   normaliza()
2.      nodoSimple   p, q
3.      p = primerNodo()
4.      q = anterior(p)
5.      while (no  finDeRecorrido(p)  and   p.retornaDato() == 0)   do
6.          desconectar(p, q)
7.          p = primerNodo()
8.      end(while)
9.  fin(normaliza)
```

El método normaliza también es bastante simple. Se recorre la lista que invoca el método y se van eliminando los nodos que tienen 0 en el campo de dato. El proceso de eliminación se efectúa con el método desconectar, el cual automáticamente actualiza el campo de primero de la lista ligada.

3. Elabore un algoritmo que multiplique dos números de alta precisión dejando el resultado en un nuevo objeto.

Solución:

Para efectuar el proceso de multiplicación hemos desarrollado primero un método que hemos llamado **smult**, el cual multiplica un objeto de alta precisión por un entero entrado como parámetro.

```
1.  altaPrecision   smult(entero   d)
2.      altaPrecision    c
3.      nodoSimple    p
4.      entero   acarreo = 0, s
5.      c = new   altaPrecision()
6.      reversaLista()
7.      p = primerNodo()
8.      while   (no finDeRecorrido(p))   do
9.              s = p.retornaDato() * d + acarreo
10.             acarreo = s / 10000
11.             s = s % 10000
12.             c.insertar(s, null)
13.             p = p.retornaLiga()
14.     end(while)
15.     if  (acarreo != 0)  then
16.         c.insertar(acarreo, null)
17.     end(if)
18.     reversaLista()
19.     return   c
20.  fin(smult)
```

Este algoritmo es más bien sencillo. Basta con recorrer la lista de derecha a izquierda e ir multiplicando el dato de cada nodo por el entero entrado como parámetro. Este proceso lo efectuamos con el ciclo desde las instrucciones 8 a 14. Dentro del ciclo efectuamos el producto y le sumamos el acarreo, al resultado le calculamos el nuevo acarreo con la instrucción 10 y separamos los últimos cuatro dígitos con la instrucción 11. Estos últimos cuatro dígitos se insertan al principio de la lista resultado en la instrucción 12. Al terminar el ciclo controlamos que el acarreo sea diferente de cero para agregar este valor en la lista resultado. Este algoritmo lo desarrollamos no sólo porque en algunas ocasiones se requiere esta operación, sino que nos es de ayuda para el algoritmo de multiplicación de dos números de alta precisión que presentamos a continuación:

```
1.  altaPrecision  multiplica(altaPrecision  b)
2.     nodoSimple  x, y
3.     entero  i = 0
4.     altaPrecision  aux, c
5.     c = new  altaPrecision()
6.     b.reversa()
7.     y = b.primerNodo()
8.     while  (no  b.finDeRecorrido(y))  do
9.            aux = smult(y.retornaDato())
10.           aux.complete(i)
11.           c = c.suma(aux)
12.           i = i + 1
13.           y = y.retornaLiga()
14.    end(while)
15.    b.reversa()
16.    return  c
17.  fin(multiplica)
```

Veamos ahora nuestro algoritmo de multiplicación. Sea **A** el número que invoca el método, **b** el número pasado como parámetro y **c** la lista resultado.

Nuestro algoritmo utiliza la técnica clásica de multiplicación de dos enteros: multiplicamos el número **A** por el último dígito de **b** y almacenamos el resultado en una nueva lista; luego volvemos a multiplicar **A** por el siguiente dígito de **b**, de derecha a izquierda, y el resultado se le suma a la lista resultado teniendo cuidado de desplazar este segundo producto una posición hacia la izquierda; volvemos a multiplicar **A** por el siguiente dígito de **b**, de derecha a izquierda, y este tercer producto se le suma nuevamente a la lista resultado teniendo cuidado de desplazarlo dos posiciones hacia la izquierda para que el resultado vaya siendo correcto, y así sucesivamente.

En el ciclo de las instrucciones 8 a 14 efectuamos este proceso: recorremos la lista **b** de derecha a izquierda; multiplicamos la lista **A** por el dato de cada nodo de **b** utilizando nuestro método **smult**, y guardamos el resultado en una lista auxiliar la cual llamamos **aux**; el desplazamiento hacia la izquierda de este resultado lo logramos añadiendo tan-

tos nodos con cero al final de la lista auxiliar como posiciones haya que desplazar la lista **aux** hacia la izquierda (método que hemos denominado **complete**); le sumamos este resultado de **aux** a la lista **c**; incrementamos en uno la variable **i** (variable que indica cuántas posiciones hay que desplazar **aux** hacia la izquierda), y nos devolvemos un nodo en la lista **b**.

A continuación presentamos nuestro algoritmo **complete**:

```
1.  void  complete(entero  i)
2.        nodoSimple  p
3.        entero  j
4.        p = ultimoNodo()
5.        for  (j = 1; j <= i; j++)  do
6.              insertar(0, p)
7.        end(for)
8.  fin(complete)
```

Es un algoritmo bastante sencillo; basta con hacer un ciclo desde 1 hasta **i** agregando nodos con dato cero al final de la lista que invoca el método.

5. Elabore un algoritmo que calcule y retorne el factorial de un número de alta precisión. El resultado debe ser otro objeto de la clase alta precisión.

Solución:

```
1.  altaPrecision  factorial()
2.        altaPrecision  ff = new  altaPrecision()
3.        altaPrecision  gg, hh
4.        ff.insertar(1, ff.primerNodo())
5.        gg = copia()
6.        hh = copia()
7.        gg = gg.resta(ff)
8.        while  (gg.compare(ff)  == 1)  do
9.                  hh = hh.multiplica(gg)
10.                 gg = gg.resta(ff)
11.       end(while)
12.       return  hh
13.  fin(factorial)
```

Nuestro algoritmo para el factorial funciona en la forma tradicional:

$$n! = n*(n-1)!$$

Creamos un número de alta precisión llamado **ff** con valor 1. Hacemos una doble copia

del número que invocó el método. Dichas copias las llamamos **gg** y **hh**. En la variable **hh** guardamos el resultado y a la variable **gg** le vamos restando **ff** hasta llegar a 1. Dentro del ciclo vamos multiplicando **hh** por **gg** hasta que **gg** sea 1. El método compare utilizado en la instrucción 8 es el siguiente:


```
1.   entero  compare(altaPrecision  b)
2.          entero  na,  nb
3.          nodoSimple  p, q
4.          na = longitud()
5.          nb = b.longitud()
6.          if  (na > nb)  then
7.               return  +1
8.          end(if)
9.          if  (na < nb)  then
10.              return  –1
11.         end(if)
12.         p = primerNodo()
13.         q = b.primerNodo()
14.         while ((no  finDeRecorrido(p))  and  (no  b.finDeRecorrido(q))  and
                          (p.retornaDato() ==  q.retornaDato()))  do
15.                  p = p.retornaLiga()
16.                  q = q.retornaLiga()
17.         end(while)
18.         if  (finDeRecorrido(p))  then
19.              return  0
20.         end(if)
21.         if (p.retornaDato() > q.retornaDato())  then
22.              return  +1
23.         end(if)
24.         return  –1
25.   fin(compare)
```

Para explicar este método llamemos **A** el objeto que invoca el método y **b** el objeto pasado como parámetro. Lo primero es determinar la longitud de cada lista, es decir, el número de nodos en cada una de las listas. La lista que tenga más nodos es la que representa el número mayor. Llamemos **na** y **nb** el número de nodos en **A** y **b**, respectivamente. Si **na** es mayor que **nb** retorna +1 (instrucciones 6 y 7); si **na** es menor que **nb** retorna –1 (instrucciones 9 y 10); si el número de nodos de **A** es igual al número de nodos de **b** recorremos ambas listas simultáneamente comparando los datos de cada nodo (ciclo de las instrucciones 14 a 17) hasta encontrar una desigualdad o hasta que termine de recorrer ambas listas. Si terminó de recorrer ambas listas simultáneamente retorna 0 (instrucciones 18 y 19), de lo contrario compara los datos de los nodos donde encontró la desigualdad y retorna +1 o –1 según la situación.

Autoevaluación 6

La presente autoevaluación está desarrollada con base en los ejercicios 1, 3, 5 y 7 propuestos en el módulo 6 del texto guía.

11. Elabore algoritmos para los diferentes métodos correspondientes al proceso de inserción en listas simplemente ligadas circulares.

Solución:

```
1.  nodoSimple  buscaDondeInsertar(objeto  d)
2.          nodoSimple  p, y
3.          p = primerNodo()
4.          if  (p == null)  then
5.                  return  null
6.          end(if)
7.          y = null
8.          if  (d < p.retornaDato())  then
9.                  return  y
10.         end(if)
11.         p = p.retornaLiga()
12.         while  (p  !=  primerNodo()  and  p.retornaDato() < d)  then
13.                 y = p
14.                 p = p.retornaLiga()
15.         end(while)
16.         return  y
17.  fin(buscaDondeInsertar)
```

En la instrucción 2 definimos las variables de trabajo: **p** para recorrer la lista e ir comparando el dato de **p** con el dato a insertar, y **y** la variable que siempre apuntará hacia el nodo anterior a **p**. Recuerde que nuestro método buscaDondeInsertar(d) retorna el nodo a continuación del cual se debe insertar un nuevo nodo con dato **d**.

En la instrucción 3 se le asigna a **p** el primer nodo. En caso de que la lista esté vacía la variable **p** queda en null, en cuyo caso se debe retornar null (instrucciones 4 a 6).

En la instrucción 7 se inicializa **y** en null (recuerde que **y** es el anterior a **p**. Si **p** es el primer nodo **y** es null, ya que el primer nodo no tiene anterior).

En la instrucción 8 se controla que el dato a insertar sea menor que el dato del primer nodo. En caso de serlo, significa que el nuevo dato debe insertarse al principio de la lista;

por consiguiente, se retorna null (el valor de **y**), indicando que el dato a insertar debe quedar al principio de la lista.

En caso de que aún no haya terminado el subprograma, es decir, que haya ejecutado algún return, significa que hay que recorrer la lista buscando dónde insertar el dato **d**. Para ello se efectúa el ciclo de las instrucciones 12 a 15, en el cual en la instrucción while se controla el fin de recorrido y que el dato de **p** sea menor que el dato d. Fíjese que el fin de recorrido se presenta cuando **p** sea igual al primer nodo, ya que la lista es circular. Fíjese también que al ciclo se entra con **p** en el segundo nodo, en virtud de la instrucción 11. Al terminar de ejecutar el ciclo se retorna el contenido de **y**.

```
1.  void  insertar(objeto  d, nodoSimple y)
2.            x = new  nodoSimple(d)
3.            conectar(x, y)
4.   fin(insertar)
```

Nuestro método insertar es idéntico a cuando se tienen las listas simplemente ligadas. Basta con conseguir un nuevo nodo, cargarlo con el dato **d** e invocar el método para conectar el nuevo nodo en la lista.

```
1.  void  conectar(nodoSimple  x, nodoSimple y)
2.            if  (y == null)  then
3.                        x.asignaLiga(primero)
4.                        if  (primero == null)  then
5.                                    ultimo = x)
6.                        end(if)
7.                        primero = x
8.                        ultimo.asignaLiga(x)
9.            else
10.                       x.asignaLiga(y.retornaLiga())
11.                       y.asignaLiga(x)
12.                       if  (y == ultimo)  then
13.                                   ultimo = x
14.                       end(if)
15.           end(if)
16.  fin(conectar)
```

Nuestro método conectar(x, y) es similar al de conectar en las listas simplemente ligadas. La única diferencia se presenta en la situación en la que hay que conectar el nodo **x** al principio de la lista (y == null). En este caso hay que actualizar también el campo de liga del último nodo, ya que la lista es circular (instrucción 8).

3. Elabore algoritmos para los diferentes métodos correspondientes al proceso de inserción en listas simplemente ligadas con nodo cabeza.

Solución:

En las listas simplemente ligadas con nodo cabeza el proceso de inserción se simplifica debido a que la situación de que la variable **y** sea null no se presenta. Recuerde que en este tipo de listas (numeral 6.4 del módulo 6 del texto guía) la lista vacía es el nodo cabeza.


```
1.  nodoSimple  buscaDondeInsertar(objeto  d)
2.          nodoSimple  p, y
3.          y = nodoCabeza()
4.          p = primerNodo()
5.          while  (no finDeRecorrido(p)  and  p.retornaDato() < d)  then
6.                  y = p
7.                  p = p.retornaLiga()
8.          end(while)
9.          return  y
10.  fin(buscaDondeInsertar)
```

Nuestro método buscaDondeInsertar(objeto **d**) utiliza las mismas dos variables auxiliaries **p** y **y**: la variable **p** para recorrer la lista e ir comparando el dato de **p** con el dato **d**, y **y** que siempre apunta hacia el anterior a **p**. Sus valores iniciales se asignan en las instrucciones 3 y 4. Fíjese que **y** nunca será null, su valor inicial es el nodo cabeza. El ciclo de las instrucciones 5 a 8 es similar al de buscar dónde insertar en las listas simplemente ligadas (numeral 4.3 del módulo 4 del texto guía).

```
1.  void  insertar(objeto  d, nodoSimple y)
2.          x = new  nodoSimple(d)
3.          conectar(x, y)
4.  fin(insertar)
```

Nuestro método insertar no presenta ningún problema. Es idéntico al de las otras clases de listas.

```
1.  void  conectar(nodoSimple  x, nodoSimple y)
2.          x.asignaLiga(y.retornaLiga())
3.          y.asignaLiga(x)
4.          if  (y == ultimo)  then
5.                  ultimo = x
6.          end(if)
7.  fin(conectar)
```

Nuestro método conectar se simplifica bastante con respecto al de la listas simplemente ligada ya que no hay que considerar la situación en la que **y** sea null. La variable **y** nunca será null debido a que la lista tiene nodo cabeza; por consiguiente, la variable primero no sufrirá modificación alguna, ya que el nodo cabeza es inamovible.

5. Elabore algoritmos para los diferentes métodos correspondientes al proceso de borrado en listas simplemente ligadas circulares con nodo cabeza.

Solución:

En las listas simplemente ligadas circulares con nodo cabeza el proceso de borrado también se simplifica bastante por la misma razón que en las listas simplemente ligadas con nodo cabeza: **y** nunca será null.

```
1.  nodoSimple  buscarDato(objeto  d, nodoSimple  y)
2.          nodoSimple  x
3.          p = primerNodo()
4.          x = anterior(p)
5.          while  (no finDeRecorrido(x)  and  x.retornaDato() != d)  do
6.                  y = x
7.                  x = x.retornaLiga()
8.          end(while)
9.          return  x
10.  fin(buscarDato)
```

Nuestro método buscar dato es idéntico al desarrollado para las listas simplemente ligadas (numeral 4.4 del módulo 4 del texto guía). La diferencia se presenta en que cuando se ejecuta el método primerNodo(), éste retorna el nodo siguiente al nodo cabeza (numeral 6.3 del módulo 6 del texto guía), y cuando ejecuta el método anterior retornará el nodo cabeza. Además, el fin de recorrido se presenta cuando el nodo **p** sea igual a primero (numeral 6.3 del módulo 6 del texto guía).

```
1.  void  borrar(nodoSimple  x, nodoSimple  y)
2.          if  (x == null)  then
3.                  write("dato no existe")
4.                  return
5.          end(if)
6.          desconectar(x, y)
7.  fin(borrar)
```

Nuestro método borrar controla simplemente que **x** sea diferente de null. En caso de que lo sea invoca el método para desconectar el nodo **x**, cuyo anterior es el nodo **y**.

```
1.  void  desconectar(nodoSimple  x, nodoSimple  y)
2.          y.asignaLiga(x.retornaLiga())
3.          if  (x == ultimo)  then
4.                  ultimo = y
5.          end(if)
6.  fin(desconectar)
```

Nuestro método desconectar es bastante simple debido a que no hay que controlar que la variable **y** sea null. Basta con modificar el campo de liga del nodo **y** y controlar que éste no hubiera sido el último nodo. Si **x** era el último nodo actualiza la variable ultimo.

7. Elabore un algoritmo para el método anterior(x). Su método debe pertenecer a la clase LSL y debe funcionar correctamente tanto para la clase LSL como para la clase LSLC-CRC.

Solución:

```
1.  nodoSimple  anterior(nodoSimple  x)
2.          nodoSimple  p, ant
3.          if (primero == null)
4.                  return(null)
5.          end(if)
6.          if (esVacia())
7.                  return(primero)
8.          end(if)
9.          p = primerNodo()
10.         if (p == primero)
11.                 ant = null
12.         else
13.                 ant = primero
14.         end(if)
15.         while  (p != x)  do
16.                 ant = p
17.                 p = p.retornaLiga()
18.         end(while)
19.         return  ant
20.  fin(anterior)
```

En la instrucción 3 controlamos que primero sea null. La variable primero podrá ser null sólo si la lista es simplemente ligada y está vacía. Si la lista es simplemente ligada circular con nodo cabeza la variable primero nunca será null, ya que en las listas simplemente ligadas con nodo cabeza la lista vacía es el nodo cabeza (primero) con el campo de liga apuntando hacia sí mismo. En otras palabras, en la instrucción 3 se cubre el hecho de que la lista sea simplemente ligada y esté vacía.

Con la instrucción 6 se cubre la posibilidad de que la lista sea simplemente ligada circular con nodo cabeza y que esté vacía. En este caso retorna la dirección de primero, ya que este nodo será el anterior a cualquier nodo que se desee insertar al principio de la lista.

En la instrucción 9 a la variable **p** se le asigna el primer nodo: si **p** queda valiendo primero significa que se trata de una lista simplemente ligada; si no, se trata de una lista

simplemente ligada circular con nodo cabeza y la variable **p** quedará apuntando hacia el nodo siguiente a primero.

Con el if de las instrucciones 10 a 14 le asignamos el valor a la variable **ant**: si la lista es simplemente ligada **ant** será null, pero si la lista es simplemente ligada circular con nodo cabeza la variable **ant** será primero.

Y, por último, en el ciclo de las instrucciones 15 a 18 se recorre la lista comparando la variable **p** con **x**. Cuando **p** sea igual a **x** termina el ciclo y retorna **ant**.

Autoevaluación 7

La presente autoevaluación está desarrollada con base en el ejercicio 2 propuesto en el módulo 7 del texto guía.

2. Dados los siguiente nodos conectados y la variable x apuntando hacia el nodo 5 (figura 7.1):

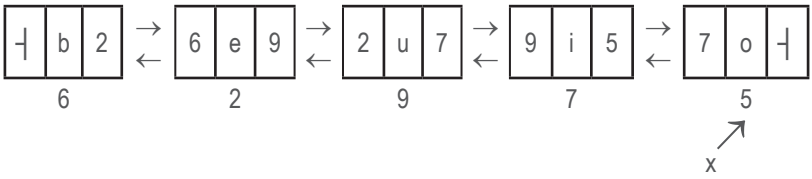


Figura 7.1

a. Escriba un algoritmo que imprima el dato del nodo 2.
b. Escriba un algoritmo que desconecte el nodo 9.
c. Escriba un algoritmo que inserte un nuevo nodo con dato "a" antes del nodo 6.
d. Escriba un algoritmo que escriba los datos de todos los nodos dibujados.

Solución:

a.

```
1. p = x.retornaLi()
2. while  (p != 2)  do
3.             p = p.retornaLi()
4. end(while)
5. write(p.retornaDato()
```

Como la única variable conocida es la variable **x**, debemos recorrer la lista partiendo de dicha variable. Para ello utilizamos una variable auxiliar, la cual llamamos **p**. El valor inicial de **p** es el nodo anterior a **x**, y como se puede observar en la figura 7.1 para llegar hasta el nodo 2 debemos movernos con el campo de liga izquierda (retornaLi()) del nodo **p**. Cuando **p** sea igual a 2 se termina el ciclo (instrucciones 2 a 4) y procedemos a escribir el dato del nodo 2 (instrucción 5).

b.

```
1. p = x.retornaLi()
2. while  (p != 9)  do
```

```
3.              p = p.retornaLi()
4.  end(while)
5.  p.retornaLi().asignaLd(p.retornaLd())
6.  p.retornaLd().asignaLi(p.retornaLi())
```


Nuevamente nos apoyamos en la figura 7.1: para desconectar el nodo 9 debemos ubicarnos con una variable en dicho nodo. Llamemos a esta variable **p**, y la inicializamos con el nodo anterior a **x** (instrucción 1). Luego recorremos la lista con **p**, moviéndonos de izquierda a derecha hasta ubicarnos en el nodo 9. Cuando estemos ubicados en este nodo estamos en una situación como en la figura 7.2, de acuerdo con la propiedad fundamental de las listas doblemente ligadas (numeral 7.2 del módulo 7 del texto guía).

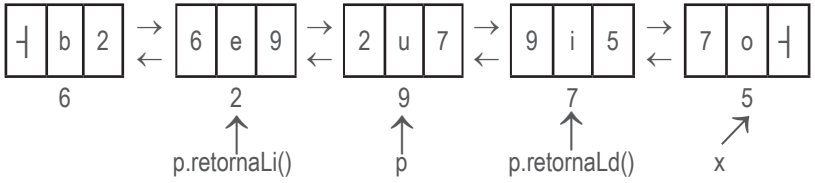


Figura 7.2

Para desconectar el nodo **p** basta con actualizar dos campos de liga: el campo liga derecha del nodo p.retornaLi(), y el campo liga izquierda del nodo p.retornaLd(). El campo liga derecha del nodo p.retornaLi() debe quedar valiendo 7, es decir, el contenido del campo liga derecha del nodo **p**, y el campo liga izquierda del nodo p.retornaLd() debe quedar valiendo 2, es decir, el contenido del campo liga izquierda del nodo **p**. Las instrucciones para modificar estos dos campos de liga son:

```
p.retornaLi().asignaLd(p.retornaLd())
                    y
p.retornaLd().asignaLi(p.retornaLi())
```

c.

```
1.  p = x.retornaLi()
2.  while  (p != 6)  do
3.              p = p.retornaLi()
4.  end(while)
5.  y = new  nodoDoble("a")
6.  y.asignaLd(p)
7.  p.asignaLi(y)
```

De nuevo nos apoyamos en la figura 7.1. Utilizamos la variable **p** para ubicarnos en el nodo 6 (instrucciones 1 a 4). Al estar **p** en el nodo 6 y ejecutar la instrucción 5 estamos en una situación como en la figura 7.3:

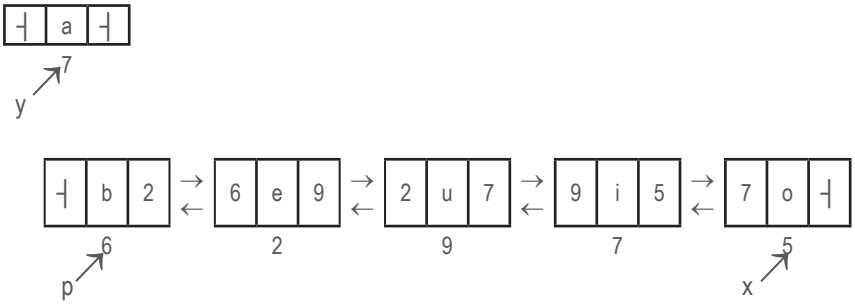


Figura 7.3

Estando en esta situación hay que actualizar el campo liga derecha del nodo **y** y el campo liga izquierda del nodo **p**. Dichas actualizaciones se logran con las instrucciones 6 y 7 respectivamente, y la lista queda así (figura 7.4):

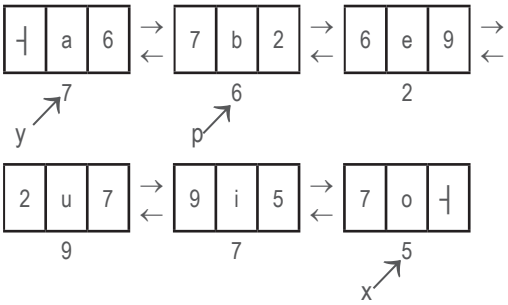


Figura 7.4

d.

```
1. p = x
2. while (p != null) do
3.             write(p.retornaDato())
4.             p = p.retornaLi()
5. end(while)
```

Para resolver este ejercicio simplemente utilizamos una variable auxiliar, la cual llamamos **p**. Dicha variable la inicializamos con **x** y recorremos los nodos conectados de derecha a izquierda, es decir, utilizando el campo liga izquierda de cada nodo para trasladarnos hacia el anterior. A medida que vamos recorriendo la lista se va imprimiendo el dato de cada nodo.

Autoevaluación 8

La presente autoevaluación está desarrollada con base en los ejercicios 1, 3, 5, 7, 9 y 11 propuestos en el módulo 8 del texto guía.

1. Defina una clase alta precisión derivada de la clase LDLCCRC, tal que en el nodo cabeza se almacene un 0 o un 1 dependiendo de si el número a representar es positivo o negativo. Elabore métodos para sumar, restar, multiplicar y dividir números de alta precisión bajo esta representación.

   Solución:

   Comenzaremos definiendo la clase altaPrecision derivada de la clase lista doblemente ligada circular con nodo cabeza.

```
Clase altaPrecisión derivada de LDLCCRC
Público
  altaPrecision  construyeNúmero(hilera  S)
  void  imprimeAltaPrecision()
  altaPrecision  suma(altaPrecision  b)
  altaPrecisión  resta(altaPrecision  b)
  altaPrecisión  multiplica(altaPrecision  b)
  altaPrecision  smult(entero  d)
  entero  compare(altaPrecision  b)
  altaPrecisión  divide(altaPrecision  b)
  altaPrecision  modulo(altaPrecision  b)
  void  evalua(entero d1, entero d2, entero acarreo)
  void  complete(entero  i)
  void  normaliza()
  boolean  signo()
  altaPrecision  factorial()
  altaPrecision  raizCuadrada()
fin(altaPrecision)
```

   En el texto guía hemos desarrollado los algoritmos correspondientes a los métodos para construye número, imprime alta precisión, suma y evalúa usando la representación como lista simplemente ligada. En la autoevaluación del módulo 5 se desarrollaron los métodos resta, normaliza, smult, multiplica, complete, compare y factorial usando la representación como lista simplemente ligada.

   En general, los algoritmos para estos métodos son los mismos cuando se tiene la repre-

sentación como lista doblemente ligada con nodo cabeza. Sólo hay que tener en cuenta el signo de los números con los cuales se va a operar, ya que las operaciones propias de las listas ligadas se resuelven invocando los métodos definidos en la clase base: la clase lista doblemente ligada con nodo cabeza.

Dado lo anterior, presentamos en este módulo el algoritmo correspondiente a la división. Es conveniente anotar que la idea central del desarrollo de este algoritmo es de Luis Fernando Suárez, estudiante, ya graduado, de Ingeniería de Sistemas de la seccional del Suroeste.

```
1.   altaPrecision  dividir(altaPrecision  b)
2.          altaPrecision   d, c
3.          entero   i, n
4.          nodoDoble   p, o, y
5.          c = new  altaPrecision()
6.          o = c.cabeza()
7.          p = cabeza()
8.          y = b.cabeza()
9.          o.asignaDato((p.retornaDato() + y.retornaDato()) % 2)
10.         if   (compare(b) == –1)
11.               return  c
12.         end(if)
13.         p = primerNodo()
14.         n = b.longitud()
15.         d = new  altaPrecision()
16.         for  (i = 1; i <= n; i++)   do
17.               d.insertar(p.retornaDato(), d.ultimoNodo())
18.               p = p.retornaLd()
19.         end(for)
20.         while  (no  finDeRecorrido(p))   do
21.                i = d.esta(b)
22.               c.insertar(i, c.ultimoNodo())
23.               d = d.resta(b.smult(i))
24.               d.insertar(p.retornaDato(), d.ultimoNodo())
25.               p = p.retornaLd()
26.         end(while)
27.         i = d.esta(b)
28.         c.insertar(i, c.ultimoNodo())
29.         return  c
30.  fin(dividir)
```

Nuestro algoritmo de división funciona como la forma tradicional de dividir dos enteros.

El signo del resultado de una división es positivo si ambos números tienen el mismo signo, o negativo si son de diferente signo. El signo del resultado lo obtenemos con la instrucción 9.

Sea **A** el dividendo (es decir, el número que invoca el método), **b** el divisor y **c** el cociente.

Nuestro primer control es averiguar si **A** es menor que **b**. Si esto es verdadero simplemente retornamos la lista **c** vacía (instrucciones 10 a 12).

Utilizamos una lista auxiliar, la cual llamamos **d**, que nos va a servir para efectuar la división. Inicialmente esta lista la construimos con la misma cantidad de nodos que tiene el divisor **b**. Luego efectuamos el ciclo principal del método (instrucciones 20 a 25): determinamos cuántas veces está **b** en **d** (instrucción 21); insertamos este valor, el cual llamamos **i**, al final de la lista resultado (instrucción 22; actualizamos **d** multiplicando el divisor **b** por **i** y restándole este resultado a **d** (instrucción 23); añadimos el siguiente nodo de **A** a **d** (instrucción 24), y avanzamos con **p** en la lista **A**.

Al terminar el ciclo determinamos el último nodo para el cociente, lo insertamos al final de **c** (instrucciones 27 y 28) y retornamos la lista **c**.

El método para determinar cuántas veces está el divisor en el dividendo auxiliar **d** lo hemos llamado **esta** y lo presentamos a continuación:

```
1.  entero  esta(altaPrecision  b)
2.          entero  x = 10000
3.          while  (compare(b.smult(x))  == –1)  do
4.                      x = x – 1000
5.          end(while)
6.          while  (compare(b.smult(x))  != –1)  do
7.                      x = x + 100
8.          end(while)
9.          x = x – 100
10.         while  (compare(b.smult(x))  != –1)  do
11.                     x = x + 10
12.         end(while)
13.         x = x – 10
14.         while  (compare(b.smult(x))  != –1)  do
15.                     x = x + 1
16.         end(while)
17.         x = x – 1
18.         return  x
19.  fin(esta)
```

Dejamos al estudiante la tarea de comprobar la correctitud de este algoritmo.

3. Elabore un algoritmo que intercambie los registros de una lista doblemente ligada así: el primero con el segundo, el tercero con el cuarto, el quinto con el sexto y así sucesivamente.

Solución:

```
1.  void  intercambiaConsecutivos()
2.            nodoDoble  x, y
3.            if  (esVacia())  then
4.                        return
5.            end(if)
6.            x = primerNodo()
7.            y = x.retornaLiga()
8.            wnile  (y  !=  null)  do
9.                        desconectar(x)
10.                       conectar(x, y)
11.                       x = x.retornaLd()
12.                       if  (x  != null)  then
13.                                 y = x.retornaLd()
14.                       else
15.                                 y = null
16.                       end(if)
17.           end(while)
18.  fin(intercambiaConsecutivos)
```

Nuestro método lo llamamos intercambiaConsecutivos(), no tiene parámetros y es tipo void, es decir, efectúa una tarea sobre el objeto que invoca el método.

En la instrucción 2 definimos las variables locales con las cuales se va a trabajar. En las instrucciones 3 a 5 se controla que la lista no esté vacía. En caso de estarlo simplemente se retorna al programa llamante puesto que no hay ninguna labor para efectuar.

Las variables **x** y **y** se inicializan en los dos primeros nodos: **x** será el primer nodo y **y** el siguiente.

Nuestro algoritmo consiste simplemente en recorrer la lista e ir desconectando el nodo **x** y conectarlo a continuación del nodo **y** (instrucciones 9 y 10); de esta manera quedan intercambiados los nodos **x** y **y**. Se avanza con **x**, y a **y** se le asigna el siguiente a **x** en caso de que **x** sea diferente de null. La terminación del ciclo de las instrucciones 8 a 17 se controla con la variable **y**, ya que ésta es la que va delante de **x**. Cuando **y** sea null significa que se terminó de recorrer la lista, y por ende de ejecutar los intercambios. Con la instrucción 12 se controla que el número de nodos en la lista sea par o impar: al avanzar con **x** en la instrucción 11 puede suceder que **x** quede valiendo null o la dirección de un nodo. Si **x** queda valiendo null se le asigna null a **y** con la instrucción 15, si **x** es diferente de null se le asigna a **y** el siguiente de **x**. En caso de que el número de nodos fuera impar la variable **y** queda valiendo null y el nodo **x** no tiene con quién intercambiarlo y el proceso termina.

5. Elabore un algoritmo que intercambie los registros de una lista doblemente ligada así: el primero con el tercero, el segundo con el cuarto, el quinto con el séptimo, el sexto con el octavo, y así sucesivamente.

Solución:


```
1.  void intercambioPorParejas()
2.          nodoDoble  x, y, z, p
3.          if  (esVacia())  then
4.                  return
5.          end(if)
6.          x = primero
7.          y = x.retornaLd()
8.          if  (y == null)  then  return
9.          z = y.retornaLd()
10.         if  (z != null)  then
11.                 p = z.retornaLd()
12.         else
13.                 p = null
14.         end(if)
15.         while  (z != null  and  p != null)  do
16.                 desconectar(x)
17.                 conectar(x, p)
18.                 desconectar(y)
19.                 conectar(y, x)
20.                 x = y.retornaLd()
21.                 if  (x == null)  then  return
22.                 y = x.retornaLd()
23.                 if  (y == null)  then  return
24.                 z = y.retornaLd()
25.                 if  (z != null)  then
26.                         p = z.retornaLd()
27.                 else
28.                         p = null
29.                 end(if)
30.         end(while)
31.         if  (p == null  and  z != null)  then
32.                 p = x.retornaLi()
33.                 desconectar(x)
34.                 conectar(x, y)
35.                 desconectar(z)
36.                 conectar(z, p)
37.         end(if)
38. fin(intercambioPorParejas)
```

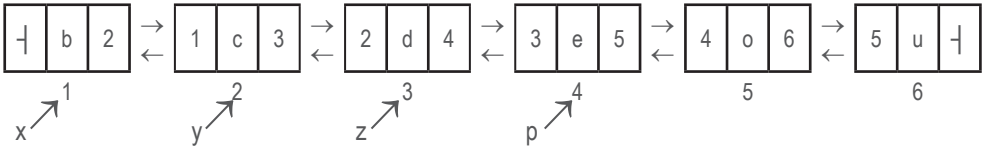
Nuestro método lo llamamos intercambioPorParejas(), no tiene parámetros y es tipo void, es decir, efectúa una tarea sobre el objeto que invoca el método.

En la instrucción 2 definimos nuestras variables de trabajo: **x** apunta hacia el primer nodo, **y** hacia el segundo nodo, **z** hacia el tercer nodo y **p** hacia el cuarto nodo. Se intercambiarán el nodo **x** con el nodo **z**, y el nodo **y** con el nodo **p**.

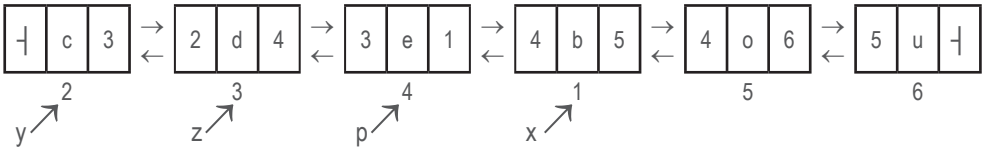
En la instrucción 3 se controla que la lista no esté vacía. En caso de estarlo simplemente retorna (instrucción 4) puesto que no hay nada para hacer.

En las instrucciones 6 a 14 se asignan los valores iniciales a las variables **x**, **y**, **z** y **p**, controlando que la lista tenga al menos tres nodos. En caso de que lista tenga tres nodos no se ejecuta el ciclo de los intercambios (instrucciones 15 a 30). En caso de que la lista tenga como mínimo cuatro nodos ejecutará el ciclo de los intercambios.

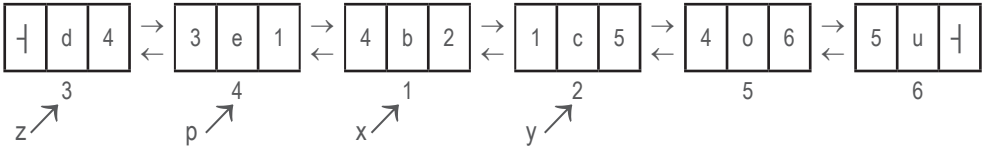
Para explicar las instrucciones del ciclo y sus controles consideremos la siguiente lista:



Al ejecutar las instrucciones 16 y 17 la lista queda así (fíjese que el nodo **x** ha quedado a continuación del nodo **p**):



Y al ejecutar las instrucciones 18 y 19 la lista queda así:



Observe que el resultado de efectuar esas instrucciones fue que se intercambió el nodo 1 con el 3 y el 2 con el 4.

En las instrucciones 20 a 29 se actualizan las variables **x**, **y**, **z** y **p**, teniendo cuidado de que la actualización sea válida. Si **x** o **y** quedan en nulo no se podrán efectuar más intercambios; por lo tanto, se retorna al programa llamante (instrucciones 21 y 23). Si **z**

queda diferente de null y **p** queda valiendo null se sale del ciclo de intercambios y efectúa el intercambio de **x** con **z** en las instrucciones 31 a 37.

Es supremamente importante que note la ventaja de usar los métodos para conectar y desconectar nodos en una lista doblemente ligada: la labor de programación es mucho más sencilla y legible.

7. Elabore algoritmos para los diferentes métodos correspondientes al proceso de inserción en listas doblemente ligadas circulares con nodo cabeza.

Solución:

```
1.  nodoDoble  buscaDondeInsertar(objeto  d)
2.          nodoDoble  p, y
3.          p = primerNodo()
4.          y = nodoCabeza()
5.          while  (no finDeRecorrido(p)  and  p.retornaDato < d)  do
6.                  y = p
7.                  p = p.retornaLd()
8.          end(while)
9.          return  y
10.  fin(buscaDondeInsertar)
```

El método para buscar dónde insertar es similar al desarrollado en el texto guía para las listas doblemente ligadas (numeral 8.2). La diferencia es que cuando ejecuta el método primer nodo (instrucción 3), éste retorna el nodo siguiente al nodo cabeza; por consiguiente, el nodo anterior al primer nodo es el nodo cabeza, el cual se obtiene al ejecutar la instrucción 4.

El ciclo de búsqueda es exactamente el mismo que en las listas doblemente ligadas, con la diferencia de que cuando se ejecuta el método finDeRecorrido(p) éste retorna verdadero cuando **p** sea igual a primero, es decir, el nodo cabeza.

```
1.  void  insertar(objeto  d,  nodoDoble  y)
2.          nodoDoble  x
3.          x = new  nodoDoble(d)
4.          conectar(x, y)
5.  fin(insertar)
```

Este método no creo que amerite explicación alguna. Es idéntico al de la lista doblemente ligada.

```
1.  void  conectar(nododoble  x,  nodoDoble  y)
2.          x.asignaLd(y.retornaLd())
3.          x.asignaLi(y)
```

```
4.          y.asignaLd(x)
5.          x.retornaLd().asignaLi(x)
6.          if  (y == ultimo)  then
7.                  ultimo = x
8.          end(if)
9.  fin(conectar)
```


El método conectar(x, y) se simplifica con respecto al método para la lista doblemente ligada debido a que la variable primero no sufre ninguna modificación, ya que ésta representa el nodo cabeza, el cual es inamovible. Además la variable **y** nunca es null por la misma razón. Cuando se va a conectar al principio de la lista significa que se conecta a continuación del nodo cabeza.

9. Elabore algoritmos para los diferentes métodos correspondientes al proceso de borrado en listas doblemente ligadas circulares.

Solución:

```
1.  nodoDoble  buscarDato(objeto  d)
2.          nodoDoble  p
3.          if  esVacia()  then  return  null
4.          p = primerNodo()
5.          if  (p.retornaDato() == d)  then
6.                  return  p
7.          end(if)
8.          p = p.retornaLiga()
9.          while  (no finDeRecorrido(p)  and  p.retornaDato() != d)  do
10.                 p = p.retornaLd()
11.         end(while)
12.         if  (finDeRecorrido(p))  then
13.                 return  null
14.         end(if)
15.         return  p
16.  fin(buscarDato)
```

En la instrucción 2 defrinimos nuestra variable de trabajo **p**.

En la instrucción 3 se controla que la lista no esté vacía. Si la lista está vacía retorna null. Recuerde que nuestro método buscar dato retorna null si no encuentra el dato en la lista. En la lista vacía es obvio que no se halla el dato **d**. Recuerde además que una lista doblemente ligada circular es vacía cuando primero sea null.

En la instrucción 4 se le asigna a **p** el primer nodo, es decir, el contenido de la variable primero. Como la lista es circular, ésta se termina de recorrer cuando la variable **p** vuelva a ser igual a primero. Esta es la razón por la que controlamos antes del ciclo que el dato

de **p** sea igual a **d**. Si esto sucede significa que el dato a buscar se halla en el primer nodo de la lista y de una vez se retorna **p**.

Si el dato del nodo **p** es diferente de **d**, avanzamos con **p** hacia el siguiente nodo y efectuamos el ciclo de las instrucciones 9 a 11. Dicho ciclo se ejecuta hasta que **p** vuelva a ser primero (finDeRecorrido(p)) o hasta que el dato de **p** sea igual a **d**. Cuando se salga del ciclo controlamos por cuál condición fue que se terminó el ciclo: si la condición de la instrucción 12 es verdadera retorna null, puesto que terminó de recorrer la lista y no encontró el dato **d**; si la condición es falsa significa que encontró el dato **d**; por lo tanto, ejecuta la instrucción 15 retornando el nodo en el cual se encuentra **d**.


```
1.  void  borrar(nodoDoble  x)
2.            if  (x == null)  then
3.                        write("dato no existe")
4.                        return
5.            end(if)
6.            desconectar(x)
7.  fin(borrar)
```

Nuestro método borrar simplemente controla que el parámetro **x** sea diferente de null: Si **x** es null produce el mensaje de que el dato no existe y retorna, de lo contrario invoca el método desconectar.

```
1.  void  desconectar(nodoDoble  x)
2.            if  (primero == ultimo  and  primero == x)  then
3.                        primero = null
4.                        ultimo = null
5.                        return
6.            end(if)
7.            x.retornaLi().asignaLd(x.retornaLd())
8.            x.retornaLd().asignaLi(x.retornaLi())
9.            if  (primero == x)  then
10.                       primero = x.retornaLd()
11.                       return
12.           end(if)
13.           if  (ultimo == x)  then
14.                       ultimo = x.retornaLi()
15.           end(if)
16.  fin(desconectar)
```

Nuestro método desconectar controla que la lista tenga un solo registro (instrucción 2). En caso de ser cierta esta condición le asigna null a las variables primero y ultimo y retorna.

En caso de que la lista tenga más de un nodo efectuamos las instrucciones propias para desconectar un nodo intermedio en una lista doblemente ligada (numeral 8.3.1 del texto guía), las cuales son las instrucciones 7 y 8. Además, en las instrucciones 9 a 15, controlamos que el nodo desconectado haya sido el primero o el último con el fin de actualizar correctamente estas variables.

11. Elabore algoritmos para los diferentes métodos correspondientes al proceso de borrado en listas doblemente ligadas con nodo cabeza.

Solución:

```
1.  nodoDoble  buscarDato(objeto  d)
2.          nodoDoble  x
3.          x = primerNodo()
4.          while  (no finDeRecorrido(x)  and  x.retornaDato() != d)  do
5.                  x = x.retornaLd()
6.          end(while)
7.          return  x
8.  fin(buscarDato)
```

Nuestro método buscar dato es idéntico a los que hemos desarrollado anteriormente. La diferencia se presenta en que el fin de recorrido es verdadero cuando **x** sea null, ya que la lista no es circular. En otras palabras, cuando se sale del ciclo significa que no encontró el dato que estaba buscando o que lo encontró: si **x** sale valiendo null significa que no lo encontró, por lo tanto retorna null, pero si lo encontró retorna el nodo en el cual se halla el dato **d**.

```
1.  void  borrar(nodoDoble  x)
2.          if  (x == null)  then
3.                  write("dato no existe")
4.                  return
5.          end(if)
6.          desconectar(x)
7.  fin(borrar)
```

El método borrar es idéntico a los anteriores.

```
1.  void  desconectar(nodoDoble  x)
2.          x.retornaLi().asignaLd(x.retornaLd())
3.          if  (x == ultimo)  then
4.                  ultimo = x.retornaLi()
5.          else
6.                  x.retornaLd().asignaLi(x.retornaLi())
7.          end(if)
8.  fin(desconectar)
```

Nuestro método desconectar comienza modificando el campo liga derecha del nodo liga izquierda de **x**. Si **x** era el último nodo sólo hay que actualizar la variable ultimo, de lo contrario modifica el campo liga izquierda del nodo liga derecha de **x**.

Autoevaluación 9

La presente autoevaluación está desarrollada con base en los ejercicios 1, 2, 5 y 7 propuestos en el módulo 9 del texto guía.

1. Elabore un algoritmo para determinar si una hilera dada constituye un palíndromo o no. Un palíndromo es una hilera que se lee igual de izquierda a derecha que de derecha a izquierda. Por ejemplo: radar, reconocer, abba. Su algoritmo debe retornar verdadero si cumple la condición, falso de lo contrario.

   Solución:

```
1.  boolean  esPalindromo()
2.              entero  i, j
3.              i = 1
4.              j = longitud()
5.              while  (i < j  and  subHilera(i, 1) == subHilera(j, 1))  do
6.                          i = i + 1
7.                          j = j – 1
8.              end(while)
9.              if  (i < j)  then
10.                         return  false
11.             end(if)
12.             return  true
13.  fin(esPalindromo)
```

   A nuestro método lo hemos denominado esPalindromo y retorna un dato de tipo boolean, es decir, retorna verdadero si la hilera que invoca el método es palíndromo, falso de lo contrario.

   En la instrucción 2 definimos las variables locales a utilizar: **i** para recorrer la hilera que invoca el método, y **j** para recorrer la misma hilera de derecha a izquierda. Dichas variables las inicializamos en 1 y en el tamaño de la hilera usando las instrucciones 3 y 4, respectivamente.

   En las instrucciones 5 a 8 efectuamos el ciclo de control de la condición de palíndromo: comparamos el carácter de la posición **i** con el carácter de la posición **j** mientras la **i** sea menor que la **j**. Si ambas condiciones son verdaderas simplemente avanzamos con **i** y nos devolvemos con **j** (instrucciones 6 y 7). El control de que **i** sea menor que **j** sirve cubrir las situaciones de que el número de caracteres de la hilera que se está procesando sea par o impar, además de que proporciona eficiencia al algoritmo ya que sólo efectúa

n/2 comparaciones, siendo **n** el número de caracteres de la hilera que invocó el método.

Es importante notar que utilizamos las operaciones definidas para objetos de la clase hilera, en este caso las operaciones longitud y subHilera.

2. Elabore un algoritmo para determinar si una hilera es anagrama de otra o no (una hilera es anagrama de otra cuando tienen las mismas letras pero en diferente orden, por ejemplo: pecan y penca, salta y atlas, monja y jamón). Considere todas las situaciones: algoritmo independiente de la representación, hilera representada como vector, hilera representada como lista ligada. Su algoritmo debe retornar verdadero si cumple la condición, falso de lo contrario.

Solución:

```
1.  boolean  esAnagrama(hilera  b)
2.          entero  i, m, n, k
3.          hilera  aux, car
4.          m = longitud()
5.          n = b.longitud()
6.          if  (m != n)  then
7.                  return  false
8.          end(if)
9.          aux = b.subHilera(1, n)
10.         i = 1
11.         car = subHilera(i, 1)
12.         k = aux.posicion(car)
13.         while  (i <= m  and  k !=0)  do
14.                 aux.borre(k, 1)
15.                 i = i + 1
16.                 car = subHilera(i, 1)
17.                 k = aux.posicion(car)
18.         end(while)
19.         if  (i > m)  then
20.                 return  true
21.         end(if)
22.         return  false
23.  fin(esAnagrama)
```

A nuestro algoritmo lo llamamos esAnagrama, tiene un parámetro (que llamamos **b**, que es un objeto de la clase hilera) y retorna un dato de tipo boolean, el cual será verdadero si la hilera que invoca el método es anagrama de la hilera enviada como parámetro, falso de lo contrario.

En las instrucciones 2 y 3 definimos las variables de trabajo.

La primera condición que se debe cumplir para que dos hileras cumplan la condición de que sean anagrama es que ambas hileras tengan la misma longitud; por lo tanto, determinamos la longitud de la hilera que invocó el método (instrucción 4) y de la hilera enviada como parámetro (instrucción 5), y en la instrucción 6 comparamos dichas longitudes: si dichas longitudes son diferentes es suficiente para decir que no son anagrama; por consiguiente, procedemos a retornar falso en la instrucción 7.

Si las hileras tienen la misma longitud efectuamos una copia de la hilera enviada como parámetro (instrucción 9), ya que como nuestro algoritmo modificaría dicha hilera trabajaremos con la copia. Fíjese que la copia la efectuamos usando la función subHilera. Se copian todos los caracteres, es decir, a partir de la posición 1 se copian **n** caracteres. Dicha copia la denominamos **aux**.

Nuestro algoritmo procesa la hilera que invocó el método carácter por carácter (instrucciones 10 y 11 antes del ciclo e instrucciones 15 y 16 dentro del ciclo). Cada carácter se busca en **aux** usando la función posición. Si lo encuentra el valor de **k** será diferente de 0, y por lo tanto procedemos aborrar dicho carácter de la hilera **aux** (instrucción 14), avanzamos en la hilera que invocó el método, extraemos el siguiente carácter, volvemos a buscarlo en la hilera **aux** y repetimos el proceso hasta que se termine de recorrer la hilera que invocó el método o que no se encuentre el carácter en la hilera **aux**.

El ciclo termina cuando **k** sea 0 o cuando **i** sea mayor que **m**. Si termina porque la **i** es mayor que la **m** significa que todas las letras de la hilera que invocó el método están en la hilera **b**, y por lo tanto retorna verdadero. Si se salió del ciclo porque **k** es igual a 0, significa que alguna letra de la hilera que invocó el método no se halla en la hilera **b**, y por consiguiente retorna falso.

5. Elabore un algoritmo que invierta todos los caracteres de una hilera dada. Por ejemplo, si la hilera es “amor”, al ejecutar su algoritmo debe quedar la hilera “roma”.

Solución:

```
1.  void  invierte()
2.            entero  i, n
3.            hilera  c
4.            n = longitud()
5.            i = 2
6.            while  (i <= n)  do
7.                        c = subHilera(i, 1)
8.                        borre(i, 1)
9.                        inserte(c, 1)
10.                       i = i + 1
11.           end(while)
12.  fin(invierte)
```

A nuestro algoritmo lo denominamos invierte y es de tipo void ya que no retornará ningún dato sino que simplemente ejecutará una tarea sobre la hilera que invocó el método.

En las instrucciones 2 y 3 definimos nuestras variables de trabajo. Para efectuar el proceso de inversión de los caracteres ejecutamos el ciclo de las instrucciones 6 a 11. Nuestra técnica para hacer este proceso consiste en recorrer la hilera que invocó el método desde la posición 2 hasta la posición **n**. Cada carácter lo extraemos con la instrucción 7, lo borramos de la hilera con la instrucción 8 y lo insertamos al principio con la instrucción 9. El ciclo termina cuando la i sea mayor que la longitud de la lista **n**.

7. Elabore un algoritmo que determine si una palabra comienza con la letra "a" y termina con el sufijo "aco". Su algoritmo debe retornar verdadero si cumple la condición, falso de lo contrario.

Solución:

```
1.  boolean  condición()
2.          entero  n
3.          hilera  s = "aco"
4.          if  (subHilera(1, 1) != "a")  then
5.                  return  false
6.          end(if)
7.          n = longitud()
8.          if  (posicion(s) == n – 2)  then
9.                  return  true
10.         end(if)
11.         return  false
12.  fin(condición)
```

Nuestro método es bastante simple. En la instrucción 3 creamos una hilera llamada **s** con la secuencia "aco". En la instrucción 4 controlamos que el primer carácter de la hilera que invocó el método sea una "a", y de no serlo retornamos falso puesto que no cumple la primera condición dada. De no haber retornado aún, determinamos la longitud de la hilera que invocó el método y buscamos la hilera **s** en la hilera que invocó el método. Si dicha hilera se encuentra en la posición n – 2 significa que la hilera termina en "aco"; por consiguiente, retornamos verdadero. Si aún nuestro método no ha retornado quiere decir que la hilera que invocó el método no termina en "aco"; por lo tanto, retorna falso.

Autoevaluación 10

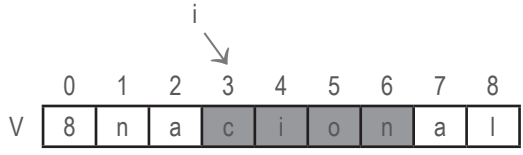
La presente autoevaluación está desarrollada con base en los ejercicios 1, 3, 5, 7 y 9 propuestos en el módulo 10 del texto guía.

1. Elabore un algoritmo para la operación subHilera considerando las hileras representadas como vectores.

Solución:

```
1.  hilera  subHilera(entero  i, entero  j)
2.          entero  n, k
3.          hilera  s
4.          n = longitud()
5.          if  (i <  1 or  i > n)  then
6.                  write("parámetro i inválido")
7.                  return  null
8.          end(if)
9.          if  (j < 0  or  i + j > n)  then
10.                 write("parámetro j inválido")
11.                 return  null
12.         end(if)
13.         s = new  hilera(j)
14.         for (k = i; k < i + j; k++)  do
15.                 s.asignaDato(V[k], k – i + 1)
16.         end(for)
17.         s.asignaDato(j, 0)
18.         return  s
19.  fin(subHilera)
```

Nuestro método retorna un objeto de la clase hilera y tiene dos parámetros: **i** y **j**. El parámetro **i** indica la posición a partir de la cual se debe hacer la copia y el parámetro **j** indica cuántos caracteres se deben copiar. Si tenemos la hilera "nacional", y los parámetros **i** y **j** valen 3 y 4, respectivamente, estamos en una situación como ésta:



en la cual a partir del carácter de la posición 3 hay que copiar cuatro caracteres (los sombreados).

En la instrucción 2 definimos las variables locales de trabajo y en la instrucción 3 la variable en la cual se retornará la copia. En la instrucción 4 calculamos la longitud de la hilera que invocó el método con el fin de controlar que los parámetros **i** y **j** sean válidos. Dicha longitud la guardamos en la variable **n**. En las instrucciones 5 a 8 se controla que el parámetro **i** sea válido: el parámetro **i** es válido si es mayor que 0 y menor o igual que **n**. En las instrucciones 9 a 12 se controla que el parámetro **j** sea válido: el parámetro **j** es válido si es mayor o igual que 0 y si al sumárselo a **i** no es mayor que la longitud de la hilera menos 1. Si tenemos una hilera con longitud 8 el parámetro **i** podrá estar entre 1 y 8. Si la hilera tiene longitud 8 y el valor de la **i** es 5, la **j** podrá tener valores entre 0 y 4.

En la instrucción 13 se crea la nueva hilera vacía. Y en el ciclo desde la instrucción 14 hasta la 16 se construye la nueva hilera en el objeto **s**, el cual es derivado de la clase vector, ya que nuestro ejercicio tiene la condición de que las hileras se representan en vector. El método asignaDato es un método de la clase vector en el cual el primer parámetro es el dato a guardar y el segundo es la posición en la cual se debe almacenar dicho dato. Es decir, en la instrucción 15 se está instruyendo a la máquina para que almacene el dato de la posición **k** del vector V en la posición **k – i** del vector correspondiente a la hilera que se está construyendo.

En la instrucción 18 se retorna la hilera construida, la cual queda así:

	0	1	2	3	4
V	4	c	i	o	n

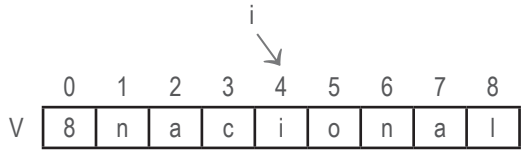
3. Elabore un algoritmo para la operación inserte considerando las hileras representadas como vectores.

Solución:

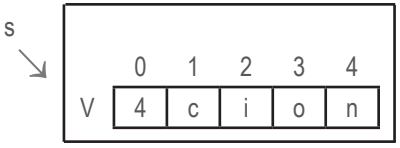
```
1.  void  inserte(hilera  s, entero  i)
2.            entero  k, n, m
3.            n = longitud()
4.            if  (i < 1  or  i > n)  then
5.                        write("parámetro i inválido")
6.                        return
7.            end(if)
8.            m = s.longitud()
9.            for (k = n; k >= i; k--)  do
10.                       V[m+k] = V[k]
11.           end(for)
12.           for (k = 0; k <= m; k++)  do
```

```
13.                     V[k + i] = s.datoEnPosicion(k + 1)
14.             end(for)
15.             V[0] = n + m
16.   fin(inserte)
```

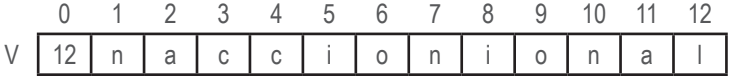

A nuestro método lo llamamos inserte, es tipo void puesto que modificará el vector correspondiente a la hilera que invocó el método y no retornará ningún valor. Los parámetros son un objeto de la clase hilera, que llamamos **s**, y un entero llamado **i** que indica la posición a partir de la cual hay que insertar los caracteres de la hilera **s** en el vector de la hilera que invocó el método. Consideremos que la hilera que invocó el método es:



y que la hilera **s** pasada como parámetro es:



y que el valor de **i** es 4; al ejecutar nuestro método inserte la hilera que invocó el método queda así:



En la instrucción 2 definimos las variables locales de trabajo. En la instrucción 3 determinamos el número de caracteres de la hilera que invocó el método con el fin de controlar que el parámetro **i** sea válido. En las instrucciones 4 a 7 controlamos la validez del parámetro **i**. Para que el parámetro **i** sea válido debe estar ente 1 y **n**. Si no lo está producimos el mensaje de que dicho parámetro es inválido y se retorna al programa llamante.

En caso de que el parámetro **i** sea válido determinamos la longitud de la hilera a insertar (instrucción 8), y en las instrucciones 9 a 11 se mueven hacia la derecha los datos que hay a partir de la posición **i** de la hilera que invocó el método. Estos datos se mueven **m** posiciones hacia la derecha, ya que **m**, que es el tamaño de la hilera a insertar, es el total de caracteres que hay que insertar.

En las instrucciones 12 a 14 se llevan los datos de la hilera **s** hacia las posiciones correspondientes en el vector V de la hilera que invocó el método. En la instrucción 13 se usa el método de la clase vector datoEnPosicion(k), que es el método con el cual se accede

el dato que está en la posición **k** de la hilera pasada como parámetro. Y por último, en la instrucción 15 actualizamos la longitud de la hilera.

5. Elabore un algoritmo para la operación replace considerando las hileras representadas como vectores.

Solución:

Recordemos lo que hace la operación replace. La operación replace tiene la forma replace(i, j, s), donde los parámetros **i** y **j** son datos enteros que significan que en la hilera que invocó el método, a partir del carácter que se halla en la posición **i** debe reemplazar **j** caracteres por la hilera **s**. Esto significa que a partir del carácter que se halla en la posición **i** se deben borrar **j** caracteres y luego, a partir de la misma posición **i**, se debe insertar la hilera **s**. Por consiguiente, nuestro algoritmo para la operación replace será simplemente invocar la operación de borrado y luego la operación de inserción. Nuestro algoritmo será:

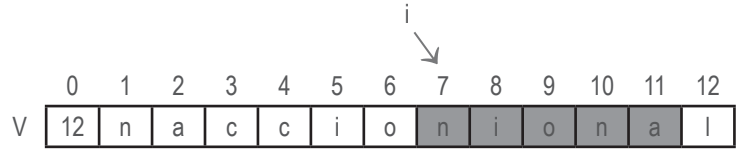
```
1.  void  replace(entero  i, entero  j, hilera  s)
2.            borre(i, j)
3.            inserte(i, s)
4.  fin(replace)
```

Para una mayor explicación presentamos el algoritmo correspondiente al método borre(i, j) teniendo las hileras representadas en vector:

```
1.  void  borre(entero  i, entero  j)
2.            entero  n, k
3.            n = longitud()
4.            if  (i < 1  or  i > n)  then
5.                        write("parámetro i inválido")
6.                        return  null
7.            end(if)
8.            if  (j < 0  or  i + j > n)  then
9.                        write("parámetro j inválido")
10.                       return
11.           end(if)
12.           for (k = i + j; k <= n; k++)  do
13.                       V[k – j] = V[k]
14.           end(for)
15.           V[0] = n – j
16. fin(borre)
```

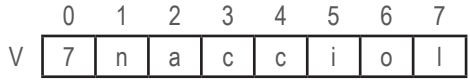
Nuestro método borre tiene dos parámetros: **i** que indica a partir de cuál posición se deben comenzar a eliminar caracteres, y **j**, que indica cuántos caracteres hay que eliminar. Si tenemos la hilera que se presenta a continuación y los valores de **i** y **j** son 7 y 5,

respectivamente, estamos en una situación como ésta:



en la cual, a partir de la posición 7 del vector, hay que eliminar cinco caracteres.

Al efectuar el proceso borre(7, 5) la hilera queda así:



En la instrucción 2 definimos las variables locales de trabajo. En la instrucción 3 determinamos la longitud de la hilera que invocó el método con el fin de controlar que los parámetros **i** y **j** sean válidos.

En las instrucciones 4 a 7 se controla que el parámetro **i** sea válido. El parámetro **i** es válido si se halla entre 1 y **n**. Si el parámetro **i** no es válido producimos el mensaje apropiado y retornamos al programa llamante.

En las instrucciones 8 a 11 se controla que el parámetro **j** sea válido. El parámetro j es válido si es mayor o igual que 0 y si al sumárselo a **i** no es mayor que la longitud de la hilera. Si tenemos una hilera con longitud 8 el parámetro **i** podrá estar entre 1 y 8. Si la hilera tiene longitud 8 y el valor de la **i** es 5, la **j** podrá tener valores entre 0 y 4.

En las instrucciones 12 a 14 desplazamos hacia la izquierda los datos que hay a partir de la posición **i + j** hasta la posición **n**, **j** caracteres. Si se tiene una hilera con trece caracteres, la **i** vale 5 y la **j** vale 6, con este ciclo desplazamos seis posiciones hacia la izquierda los caracteres que hay desde la posición 11 hasta la posición 13.

Por último, actualizamos la longitud de la hilera, la cual manejamos en la posición 0 del vector.

7. Elabore un algoritmo para la operación borre considerando las hileras representadas como listas simplemente ligadas.

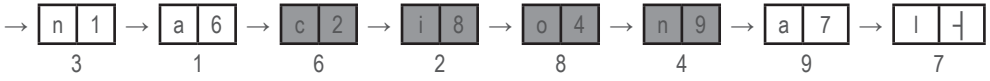
Solución:

```
1.  void  borre(entero  i, entero  j)
2.            entero  k, n
3.            n = longitud()
```

```
4.          if  (i < 0  or  i > n)  then
5.                      write("parámetro i inválido")
6.                      return
7.          end(if)
8.          if  (j < 0  or  i + j > n)  then
9.                      write("parámetro j inválido")
10.                     return
11.         end(if)
12.         nodoSimple  p, q
13.         p = primerNodo()
14.         q = anterior(p)
15.         k = 1
16.         while  (k < i)  do
17.                     q = p
18.                     p = p.retornaLiga()
19.         end(while)
20.         k = 0
21.         while  (k < j)  do
22.                     desconectar(p, q)
23.                     p = q.retornaLiga()
24.         end(while)
25.  fin(borre)
```

Nuestro algoritmo borre considerando las hileras representadas en listas ligadas es idéntico al de la representación en vector en lo concerniente al control de la validez de los parámetros **i** y **j**.

A partir de la instrucción 12 las instrucciones son muy diferentes. Considerando la hilera "nacional" representada como lista simplemente ligada y que tenemos la instrucción borre(3, 4), la situación es la siguiente:



en la cual los nodos a eliminar son los sombreados.

En la instrucción 12 definimos las variables de trabajo **p** y **q**, con las cuales recorreremos la lista ligada para ubicarnos en el nodo que ocupa la posición **i** en la lista ligada. Para ello está el ciclo de las instrucciones 16 a 19. Usamos un contador, llamado **k**, el cual inicializamos en 1. Cuando **k** sea igual a **i** significa que la variable **p** está ubicada en el *i*-ésimo nodo de la lista ligada. Luego restauramos dicho contador en 0 y con el ciclo de las instrucciones 21 a 24 eliminamos **j** nodos utilizando el método desconectar que habíamos desarrollado para objetos de la clase lista simplemente ligada. Recuerde que cuando representemos la clase hilera como lista ligada, se define derivada de la clase LSL.

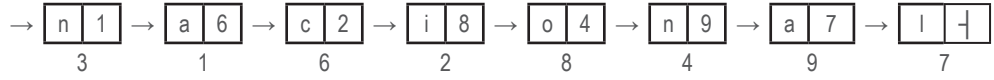
9. Elabore un algoritmo para la operación posición considerando las hileras representadas como listas simplemente ligadas.

Solución:

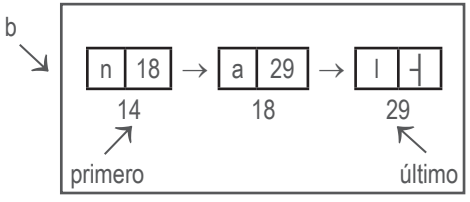
```
1.  nodoSimple posición(hilera b)
2.          nodoSimple p, pp, q
3.          p = primerNodo()
4.          pp = p
5.          q = b.primerNodo()
6.          while (no finDeRecorrido(p) and no b.finDeRecorrido(q)) do
7.                  if (p.retornaDato() == q.retornaDato()) then
8.                          p = p.retornaLiga()
9.                          q = q.retornaLiga()
10.                         if (b.finDeRecorrido(q)) then
11.                                 return pp
12.                         end(if)
13.                 else
14.                         pp = pp.retornaLiga()
15.                         p = pp
16.                         q = b.primerNodo()
17.                 end(if)
18.         end(while)
19.         return null
20. fin(posicion)
```

Nuestro método lo llamamos posición y retorna un objeto de la clase nodoSimple. Tiene un parámetro que es un objeto de la clase hilera, el cual llamamos **b**. Recuerde que nuestro método lo que hará será buscar la hilera pasada como parámetro (la hilera **b**) en la hilera que invocó el método. Si la encuentra retornará el nodo a partir del cual encontró la hilera **b** en la hilera que invocó el método; de lo contrario, retornará null.

Supongamos que la hilera que invocó el método es la siguiente:



y que la hilera enviada como parámetro es:



Al ejecutar nuestro algoritmo posición, éste retornará 4, ya que a partir del nodo 4 de la hilera que invocó el método se encuentra la hilera “nal”.

En la instrucción 2 definimos las variables locales de trabajo: las variables **p** y **q** se utilizan para recorrer la lista que invocó el método y la lista pasada como parámetro, respectivamente, e ir comparando el dato de **p** con el dato de **q**; la variable **pp** sirve para indicar a partir de cúal nodo de la hilera que invocó el método se comenzaron a hacer comparaciones.

Inicialmente las variables **p** y **pp** apuntan hacia el primer nodo de la hilera que invocó el método (instrucciones 3 y 4) y la hilera **q** apunta hacia el primer nodo de la hilera **b** (la hilera pasada como parámetro) mediante la instrucción 5.

En el ciclo de las instrucciones 6 a 18 efectuamos el proceso. Mientras **p** y **q** sean diferentes de null (finDeRecorrido()), se compara el dato de **p** con el dato de **q**. Si son iguales avanzamos con **p** y **q** y controlamos que se haya terminado de recorrer la lista **b**: si **q** es null significa que se encontró la hilera **b** en la hilera que invocó el método; por lo tanto, se retorna el valor de **pp**. Si el dato de **p** es diferente del dato de **q** avanzamos con **pp**, asignamos a **p** el valor de **pp** y restauramos **q** al principio de la lista **b**.

Si termina el ciclo y no retornó significa que la hilera **b** no se halla en la hilera que invocó el método; por lo tanto, retorna null.

Autoevaluación 11

La presente autoevaluación está desarrollada con base en los ejercicios 1, 3, 5 y 8 propuestos en el módulo 11 del texto guía.

1. Elabore un algoritmo colaLlena para el algoritmo encolar desarrollado en el numeral 11.3 del texto guía. Recuerde que este algoritmo debe producir el mensaje apropiado y restaurar la variable ultimo en el valor que tenía antes de ser modificada. Trate de no usar instrucciones de decisión.

   Solución:

   ```
   1. void colaLlena()
   2.         write("Cola llena")
   3.         ultimo = (ultimo – 1 + n) % n
   4. fin(colaLlena)
   ```

   Nuestro algoritmo es bastante sencillo. Simplemente produce el mensaje de que la cola está llena y restaura la variable ultimo en el valor que tenía antes de haberla incrementado en 1. Recuerde que la variable ultimo se incrementó en 1 utilizando el operador módulo:

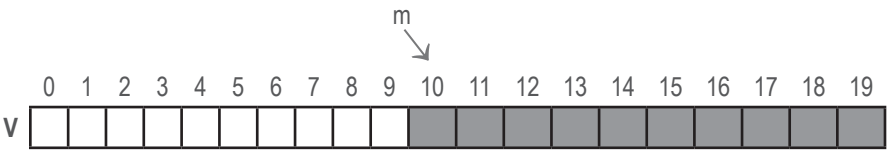
   ```
   ultimo = (ultimo + 1) % n
   ```

   Para decrementar ultimo en 1 sería un error codificar la instrucción ultimo = (ultimo – 1) % n, ya que cuando ultimo hubiera quedado en 0, al restarle 1 daría un número negativo y el resultado sería erróneo. Para obviar esta dificultad se le suma **n** al resultado de ultimo – 1 antes de aplicar el operador módulo, y ésa es la instrucción 3 de nuestro algoritmo colaLlena.

3. Elabore un algoritmo que maneje una pila y una cola en un vector. La cola se debe manejar circularmente. Elabore algoritmos para apilar o encolar y desapilar o desencolar un dato en la estructura definida por usted.

   Solución:

   La forma como efectuamos este diseño es definiendo un vector desde 0 hasta **n**, o sea que los subíndices del vector variarán desde 0 hasta **n – 1**. Dicho vector lo dividimos por la mitad. Llamemos **m = n/2**. La primera mitad, desde la posición 0 hasta la posición **m – 1**, le corresponderá a la cola, y la segunda mitad, desde **m** hasta **n**, le corresponderá a la pila. Gráficamente, para un valor 20 de **n** tenemos:



Teniendo este diseño, el manejo circular de la cola es el mismo que si tuviéramos una cola representada en un vector; por lo tanto, el manejo circular de ella es idéntico al visto en el numeral 11.3 del texto guía, usando la variable **m** en vez de la variable **n**. La pila se representa desde **m** hasta **n – 1**. La pila estará vacía cuando la variable tope valga **m – 1**, y estará llena cuando la variable tope valga **n – 1**. Teniendo definido este diseño, los algoritmos para apilar o encolar y para desencolar o desapilar se presentan a continuación.

```
1.  void  apilarEncolar(objeto  d, entero  sw)
2.            if  (sw == 1)  then
3.                        if  (tope == n – 1)  then
4.                                    unaLlena(sw)
5.                        end(if)
6.                        tope = tope + 1
7.                        V[tope] = d
8.            else
9.                        ultimo = (ultimo + 1) % m
10.                       if  (ultimo == primero)  then
11.                                   unaLlena(sw)
12.                       end(if)
13.                       V[ultimo] = d
14.           end(if)
15.  fin(apilarEncolar)
```

Nuestro algoritmo para apilar o encolar es de tipo void y tiene dos parámetros: uno el dato a encolar o apilar y el otro un indicador que define cuál operación realizar, si apilar o encolar. Cuando el parámetro **sw** sea igual a 1 significa que el dato pasado como parámetro hay que guardarlo en la pila, de lo contrario significa que hay que guardarlo en la cola. En las instrucciones 3 a 7 se efectúa la operación de apilar, y en las instrucciones 9 a 13 la de encolar.

```
1.  objeto  desapilarDesencolar(entero  sw)
2.            if  (sw == 1)  then
3.                        if  (tope == m – 1)  then
4.                                    write(“Pila  vacía”)
5.                                    return  null
6.                        end(if)
7.                        d = V[tope]
8.                        tope = tope – 1
9.                        return  d
```

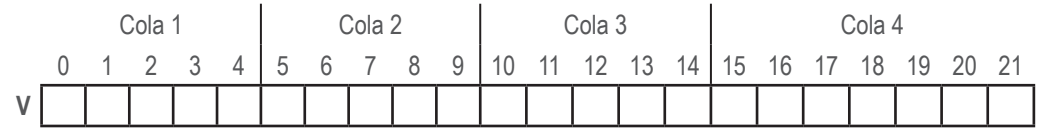
```
10.         else
11.                 if  (primero == ultimo)  then
12.                         write("Cola vacía")
13.                         return  null
14.                 end(if)
15.                 primero = (primero + 1) % m
16.                 return  V[primero]
17.         end(if)
18.  fin(desapilarDesencolar)
```

Nuestro método para desencolar retorna un objeto, el que se desencola o el que se desapila. Tiene un parámetro, el cual es el **sw**, que indica si la operación a realizar es desencolar o desapilar. Si el parámetro **sw** es igual a 1 quiere decir que la operación a efectuar es desapilar, de lo contrario significa que la operación a efectuar es desencolar. Las instrucciones 3 a 9 son las que efectúan la operación de desapile, mientras que las instrucciones 11 a 16 son las que desencolan.

5. Diseñe la forma de manejar **n** colas en un vector de **m** elementos. Cada cola se debe manejar circularmente. Elabore algoritmos para encolar o desencolar un dato de alguna cola.

Solución:

Para representar **n** colas en un vector de **m** elementos, inicialmente definimos la porción de vector que le corresponde a cada cola. Inicialmente la porción que le corresponde a cada cola es **n/m**. Como dicha operación no siempre arroja un resultado exacto significa que la última cola (la cola **n**) tendrá más posiciones que las otras. Eso no es problema. Si consideremos que **m** es 22 y que **n** es 4, la configuración inicial es:



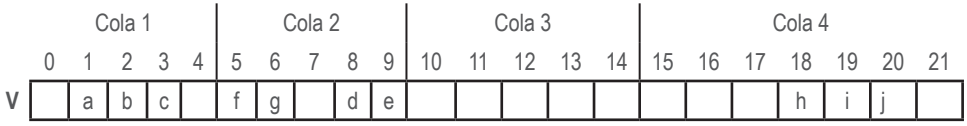
Para identificar cuál es la porción que le corresponde a cada cola manejaremos un vector que llamaremos bases. Bases[i] apuntará hacia la posición en la cual comienza la cola **i**. Para nuestro ejemplo, el vector de bases es:

	1	2	3	4	5
bases	0	5	10	15	22

donde el tamaño de la cola **i** se obtiene efectuando la resta entre bases[i + 1] y bases[i]. Esa es la razón por la cual si tenemos **n** colas entonces el vector de bases tendrá **n + 1** elementos donde el valor de la posición **n + 1** es **m** (el tamaño del vector).

Adicionalmente manejaremos dos vectores: uno que llamaremos primeros, que indicará en cuál posición del vector se halla el dato de cada cola, y otro que llamaremos ultimos, que indicará en cuál posición del vector se halla el último dato de cada cola.

Si tuviéramos el vector de la siguiente forma:



el contenido de los vectores primeros y ultimos es:

Recuerde que cada cola se maneja circularmente. El hecho de que primeros[1] valga 0 significa que el primer dato de la cola 1 se halla en la posición 1 del vector **V**; el hecho de que primeros[2] valga 7 significa que el primer dato de la cola 2 se halla en la posición 8 del vector, y así sucesivamente.

Ahora, las condiciones de cola vacía y de cola llena son las mismas que cuando se maneja una sola cola. Es decir, la cola **i** está vacía o llena si primeros[i] es igual a últimos[i]. Recuerde que esta condición se controla en encolar después de incrementar ultimos[i] en 1. En nuestro ejemplo, la cola 3 está vacía y la cola 2 está llena.

Los algoritmos para encolar y desencolar de una cola i cualquiera se presentan a continuación:

```
1.  void  encolar(objeto  d,  entero  i)
2.            ultimos[i] = (ultimos[i] + 1 – bases[i] ) % (bases[i + 1] – bases[i]) + bases[i]
3.            if  (ultimos[i] == primero[i])  then
4.                        colaLlena(i)
5.            end(if)
6.            V[ultimos[i]] = d
7.  finEncolar)
```

Nuestro método para encolar es de tipo void y tiene dos parámetros: uno el dato a encolar y otro la pila en la cual se debe encolar el dato enviado.

Para encolar, lo primero que se hace es incrementar ultimos[i] en 1. Dicho incremento lo efectuamos usando el operador módulo (recuerde que cada cola debe ser circular). Con el fin de manejar cada cola circularmente debemos restar bases[i] a ultimos[i] + 1, con el fin de que temporalmente los subíndices de esa cola estén entre 0 y el tamaño

de la cola menos 1. Recuerde que el tamaño de cada cola se obtiene con bases[i + 1] – bases[i]. Luego de efectuada esta operación restauramos ultimos[i] en la posición que le corresponde sumándole bases[i], que era lo que se le había restado. En la instrucción 2 efectuamos esa tarea. Luego controlamos que la cola no esté llena (instrucción 3). En caso de estarlo se invoca el subprograma colaLlena(i) indicándole que la cola **i** está llena para que dicho programa tome las acciones del caso.


```
1.  objeto  desencolar(entero  i)
2.          if  (primeros[i] == ultimos[i])  then
3.                  write("Cola ", i, "vacía")
4.                  return  null
5.          end(if)
6.          primeros[i] = (primeros[i] + 1 – bases[i] ) % (bases[i + 1] – bases[i]) +
            bases[i]
7.          return  V[primeros[i]]
8.  fin(desencolar)
```

Nuestro método para desencolar retorna un dato de la clase objeto y tiene como parámetro un entero que indica de cuál cola se debe desencolar.

Lo primero que se hace es controlar que la cola no esté vacía. En caso de estarlo se produce el mensaje apropiado y retorna null.

Si la cola no está vacía simplemente se le agrega 1 a primeros[i] considerando que cada cola se debe manejar circularmente (instrucción 6) y se retorna el dato de la posición primeros[i].

8. Se tiene una cola representada circularmente en un vector. Elabore un algoritmo que imprima los datos en la cola desde el último hacia el primero.

Solución:

```
1.  void  recorreHaciaAtras()
2.          entero  p
3.          p = ultimo
4.          while  (p !=  primero)  do
5.                  write(V[p])
6.                  p = (p + n – 1) % n
7.          end(while)
8.  fin(recorreHaciaAtras)
```

Nuestro algoritmo es tipo void y no tiene parámetros. El único problema que se podría presentar es cuando **p** sea igual a 0. Recuerde que la cola es circular. Para obviar este problema nos devolvemos con la variable **p** usando el operador módulo y sumándole **n** (el tamaño del vector) de tal manera que dé la vuelta automáticamente cuando **p** sea 0 (vea el ejercicio 1 de esta misma guía).

Autoevaluación 12

La presente autoevaluación está desarrollada con base en los ejercicios 1, 2, 3 y 4 propuestos en el módulo 12 del texto guía.

1. Elabore un algoritmo pilaLlena para la representación descrita en el módulo 11, numeral 11.5 del texto guía, de tal manera que la búsqueda de la pila que tenga espacio sea así: si la pila llena es la pila **i**, primero busca en la pila **i + 1**, si está llena la pila **i + 1** busca en la pila **i – 1**, si están llenas esas dos pilas entonces busca en la pila **i + 2**, si ésta está llena entonces busca en la pila **i – 2**, y así sucesivamente. Cuando encuentre espacio en una pila deberá hacer los movimientos correspondientes para abrir espacio en la pila **i** y poder apilar en dicha pila.

   Solución:

```
1.  void  pilaLlena(entero  i)
2.            entero j, k, sw
3.            k = 1
4.            sw = 0
5.            while  ((i – k  >  0)  and  (i + k  <=  n))  do
6.                        j = i + k
7.                        if  (topes[j] == bases[j + 1])  then
8.                                    j = i – k
9.                                    if  (topes[j] == bases[j + 1])  then
10.                                             k = k + 1
11.                                   else
12.                                            sw = 1
13.                                            break
14.                                   end(if)
15.                       else
16.                                   sw = 1
17.                                   break
18.                       end(if)
19.           end(while)
20.           while  ((i – k  >  0)  and  (sw == 0))  do
21.                       j = i – k
22.                       if  (topes[j] == bases[j + 1])  then
23.                                   k = k + 1
24.                       else
25.                                   sw = 1
26.                                   break
```

```
27.                     end(if)
28.             end(while)
29.             while  ((i + k  <=  n)  and  (sw == 0))do
30.                     j = i + k
31.                     if  (topes[j] == bases[j + 1])  then
32.                             k = k + 1
33.                     else
34.                             sw = 1
35.                             break
36.                     end(if)
37.             end(while)
38.             if  (sw == 0)  then
39.                     write("todas las pilas están llenas, no se puede apilar)
40.                     STOP
41.             end(if)
42.             if  ((i + k  <=  n)  and  (topes[j]  !=  bases[j + 1]))  then
43.                     k = topes[j]
44.                     while   (k > topes[i])  do
45.                             V[k + 1] = V[k]
46.                                k = k – 1
47.                     end(while)
48.                     for   (k = i + 1; k <= j; k++)  do
49.                             topes[k] = topes[k] + 1
50.                             bases[k] = bases[k] + 1
51.                     end(for)
52.                     return
53.             end(if)
54.             for   (k = bases[j+1];  k < topes[i];  k++)  do
55.                     V[k] = V[k + 1]
56.             end(for)
57.             for   (k = j + 1;  k <= i;  k++)  do
58.                     topes[k] = topes[k] – 1
59.                     bases[k] = bases[k] – 1
60.            end(for)
61.   fin(pilaLlena)
```

A nuestro método lo llamamos pilaLlena, es de tipo void ya que sólo ejecutará una tarea y tiene como parámetro un entero i que indica cuál es la pila que está llena.

En la instrucción 2 definimos las variables locales de trabajo. En la instrucción 3 inicializamos la variable **k** en 1 y en el ciclo de las instrucciones 5 a 19 se efectúa el proceso de búsqueda solicitado. Cuando **k** vale 1 se investiga si en la pila **i + k** hay espacio (instrucción 7), y de ser así se sale del ciclo (instrucción 17). En caso de que no haya espacio en la pila **i + k** se investiga si hay espacio en la pila **i – k** (instrucción 9), y de ser así se sale del ciclo (instrucción 13); si no encontró espacio en las pilas **i + 1** e **i – 1** se incrementa

k en 1, para que en la próxima ejecución del ciclo investigue si hay espacio en la pila **i + 2** o en la pila **i – 2**. Este ciclo se ejecuta hasta que encuentre espacio en alguna pila (se sale con break en las instrucciones 13 o 17) o hasta que se agoten las pilas a la izquierda de la pila i (condición **i – k > 0** de la instrucción 5) o las pilas a la derecha de la pila **i** (condición **i + k <= n** en la misma instrucción 5). Fíjese que si se sale del ciclo porque encontró espacio en alguna pila la variable **sw** queda en 1, indicando que ya encontró espacio, con el fin de que los dos siguientes ciclos no se efectúen.

Si se sale del ciclo porque se agotaron las pilas a la derecha de la pila **i** se continúa buscando espacio en las pilas a la izquierda de la pila **i** con el ciclo de las instrucciones 20 a 28. Si se sale del ciclo porque se agotaron las pilas a la izquierda de la pila i entonces continúa buscando en las pilas a la derecha de la pila **i** (instrucciones 29 a 37).

En las instrucciones 38 a 41 se controla que sí se haya encontrado espacio en alguna pila. Si la variable **sw** está en 0 significa que todas las pilas estaban llenas; por consiguiente, no se podrá apilar en la pila **i**: se produce el mensaje apropiado y se cancela el proceso. Fíjese que la variable **sw** se pone en 1 cuando se encuentre espacio en alguna pila. La pila en la cual se encuentra espacio la estamos llamando **j**.

En las instrucciones 42 a 53 se procesa el caso en el que se haya encontrado espacio en una pila a la derecha de la pila **i**. En este caso hay que mover los datos del vector (desde la posición topes[j] hasta la posición topes[i] + 1) una posición hacia la derecha (instrucciones 44 a 47) y actualizar los vectores de topes y bases desde la pila **i + 1** hasta la pila j (instrucciones 48 a 51). Al terminar de hacer estas actualizaciones retorna al programa llamante para efectuar la operación de apilar (instrucción 52).

En las instrucciones 54 a 60 se procesa la situación en la cual se encontró espacio en una pila a la izquierda de la pila **i**. Por lo tanto, hay que desplazar una posición hacia la derecha todos los caracteres desde el primero de la pila **j + 1** hasta el último de la pila **i** (ciclo de las instrucciones 54 a 56) y actualizar los vectores bases y topes desde la pila **j + 1** hasta la pila **i** (instrucciones 57 a 60). Al terminar de ejecutar estas acciones retorna al programa apilar para guardar el dato en la pila **i**.

2. Supongamos que TEST es una función boolean que recibe un entero y retorna un valor igual o distinto a 0. Consideremos el siguiente segmento de código:

```
n = 3
p = new  pila()
for  (i = 1; i <= n; i++)  do
      if  (TEST(i))  then
            write(i)
      else
            p.apilar(i)
      end(if)
end(for)
```

```
while (no p.esVacia()) do
      i = p.desapilar()
      write(i)
end(while)
```


¿Cuáles de las siguientes son posibles salidas del código anterior?

a. 1 2 3
b. 1 3 2
c. 2 1 3
d. 3 1 2
e. 2 3 1
f. 3 2 1

Solución:

La solución es que salidas posibles son **a**, **b**, **e** y **f** por:

1. La salida 1 2 3 se produce cuando TEST devuelve valor distinto de 0 en los tres casos.
2. La salida 1 3 2 se produce cuando TEST devuelve 0 sólo para el 2.
3. La salida 2 3 1 se produce cuando TEST devuelve 0 sólo para el 1.
4. La salida 3 2 1 se produce cuando TEST devuelve 0 en los tres casos.

En general, serán salidas correctas aquellas que impriman una secuencia de N números de forma que se pueda dividir en dos partes:

1. Los M (0 <= M <= N) primeros $\mathbf{m_i}$ números son tales que $\mathbf{m_i < m_{i+1}}$ para todo **i** tal que **i** sea mayor o igual que 1 y menor que M, es decir, los no almacenados en la pila. Dicho de otra forma, los números **i** tal que TEST(i) es diferente de 0.

2. Los N – M últimos números que consisten en todos los números entre 1 y N que no hayan sido listados en la primera parte y además listados de forma decreciente (los almacenados en la pila, es decir, los números **i** tal que TEST(i) es 0).

3. Utilice una pila para resolver un problema clásico de emparejamiento de símbolos tales como {}, [], (). Su programa debe leer una hilera de símbolos e indicar si los caracteres anteriores están bien o mal emparejados.

Solución:

```
1. boolean controlaEmparejamientos(hilera s)
2.             entero n, i, b
3.             hilera car, aux
4.             stack pila = new stack()
```

```
5.          n = s.longitud()
6.          b = 0
7.          for (i = 1; i < n; i++)  do
8.                      car = s.subHilera(i, 1)
9.                      casos  de  car
10.                                 “(“, “[“, “{“:
11.                                             pila.apilar(car)
12.                                  “)”:
13.                                             aux = pila.desapilar()
14.                                             if  (aux  !=  “(“)  then
15.                                                         b = 1
16.                                                         break
17.                                             end(if)
18.                                  “]”:
19.                                             aux = pila.desapilar()
20.                                             if  (aux  !=  “[“)  then
21.                                                         b = 1
22.                                                         break
23.                                             end(if)
24.                                  “}”:
25.                                             aux = pila.desapilar()
26.                                             if (aux  !=  “{“)  then
27.                                                         b = 1
28.                                                         break
29.                                              end(if)
30.                     fin(casos)
31.         end(for)
32.         if  (b == 0)  then
33.                     return  true
34.         else
35.                     return  false
36.         end(if)
37.  fin(controlaEmparejamientos)
```

A nuestro método lo denominamos controlaEmparejamientos, retorna un dato de tipo boolean y tiene como parámetro una hilera **s** que es la que contiene la secuencia de símbolos (, {, [,], },) a los cuales se les debe controlar la condición de que estén correctamente emparejados, es decir, que a cada abre símbolo le corresponda un cierre símbolo del mismo tipo y que no haya traslapos entre ellos. Un traslapo sería: “([ )]”.

En las instrucciones 2 a 4 definimos las variables locales de trabajo: **n** para guardar la longitud de la hilera **s**, **i** para recorrer la hilera e ir procesando cada uno de los caracteres, **b** para controlar cuando hay algún error, **car** para hacer una copia de cada carácter de la hilera **s** y procesarlo, **aux**, para desafilar un dato de la pila y comprobar la correctitud de él, y **pila** (variable tipo stack) para guardar los símbolos de apertura de cada una de las parejas a controlar.

En la instrucción 5 se calcula la longitud de la hilera a procesar. En la instrucción 6 se inicializa la variable **b** en 0. Si durante el proceso la variable se convierte en 1 significa que se encontró un error. Si termina el ciclo y la variable **b** se conservó en 0 significa que la hilera procesada cumple las condiciones de emparejamiento.

En el ciclo de las instrucciones 7 a 31 se procesa la hilera **s**. Para cada valor de **i** se extrae el carácter de esa posición y se evalúa cuál símbolo es. Si es un símbolo de apertura (abre paréntesis, abre corchete, abre llave) simplemente se almacena en la pila (instrucciones 10 y 11).

Si el carácter es un símbolo de cierre (cierre paréntesis, cierre corchete, cierre llave) se controla cada uno de ellos por separado.

En las instrucciones 12 a 17 se controla cuándo el carácter es un cierre paréntesis. Se procede a desapilar un símbolo de la pila y comprobar que sea un abre paréntesis; de no ser así significa que hay un cierre paréntesis que no tiene su correspondiente abre paréntesis o que lo tiene mal ubicado.

En las instrucciones 18 a 23 se controla cuándo el carácter es un cierre corchete. Se procede a desapilar un símbolo de la pila y comprobar que sea un abre corchete; de no ser así significa que hay un cierre corchete que no tiene su correspondiente abre corchete o que lo tiene mal ubicado.

En las instrucciones 24 a 29 se controla cuándo el carácter es un cierre llave. Se procede a desapilar un símbolo de la pila y comprobar que sea un abre llave; de no ser así significa que hay un cierre llave que no tiene su correspondiente abre llave o que la tiene mal ubicada.

Al terminar el ciclo se controla el valor de la variable **b** para determinar si se retorna verdadero o falso (instrucciones 32 a 36).

4. Se tiene una matriz de **m** filas y **n** columnas. En cada posición hay un 0 o un 1: 0 significa que se puede avanzar hacia esa casilla, y 1 que esa casilla está bloqueada. Teniendo definida una casilla de entrada y una casilla de salida elabore un algoritmo que determine e imprima el camino a seguir para ir desde la entrada hacia la salida.

Solución:

Definiremos un arreglo de dos dimensiones llamado laberinto, en el cual cada casilla tendrá un 0 o un 1 indicando si se puede avanzar hacia esa casilla o no. Adicionalmente trabajaremos una matriz que llamaremos marca, con la cual se controla cuáles casillas se han utilizado ya buscando la salida: marca[i][j] es 0 si no se ha visitado la casilla laberinto[i][j]; de lo contrario, es 1. Ambas matrices se definen desde la fila 0 hasta la fila **m + 1** y desde la columna 0 hasta la columna **n + 1** con el fin de controlar movimientos

inválidos cuando se esté en la fila 1 o en la fila **m** o en la columna 1 o en la columna **n**. Los datos correspondientes a las filas 0, filas **m + 1**, columnas 0 y columnas **n + 1** de dichas matrices valdrán 1, indicando que hacia esas casillas no se puede pasar. Estas filas y columnas adicionales evitan establecer controles adicionales cuando se esté en una fila o en una columna extrema.

Definimos también una matriz adicional para determinar hacia dónde se puede mover estando en una casilla de la fila **i** y de la columna **j**. Para una ubicación en la fila **i** columna **j** hay ocho posibles sitios hacia dónde trasladarse. Estos posibles movimientos los guardamos en la matriz de dos columnas que llamamos movimiento.

Por facilidad vamos a considerar que la entrada al laberinto es en la fila 1 columna 1 y que las salida está en la fila **m** columna **n**.

Utilizamos además una pila en la cual se van almacenando las casillas por las cuales se va pasando buscando la salida. Cuando se encuentre la salida se imprimirá el contenido de la pila indicando el camino que se siguió para salir del laberinto. Es conveniente aclarar que nuestro algoritmo imprimirá el primer camino que encuentre para salir del laberinto. Puede que haya más caminos y que el escrito no sea el más corto. Otro algoritmo podría ser escribir todos los posibles caminos y otro sería detectar el camino más corto para hallar la salida. Nuestro algoritmo es el siguiente:

```
1.  void  saleLaberinto(entero laberinto [][], entero  m, entero  n)
2.            entero  marca[m + 1][n + 1], movimiento[8][2]
3.            entero i, j, g, h, p, mov
4.            stack  pila = new  stack()
5.            movimiento[1][1] = –1
6.            movimiento[1][2] = 0
7.            movimiento[2][1] = –1
8.            movimiento[2][2] = +1
9.            movimiento[3][1] = 0
10.           movimiento[3][2] = +1
11.           movimiento[4][1] = +1
12.           movimiento[4][2] = +1
13.           movimiento[5][1] = +1
14.           movimiento[5][2] = 0
15.           movimiento[6][1] = +1
16.           movimiento[6][2] = –1
17.           movimiento[7][1] = 0
18.           movimiento[7][2] = –1
19.           movimiento[8][1] = –1
20.           movimiento[8][2] = –1
21.           for (i = 1; i <= m; i ++)  do
22.                     for (j = 1; j <= n; j++)  do
23.                               marca[i][j] = 0
24.                     end(for)
```

```
25.          end(for)
26.          for (i = 1; i <= n; i++)  do
27.                    marca[0][i] = 1
28.                    marca[m + 1][i] = 1
29.          end(for)
30.          for (i = 1; i <= n; i++)  do
31.                    marca[i][0] = 1
32.                    marca[i][m + 1] = 1
33.          end(for)
34.          marca[1][1] = 1
35.          pila.apilar(1)
36.          pila.apilar(1)
37.          pila.apilar(2)
38.          while  (no pila.esVacia())  do
39.                    mov = pila.desapilar()
40.                    j = pila.desapilar()
41.                    i = pila.desapilar()
42.                    while  (mov <= 8)  do
43.                              g = movimiento[mov][1]
44.                              h = movimiento[mov][2]
45.                              if  ((g == m)  and  (h == n))  then
46.                                        for  (p = 1; p <= pila.retornaTope(); p= p + 3)  do
47.                                                  write(pila.retornaDato(p))
48.                                                  write(pila.retornaDato(p + 1)
49.                                         end(for)
50.                                         write(i, j)
51.                                         write(m, n)
52.                                         return
53.                              end(if)
54.                              if ((laberinto[g][h] == 0)  and  (marca[g][h] == 0))  then
55.                                        marca[g][h] = 1
56.                                        pila.apilar(i)
57.                                        pila.apilar(j)
58.                                        pila.apilar(mov)
59.                                        mov = 0
60.                                        i = g
61.                                        j = h
62.                              end(if)
63.                              mov = mov + 1
64.                    end(while)
65.          end(while)
66.          write("no hay salida del laberinto")
67.  fin(saleLaberinto)
```

En las instrucciones 2 a 4 definimos las variables locales de trabajo. En las instrucciones 5 a 33 asignamos los valores iniciales de las matrices marca y movimiento.

En la instrucción 34 señalamos la casilla inicial como visitada y en las siguientes tres instrucciones se guardan en la pila las coordenadas en las cuales se halla la persona y el próximo movimiento que debe efectuar. El ciclo de intruciones 38 a 65 es el corazón del algoritmo. En la pila se almacena el camino que se sigue en busca de la salida y el movimiento con el que se debe continuar, en caso de que el camino elegido no tenga salida. El ciclo terminará cuando la pila quede vacía, es decir, cuando para cada casilla del laberinto se hayan examinado todos los posibles caminos y todos estén bloqueados. La salida se detecta en la instrucción 45; cuando dicha condición sea verdadera se imprimen todos los datos de la pila, correspondientes a la ruta que se ha seguido para salir del laberinto.

Autoevaluación 13

La presente autoevaluación está desarrollada con base en los ejercicios 1, 3, 5 y 7 propuestos en el texto guía.

1. Elabore un algoritmo para convertir una expresión de infijo a prefijo.

Solución:

A nuestro algoritmo lo denominaremos inToPre, será tipo void y tendrá un parámetro, llamado **s**, que es la hilera que contiene una expresión en infijo y que se desea convertir a prefijo. El proceso es similar al desarrollado en el numeral 13.4 del texto guía en el cual se convierte una expresión infijo a posfijo. La principal diferencia es que la hilera infijo se debe recorrer en sentido inverso, es decir, desde el último símbolo hacia el primero. Para lograr esto, lo primero que haremos será guardar la expresión infijo (la hilera **s**) en una pila, y luego procesamos la pila. Esto es equivalente a recorrer la hilera **s** en sentido inverso. Como en el ejercicio del numeral 13.4, al procesar la hilera se deben guardar los operadores en una pila con el fin de llevarlos hacia la expresión prefijo en el momento indicado. Cuando se está convirtiendo desde infijo hacia posfijo (numeral 13.4 del texto guía) todo operador entra a la pila, pero antes de entrarlo controlamos que la prioridad del operador que está en el tope de la pila sea mayor o igual que la prioridad del operador que va a entrar. Si la prioridad del operador que está en el tope de la pila es mayor o igual que la prioridad del operador que va a entrar a la pila se desapila el operador que está en el tope de la pila y se lleva hacia la expresión posfijo que se está construyendo. Este proceso se repite hasta que la pila quede vacía o que la prioridad del operador que está en el tope de la pila sea menor que la prioridad del operador que va a entrar. En este caso, conversión desde infijo hacia prefijo, se sacarán operadores de la pila hacia la expresión cuando la prioridad del operador que está en el tope de la pila sea mayor que la prioridad del operador que va a entrar a la pila.

Esas son las dos diferencias: recorrer la expresión infijo en sentido inverso y la condición para sacar los operadores de la pila de operadores hacia la expresión prefijo.

El otro aspecto es que la expresión prefijo se va a construir en una pila, porque debido a nuestra técnica de conversión la expresión prefijo queda en sentido inverso. Al construir la expresión prefijo en una pila, bastará con sacarla de la pila y se obtiene en el orden correcto.

Solución:

Teniendo definido el método de conversión, veamos nuestro algoritmo:

```
1.  void inToPre(hilera s)
2.          stack pilaIn, pilaTra, pilaPre
3.          hilera x, y
4.          x = s.siguienteToken()
5.          while (x != "┤") do
6.                  pilaIn.apilar(x)
7.                  x = s.siguienteToken()
8.          end(while)
9.          while (no pilaIn.esVacia()) do
10.             x = pilaIn.desapilar()
11.             casos de x
12.                 in operador:
13.                     while (no pilaTra.esVacia() and pdp(pilaTra.cima()) > pfp(x))
                                do
14.                             y = pilaTra.desapilar()
15.                             pilaPre.apilar(y)
16.                     end(while)
17.                     pilaTra.apilar(x)
18.                 "(":
19.                         while (pilaTra.cima() != ")") do
20.                                 y = pilaTra.desapilar()
21.                                 pilaPre.apilar(y)
22.                         end(while)
23.                         y = pilaTra.desapilar()
24.                 else:
25.                         pilaPre.apilar(x)
26.             fin(casos)
27.             x = s.siguienteToken()
28.         end(while)
29.         while (no pilaTra.esVacia()) do
30.                 y = pilaTra.desapilar()
31.                 pilaPre.apilar(y)
32.         end(while)
33.         while (no pilaPre.esVacia()) do
34.                 y = pilaPre.desapilar()
35.                 write(y)
36.         end(while)
37.  fin(inToPre)
```

En la instrucción 2 definimos las pilas con las cuales vamos a trabajar: pilaIn, que es aquella en la cual se guarda la hilera **s** con el fin de recorrerla en sentido inverso; pilaTra, que es aquella en la cual se guardan los operadores que se van leyendo para luego llevarlos a la expresión prefijo en el momento apropiado; y pilaPre, que es aquella en la cual guardamos la expresión prefijo que se está construyendo.

En la instrucción 3 definimos las variables auxiliares **x** y **y**.

En las instrucciones 5 a 8 se guarda la expresión infijo (la hilera **s**) en la pila pilaIn.

En instrucciones 9 a 28 se procesa la pila pilaIn. Cada carácter que se saca de la pila (instrucción 10) se evalúa con la instrucción casos de la línea 11. Si es un operador se debe llevar a la pila, pero antes de entrarlo controlamos que la pila no esté vacía y que la prioridad del operador que está en el tope de la pila sea mayor que la prioridad del operador que va a entrar a la pila. En caso de que estas condiciones sean verdaderas sacamos el operador que está en el tope de la pila y lo llevamos a la pila en la cual se está construyendo la expresión prefijo (instrucciones 13 a 16). En la instrucción 17 guardamos el operador en la pila de trabajo. En las instrucciones 18 a 23 se procesa el abre paréntesis: simplemente sacamos los símbolos de la pila de trabajo hacia la expresión prefijo (pilaPre) hasta hallar en el tope de la pila un cierre paréntesis. Cuando esto sucede nos salimos del ciclo y desapilamos el cierre paréntesis. Si el carácter extraído de la pila en la cual está la expresión infijo es un operando simplemente se lleva a la pila en la cual se está construyendo la expresión prefijo (instrucción 25).

Al terminar el ciclo en el cual se procesa la expresión infijo se llevan los operadores restantes que están en la pila de trabajo hacia la pila en la cual se construye la expresión prefijo (instrucciones 29 a 32).

Por último, vaciamos la pila en la cual fue construida la expresión prefijo para obtenerla en la forma correcta.

3. Elabore un algoritmo para convertir una expresión de posfijo a infijo.

Solución:

Para convertir una expresión en posfijo hacia infijo hay que tener en cuenta que la expresión debe salir completamente patentizada con el fin de garantizar que la expresión posfijo esté correcta.

Nuestro método consistirá en recorrer la expresión posfijo en sentido inverso y utilizar una pila, que llamaremos pilaIn, en la cual se va construyendo la expresión infijo correspondiente a la expresión posfijo. Para recorrer la expresión posfijo en sentido inverso la guardamos en una pila que llamamos pilaPos. Cuando procesemos esta pila puede suceder que cada carácter sea un operador o un operando. Si es un operador lo guardamos en la pila en la cual construiremos la expresión infijo; de lo contrario, controlamos de qué tipo es el símbolo que se halla en el tope de la pila pilaIn: si es un operador, simplemente guardamos en dicha pila el operando que se sacó de la pila pilaPos; de lo contrario, el símbolo que se halla en el tope de la pila será el segundo operando correspondiente a una operación, cuyo operador es el símbolo que quedó en el tope de la pila pilaIn: desapilamos dicho operador y concatenamos abre paréntesis, el operando sacado de la pila pilaPos, el operador sacado de la pila pilaIn, el operando sacado de la pila pilaPos y

un cierre paréntesis. Esta hilera resultante se lleva a la pila pilaIn cuando el símbolo que haya quedado en el tope de pilaIn sea un operador; de lo contrario, repetimos este proceso que acabamos de describir hasta que en el tope de la pila pilaIn haya un operador.

El algoritmo para ejecutar la tarea aquí descrita se presenta a continuación.


```
1.  void posToIn(hilera s)
2.          stack pilaPos, pilaIn
3.          hilera x, y
4.          x = siguienteToken()
5.          while (x != "┤") do
6.                  pilaPos.apilar(x)
7.                  x = siguienteToken()
8.          end(while)
9.          while (no pilaPos.esVacia()) do
10.                 x = pilaPos.desapilar()
11.                 if (x in operadores) then
12.                         pilaIn.apilar(x)
13.                 else
14.                         if (pilaIn.cima() in operadores) then
15.                                 pilaIn.apilar(x)
16.                         else
17.                                 while (no pilaIn.esVacia() and
18.                                         pilaIn.cima() no in operadores) do
19.                                     y = pilaIn.desapilar()
20.                                     opdor = pilaIn.desapilar()
21.                                     x = "(" + x + opdor + y + ")"
22.                                 end(while)
23.                                 pilaIn.apilar(x)
24.                         end(if)
25.                 end(if)
26.         end(while)
27.         x = pilaIn.desapilar()
28.         write(x)
29. fin(posToIn)
```

A nuestro método lo denominamos posToIn, es de tipo void y tiene un parámetro, que llamamos **s**, en el cual se halla la hilera que representa una expresión posfijo y que se desea convertir a infijo.

En las instrucciones 2 y 3 definimos nuestras variables locales de trabajo. En las instrucciones 5 a 8 guardamos la expresión posfijo (la hilera **s**) en la pila pilaPos con el fin de procesarla en sentido inverso.

El proceso descrito anteriormente se ejecuta con el ciclo de las instrucciones 9 a 26. Con

la instrucción 10 sacamos un carácter de la pila pilaPos. Si dicho carácter, que llamamos **x**, es un operador simplemente lo llevamos hacia la pila pilaIn; de lo contrario, efectuamos el proceso descrito con el ciclo de las instrucciones 17 a 23.

Al terminar de ejecutar el ciclo de las instrucciones 9 a 26 en el tope de la pila pilaIn está la expresión en infijo. Procedemos a desafilarla y a escribirla (instrucciones 27 y 28).

5. Elabore un algoritmo para convertir una expresión de prefijo a posfijo.

Solución:

Para convertir una expresión prefijo a posfijo basta con utilizar una pila. La expresión prefijo la recorreremos de izquierda a derecha, carácter por carácter hasta terminar de recorrerla. Si el carácter que se está procesando es un operador, simplemente lo guardamos en la pila; de lo contrario, se procede a escribirlo en la expresión posfijo y a controlar si en el tope de la pila hay un operador o un operando. Si en el tope hay un operador se procede a apilar el operando leído; de lo contrario, se desafila el operando que está en el tope y luego se lleva hacia la expresión posfijo el operador que quedó en el tope de la pila. Este proceso se repite hasta que en el tope de la pila quede un operador para proceder a almacenar el operando leído en la pila. El proceso termina cuando el símbolo que se lea sea el de fin de secuencia (┤). El algoritmo para efectuar esta tarea es el siguiente:

```
1.  void  preToPos(hilera  s)
2.            stack  pila = new  stack()
3.            hilera  x
4.            x = s.siguienteToken()
5.            while  (x != "┤ ")  do
6.                        if  (x in operadores)  then
7.                                    pila.apilar(x)
8.                        else
9.                                    write(x)
10.                                   if  (pila.cima() in operadores)  then
11.                                               pila.apilar(x)
12.                                   else
13.                                               while  ((no pila.esVacia())  and  pila.cima()
                                                            no  in operadores)  do
14.                                                           y = pila.desapilar()
15.                                                           y = pila.desapilar()
16.                                                           write(y)
17.                                               end(while)
18.                                               pila.apilar(x)
19.                                   end(if)
20.                       end(if)
21.                       x = s.siguienteToken()
22.           end(while)
23.  fin(preToPos)
```

A nuestro algoritmo lo llamamos preToPos, es de tipo void y tiene como parámetro una hilera s en la cual se halla la hilera prefijo que se desea convertir a posfijo.

En las instrucciones 2 y 3 definimos las variables locales de trabajo. En el ciclo de las instrucciones 5 a 22 se efectúa el proceso descrito anteriormente. Si el símbolo leído es un operador simplemente se lleva a la pila (instrucciones 6 y 7); de lo contrario, se escribe en la expresión posfijo y se evalúa cuál es el símbolo que hay en el tope (instrucción 10): si es un operador basta con apilar el operando en la pila; de lo contrario, significa que en el tope hay un operando, por consiguiente se saca de la pila ese operando (instrucción 14), y como en el tope debe haber quedado un operador, se desapila dicho operador y se lleva a la expresión posfijo (instrucciones 15 y 16). Terminado este ciclo se procede a apilar el operando leído (instrucción 18).

7. Elabore un algoritmo para evaluar una expresión en prefijo.

Solución:

Para evaluar una expresión en prefijo basta con recorrerla de izquierda a derecha e ir procesando carácter por carácter utilizando una pila. Si se lee un operador lo guardamos en la pila. Si se lee un operando se evalúa si en el tope de la pila hay un operando o un operador: si hay un operador simplemente se apila el operando leído; de lo contrario, se desapila el operando que hay en el tope y el operador que está ahí debajo con el fin de aplicárselo a dichos operandos: el leído y el que se desapiló. El resultado de esta operación se guarda en la pila y se continúa el proceso. El algoritmo para ejecutar este proceso es el siguiente:

```
1.  float evaPre(hilera s)
2.          stack  pila = new stack()
3.          float  res, op1
4.          hilera  opdor, x
5.          x = s.siguienteToken()
6.          while (x != "┤") do
7.                  if (x in operadores) then
8.                          pila.apilar(x)
9.                  else
10.                         if (pila.cima() in operadores) then
11.                                 pila.apilar(x)
12.                         else
13.                                 op1 = pila.desapilar()
14.                                 opdor = pila.desapilar()
15.                                 casos de opdor
16.                                         "^":  res = op1 ^ x
17.                                         "*":  res = op1 * x
18.                                         "/":  res = op1 / x
19.                                         "%": res = op1 % x
```

```
20.                                                    "+":  res = op1 + x
21.                                                    "–" : res = op1 – x
22.                                            fin(casos)
23.                                            pila.apilar(res)
24.                                    end(if)
25.                            end(if)
26.                            x = s.siguienteToken()
27.                    end(while)
28.                    return  pila.cima()
29.    fin(evaPre)
```

A nuestro algoritmo lo llamamos evaPre, retorna un dato de tipo float y tiene como parámetro un objeto de la clase hilera (llamado **s**) en el cual se halla la expresión prefijo a evaluar.

En las instrucciones 2 a 4 se definen las variables locales de trabajo. En el ciclo de las instrucciones 6 a 27 se efectúa el proceso descrito. Si el carácter leído representa un operador se almacena en la pila (instrucciones 7 y 8). Si el carácter leído es un operando se apila o se procesa. Si en el tope de la pila hay un operador, simplemente se apila (instrucciones 10 y 11); de lo contrario, el operando en el tope de la pila será el primer operando y el operador será el que quedó en el tope, el cual también se desapila (instrucciones 13 y 14). Luego, dependiendo del operador que se haya desapilado, se efectúa la operación correspondiente (instrucciones 15 a 22) y su resultado se lleva nuevamente a la pila. El proceso termina cuando se lea el símbolo de fin de secuencia.

En el tope de la pila se halla el resultado de evaluar la expresión; por lo tanto, se retorna dicho resultado con la instrucción 28.

Autoevaluación 14

La presente autoevaluación está desarrollada con base en los ejercicios 1, 2 y 5 propuestos en el módulo 14 del texto guía.

1. Escriba una función recursiva que calcule e imprima 2n utilizando únicamente la operación de suma. Con base en su definición construya su correspondiente algoritmo.

   Solución:

   $$2^n = \begin{cases} 1, \text{si } n == 0 \\ 2^{n-1} + 2^{n-1} \end{cases}$$

   Para elaborar esta definición tuvimos en cuenta dos cosas. Primero, la definición del caso base, es decir, el punto de parada. Según las definiciones matemáticas se conoce que todo número elevado a la potencia 0 produce como resultado 1. Por lo tanto, $2^0 = 1$, o sea que cuando **n** sea igual a 0 el resultado es 1 (primera parte de la definición). Ahora, con base también en las propiedades matemáticas, se sabe que $2^n = 2*2^{n-1}$, pero como nos están restringiendo a que sólo se utilice la operación de suma, $2*2^{n-1}$ es igual a $2^{n-1} + 2^{n-1}$, con lo cual obtuvimos la segunda parte de la definición. Teniendo elaborada la definición recursiva, la construcción del algoritmo recursivo es prácticamente directa. Basta con escribir en forma algorítmica la definición dada. Dicho algoritmo se presenta a continuación.

   ```
   1.  int  potenciaDeDos(int  n)
   2.           if  (n == 0)  then
   3.                    x = 1
   4.           else
   5.                    x = potenciaDeDos(n – 1)
   6.                    x = x + x
   7.           end(if)
   8.           return  x
   9.  fin(potenciaDeDos)
   ```

2. La función de Ackerman se define así:

   ack(m, n) = n + 1 si m = 0
   ack(m, n) = ack(m – 1,1) si n = 0
   ack(m, n) = ack(m – 1, ack(m, n – 1)) en otro caso.

   Escriba un algoritmo recursivo para evaluar dicha función.

Solución:

```
1.  entero  ackerman(entero  m, entero  n)
2.          entero  x
3.          if  (m == 0)  then
4.                  x = n + 1
5.          else
6.                  if  (n == 0)  then
7.                          x = ackerman(m – 1, 1)
8.                  else
9.                          x = ackerman(m, n – 1)
10.                         x = ackerman(m – 1, x)
11.                 end(if)
12.         end(if)
13.         return  x
14.  fin(ackerman)
```

Como podrá observar es un algoritmo bastante simple. Con base en la definición recursiva hemos escrito las instrucciones correspondientes a cada una de las partes de la definición. Se ha utilizado la variable **x** para retornar el valor de la función cualquiera que sea el caso que se presente. Las instrucciones 3 y 4 corresponden al caso en que **m** sea igual a 0 (primera parte de la definición); las instrucciones 6 y 7 corresponden a la segunda parte de la definición, es decir, cuando **n** sea 0; y las instrucciones 9 y 10 al tercer caso de la definición, es decir, cuando **m** y **n** son diferentes de 0.

5. Haga seguimiento a los siguientes algoritmos recursivos y determine la diferencia entre ellos:

```
void  recorreLista1(nodoSimple  L)
      if  (L !=  null)  then
              write(L.retornaDato())
              recorreLista1(L.retornaLiga())
      end(if)
fin(recorreLista)
```

(1)

```
void  recorreLista2(nodoSimple  L)
    if  (L != null)  then
          recorreLista2(L.retornaLiga())
           write(L.retornaDato())
    end(if)
fin(recorreLista)
```

(2)

al ser invocados por el siguiente programa principal:

```
q = a.primerNodo()
recorreLista1(q)
recorreLista2(q)
```

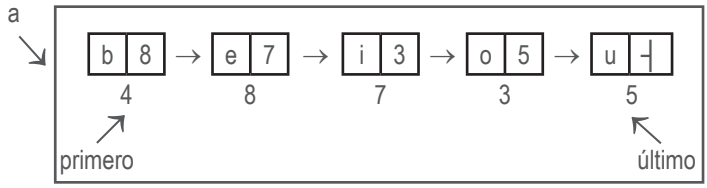


Figura 14.1

Solución:

Comencemos analizando el algoritmo (1). La instrucción de retorna, que llamaremos L1, es el end(if). Definimos una pila vacía en la cual se guardará la dirección de retorno y el parámetro por valor L. No hay variables locales; por consiguiente, esos son los únicos datos a trabajar en la pila.

Reescribamos el algoritmo con su correspondiente programa principal con el fin de ilustrar mejor lo que son las direcciones de retorno:

```
1.  void  recorreLista1(nodoSimple  L)
2.            if  (L !=  null)  then
3.                        write(L.retornaDato())
4.  L1:   ⟶               recorreLista1(L.retornaLiga())
5.            end(if)
6.  fin(recorreLista)

              programaPrincipal
                        q = a.primerNodo()
                        recorreLista1(q)
L0:  ⟶  fin(programaPrincipal)
```

Recuerde que como nuestro algoritmo es tipo void la instrucción de retorno es la instrucción siguiente a la instrucción que invoca el subprograma recursivo. En el programa principal hemos llamado esta instrucción L0, y en el subprograma recorreLista1 la hemos llamado L1.

En la figura 14.2 presentamos la forma como va evolucionando la pila a medida que se ejecuta el algoritmo:

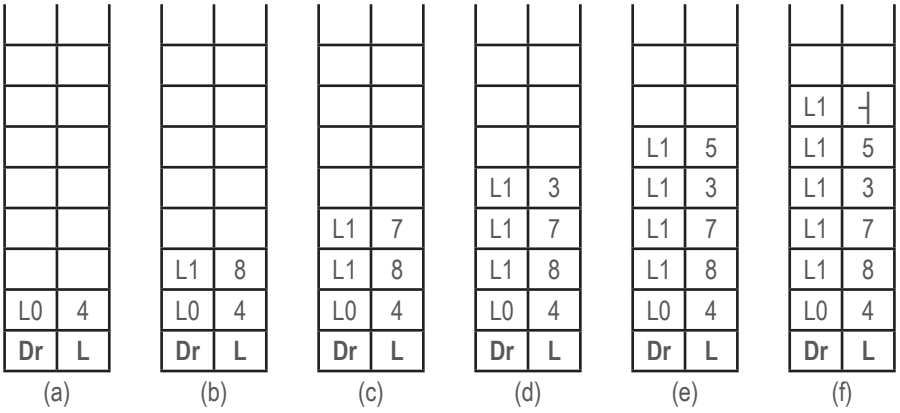


Figura 14.2

Al estar ejecutando el programa principal, a la variable **q** se le asigna el primer nodo del objeto **a** mostrado en la figura 14.1 y luego se invoca el subprograma recorreLista1 enviándole como parámetro el contenido de **q**, es decir, 4. Al ejecutar la instrucción recorreLista1(q), ésta es una interrupción a la ejecución del programa principal; por lo tanto, hay que guardar en la pila la dirección de retorno correspondiente a esta interrupción y el valor del parámetro con el cual hay que ejecutar el subprograma recorreLista1. Esta acción se muestra en el caso (a) de la figura 14.2. Este estado de la pila significa: se va a ejecutar el subprograma recorreLista1 con la variable **L** valiendo 4 y cuando termine debe regresar a la instrucción llamada L0 para continuar con la ejecución del programa principal.

Al ejecutar el subprograma recorreLista1 con la **L** valiendo 4, la condición de la instrucción 2 es verdadera, y por lo tanto ejecutará las instrucciones 3 y 4. Con la instrucción 3 escribe el dato que hay en el nodo 4, es decir, la **b**, y en la instrucción 4 encuentra el llamado al subprograma recorreLista1, lo cual es una interrupción a la ejecución del subprograma recorreLista1 con el valor de la **L** en 4; por consiguiente, hay que guardar en la pila la dirección de retorno correspondiente a esta interrupción y el valor con el cual se invoca el subprograma, el cual es el contenido del campo de liga del nodo 4, cuyo valor es 8. Esta acción se presenta en el caso (b) de la figura 14.2. Este caso significa: se va ejecutar el subprograma recorreLista1 con la **L** valiendo 8, cuando termine debe regresar a la instrucción llamada L1 para continuar la ejecución de recorreLista1 con la L valiendo 4, y cuando termine esta ejecución debe retornar a la instrucción llamada L0 para continuar con la ejecución del programa principal.

Al ejecutar el subprograma recorreLista1 con la **L** valiendo 8, la condición de la instrucción 2 es verdadera, y por lo tanto ejecutará las instrucciones 3 y 4. Con la instrucción 3 escribe el dato que hay en el nodo 8, es decir, la **e**, y en la instrucción 4 encuentra el llamado al subprograma recorreLista1, lo cual es una interrupción a la ejecución del subprograma recorreLista1 con el valor de la **L** en 8; por consiguiente, hay que guardar

en la pila la dirección de retorno correspondiente a esta interrupción y el valor con el cual se invoca el subprograma, el cual es el contenido del campo de liga del nodo 8, cuyo valor es 7. Esta acción se presenta en el caso (c) de la figura 14.2.

Se continúa con este proceso hasta que el llamado se efectúe con el valor de **L** en ┤, cuya situación se presenta en el caso (f) de la figura 14.2. Fíjese que en el proceso que ha venido efectuando ha escrito el dato de cada nodo, y por consiguiente la impresión es: **b**, **e**, **i**, **o**, **u**.

Cuando estamos en el caso (f) de la figura 14.2 significa que se va a ejecutar el subprograma recorreLista1 con la L valiendo ┤. Al evaluar la condición de la instrucción 2, ésta resulta en falso; por consiguiente, no ejecuta las instrucciones 3 y 4 y continúa la ejecución en la instrucción siguiente al end(if). Dicha instrucción es el fin del subprograma, lo que significa que terminó de ejecutar el subprograma recorreLista1 con la L valiendo ┤. La acción a efectuar aquí es que desapila los datos que están en el tope de la pila y retorna a la instrucción cuya dirección de retorno acabó de desapilar, es decir, L1.

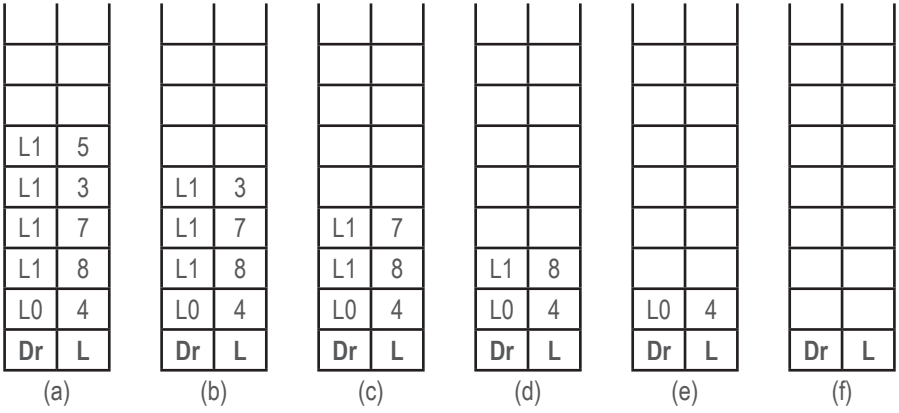


Figura 14.3

La situación queda como en el caso (a) de la figura 14.3, el cual significa que va a continuar la ejecución del subprograma recorreLista1 con la L valiendo 5, y cuando termine retornará a la instrucción llamada L1 para continuar con la ejecución de recorreLista1 con el valor de la L en 3.

El proceso continúa de esta forma hasta que la pila quede vacía.

Consideremos ahora el algoritmo (2). La instrucción de retorna, que llamaremos L1, es el end(if). Definimos una pila vacía en la cual se guardará la dirección de retorno y el parámetro por valor L. No hay variables locales; por consiguiente, esos son los únicos datos a trabajar en la pila.

Reescribamos el algoritmo con su correspondiente programa principal con el fin de ilustrar mejor lo que son las direcciones de retorno:

```
1.  void  recorreLista2(nodoSimple  L)
2.            if  (L !=  null)  then
3.                        recorreLista2(L.retornaLiga())
4.  L1: ⟶                 write(L.retornaDato())
5.            end(if)
6.  fin(recorreLista)

              programaPrincipal
                        q = a.primerNodo()
                        recorreLista2(q)
L0:     ⟶ fin(programaPrincipal)
```

Recuerde que como nuestro algoritmo es tipo void la instrucción de retorno es la instrucción siguiente a la instrucción que invoca el subprograma recursivo. En el programa principal hemos llamado a esta instrucción L0, y en el subprograma recorreLista2 la hemos llamado L1.

Realmente, las figuras 14.2 y 14.3 sirven para hacer el seguimiento para este nuevo algoritmo. La diferencia se presenta en que cuando retorna a L1, esta instrucción es la instrucción para escribir el dato de L. Esto nos lleva a que cuando estamos en la situación (a) de la figura 14.3 escribirá el dato del nodo 5 (es decir, la "u"), cuando estemos en el caso (b) de la misma figura escribe la "o", y así sucesivamente. Como resultado se observa que imprime los datos de la lista en sentido inverso.

En otras palabras, usando la recursión se puede recorrer una lista simplemente ligada en sentido inverso, sin tener que reversar sus apuntadores.

Autoevaluación 15

La presente autoevaluación está desarrollada con base en los ejercicios 1, 3 y 5 propuestos en el módulo 15 del texto guía.

1. Escriba una función recursiva que evalúe el determinante asociado a una matriz cuadrada de orden **n** utilizando el método de los menores.

   Solución:

```
1.  float  determinante(float  matriz[][], entero  n)
2.          entero  i, s, l, k
3.          float  valdet,  menor[][]
4.          valdet = 0
5.          if  (n == 1)  then
6.                valdet = matriz[1][1]
7.          else
8.                for  (i = 1; i <= n; i++)  do
9.                     for  (k = 2; k <= n; k++) do
10.                         s = 1
11.                         for  (l = 1; l <= n; l++) do
12.                              if  (l <> i)  then
13.                                   menor[k – 1][s] = matriz[k][l]
14.                                    s = s + 1
15.                              end(if)
16.                         end(for)
17.                    end(for)
18.                    if  (i % 2 == 1)  then
19.                         valdet = valdet + matriz[1][i]*determinante(menor, n – 1)
20.                    else
21.                         valdet = valdet –  matriz[1][i]*determinante(menor, n – 1)
22.                    end(if)
23.               end(for)
24.         end(if)
25.         return  valdet
26.  fin(determinante)
```

A nuestro método lo llamamos determinante y retorna un dato de tipo real (float). Tiene como parámetros la matriz a la cual se le desea calcular el determinante y el orden de dicha matriz. En las instrucciones 2 y 3 se definen las variables locales de trabajo.

Con el fin de calcular el valor del determinante usando el método de menores, recorreremos la fila 1 de la matriz entrada como parámetro y construiremos el menor de cada ocasión eliminando siempre la fila 1 y la columna **i**. La construcción del menor se efectúa en los ciclos planteados en las instrucciones 9 a 17. El recorrido de la fila 1 se hace con el ciclo de la instrucción 8. Cuando se termine de construir el menor correspondiente a cada llamada debemos controlar si el producto del dato de la posición fila 1, columna i, multiplicado por el valor del determinante de su correspondiente menor se debe sumar o restar al valor del determinante que se está calculando. El producto se suma cuando la columna es impar y se resta cuando la columna es par. Dicho control se establece en las instrucciones 18 a 22.

Fíjese que la llamada recursiva, instrucciones 19 y 21, se hace enviando el menor construido en las instrucciones 9 a 17 y el tamaño de la matriz entrada como parámetro disminuido en 1. Cuando el valor de de **n** sea 1 significa que se llegó al caso base; por lo tanto, el valor del determinante es simplemente el contenido del elemento de la fila 1, columna 1.

3. Escriba un algoritmo recursivo que imprima las 2^n posibles combinaciones de **n** variables lógicas.

Solución:

```
1.  void  combina(boolean  V[], entero  i,  entero  n)
2.            if  (i > n)  then
3.                        write(V)
4.            else
5.                        combina(V, i+1, n)
6.                        V[i] = no V[i]
7.                        combina(i+1, n)
8.            end(if)
9.   fin(combina)
```

A nuestro método lo denominamos combina y es de tipo void puesto que no retorna dato alguno sino que ejecuta una tarea sobre los datos enviados como parámetros. Tiene como parámetros un vector de tipo lógico (boolean) y dos enteros **i** y **n**. El parámetro **i** indica a partir de cuál posición se deben hacer combinaciones y el parámetro **n** hasta cuál posición se efectúan dichas combinaciones.

La técnica que usamos para construir las diferentes posibles combinaciones es dejar invariable el dato de la posición **i** y efectuar las combinaciones de los **n – i** restantes elementos. Cuando se terminen de efectuar las combinaciones de estos últimos **n – i** elementos le cambiamos el estado al dato de la posición **i** y volvemos a generar las combinaciones de los últimos **n – i** elementos. El proceso termina cuando el valor de la **i** sea mayor que **n**.

Esquemáticamente, si tenemos cuatro variables lógicas el proceso es:

	1 2 3 4
1	T T T T
2	T T T F
3	T T F T
4	T T F F
5	T F T T
6	T F T F
7	T F F T
8	T F F F
9	F T T T
10	F T T F
11	F T F T
12	F T F F
13	F F T T
14	F F T F
15	F F F T
16	F F F F

En las líneas 1 a 8 hemos dejado invariable el estado del dato de la posición 1 y hemos generado las combinaciones de los datos desde la posición 2 hasta la posición 4 (llamada recursiva de la instrucción 5). Terminado esto, le hemos cambiado el estado al dato de la posición 1 y hemos vuelto a generar las combinaciones de los datos desde la posición 2 hasta la posición 4 (llamada recursiva de la instrucción 7). En la instrucción 6 se le cambia el estado a la variable de la posición **i** del vector: si estaba en verdadero se pone en falso, y viceversa.

5. La información correspondiente a los padres de un mico se guardan en una tripleta de la siguiente forma: el primer dato corresponde al código del mico, el segundo dato al código del papá del mico y el tercer dato al código de la mamá del mico. Dada una matriz de tripletas, escriba un algoritmo recursivo que imprima el árbol genealógico de un mico dado.

Solución:

```
1.  void arbolGenealogico(hilera mat[][], entero n, hilera s)
2.          entero i
3.          i = 1
4.          while (i <= n and mat[i][1] != s) do
5.                  i = i + 1
6.          end(while)
7.          if (i <= n) then
8.                  write(s, mat[i][2], mat[i][3])
9.                  arbolGenealogico(mat, n, mat[i][2]
```

```
10.                    arbolGenealogico(mat, n, mat[i][3]
11.            end(if)
12.  fin(arbolGenealogico)
```


A nuestro método lo llamamos arbolGenealogico y es de tipo void puesto que simplemente ejecuta una tarea, no retorna ningún valor. Los parámetros son una matriz de tres columnas (matriz de tripletas), la cual contiene los códigos de los micos, un entero **n** que indica el número de filas de dicha matriz y una hilera **s** que representa el código del mico cuyo árbol genealógico se desea mostrar.

En la instrucción 2 definimos nuestra variable de trabajo **i**, en la instrucción 3 inicializamos dicha variable en 1 y en el ciclo de las instrucciones 4 a 6 buscamos el código del mico comparando el dato de la fila **i** columna 1 con s (el código del mico enviado como parámetro). El ciclo termina cuando encuentre el mico o cuando haya recorrido toda la matriz y no lo haya encontrado. Si encontró el mico, el valor de la **i** es menor o igual que **n**, en cuyo caso procedemos a escribir los códigos de su padre y de su madre (instrucción 8) y continuamos buscando los ancestros del padre y de la madre haciendo llamadas recursivas con dichos códigos (instrucciones 9 y 10). Recuerde que el código del padre se halla en la columna 2 de la fila **i** y el de la madre en la columna 3 de la fila **i**.

Autoevaluación 16

La presente autoevaluación está desarrollada con base en los ejercicios 1 y 3 propuestos en el módulo 16 del texto guía.

3. Haga seguimiento detallado, paso por paso, al proceso de ordenamiento del vector mostrado a continuación usando el método quickSort.

	0	1	2	3	4	5	6	7	8
V	8	3	1	6	2	8	4	9	7

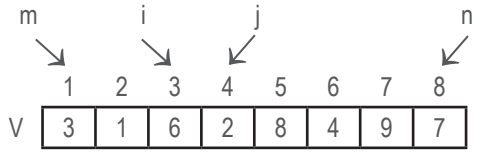
Solución:

Primero recordemos nuestro algoritmo de ordenamiento ascendente por el método quickSort:

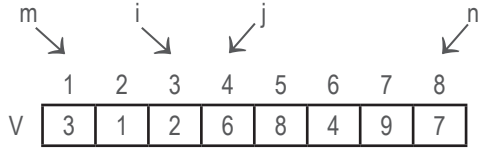
```
1.  void quickSort(entero m, entero n)
2.          entero i, j
3.          if (m < n) then
4.                  i = m
5.                  j = n + 1;
6.                  while (i < j) do
7.                          do
8.                                  i = i + 1
9.                          while ((i <= n) and (V[i] <= V[m]))
10.                         do
11.                                 j = j – 1
12.                         while ((j != m) and (V[j] >= V[m]))
13.                         if (i < j) then
14.                                 intercambia(i, j)
15.                         end(if)
16.                 end(while)
17.                 intercambia(j, m)
18.                 quickSort(m, j – 1)
19.                 quickSort(j + 1, n)
20.         end(if)
21. end(quickSort)
```

El llamado inicial es con **m** = 1 y **n** = 8; por lo tanto, los valores de **i** de **j** son 1 y 9 respectivamente, de acuerdo con las instrucciones 4 y 5. Al ejecutar los ciclos de las ins-

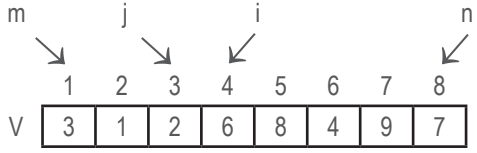
trucciones 7 a 12 los valores de **i** y de **j** quedan en 3 y 4, respectivamente. La situación es la siguiente:



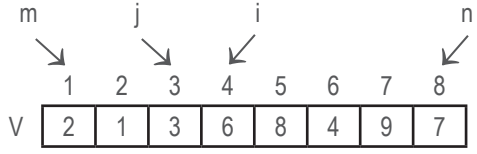
y al efectuar el intercambio (instrucciones 13 a 15) el vector queda así:



Al ejecutar nuevamente los ciclos de las instrucciones 7 a 12 los valores de **i** y **j** son 4 y 3, respectivamente. La situación es:

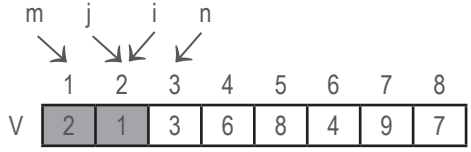


Como la **j** no es menor que la **i**, no se efectúa el intercambio de las instrucciones 13 a 15 y se sale del ciclo de la instrucción 6. Por consiguiente, continúa ejecutando el intercambio de la instrucción 17. El vector queda así:

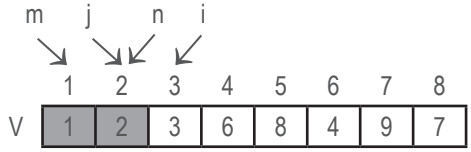


Al efectuar este intercambio el proceso a seguir es ordenar el subvector desde la posición 1 a la 2 (instrucción 18) y el subvector de la posición 4 a la 8 (instrucción 19).

Para ordenar el subvector desde la posición 1 a la posición 2 la ejecución del subprograma quickSort comienza con **m** = 1 y **n** = 2. Al ejecutar los ciclos desde la instrucción 7 a la 12 los valores de **i** y **j** quedan en 3 y 2, respectivamente. La situación es la siguiente:

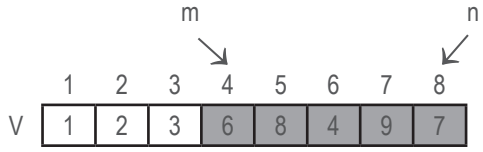


Como la **j** es mayor que la **i** no efectúa el intercambio de las instrucciones 13 a 15 y se sale del ciclo de la instrucción 6; por lo tanto, ejecuta el intercambio de la instrucción 17 y el vector queda así:

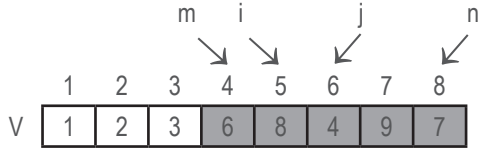


Al ejecutar nuevamente las instrucciones 18 y 19 con los valores de **m** y **n** en 1 y en 2, respectivamente, termina la ejecución de ordenar los datos anteriores a la posición 3.

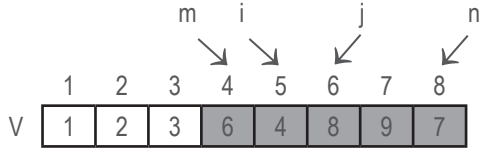
Ahora, ordenemos los datos desde la posición 4 a la 8. En esta llamada la **m** entra valiendo 4 y la **n** entra valiendo 8. La situación es:



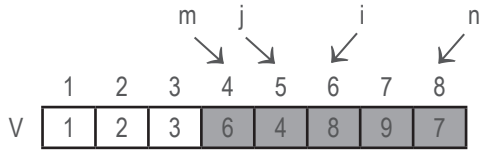
Al ejecutar los ciclos de la instrucción 7 a 12 los valores de **i** y **j** quedan en 5 y 6, respectivamente. La situación es:



Se efectúa el intercambio del dato de la posición **i** con el dato de la posición **j** y el vector queda así:

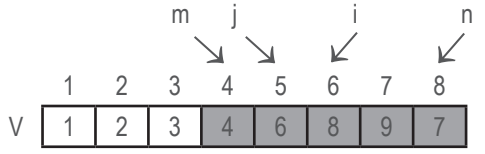


Y al ejecutar nuevamente los ciclos de la instrucción 7 a la 12 los valores de **i** y de **j** serán 6 y 5, respectivamente. La situación es:



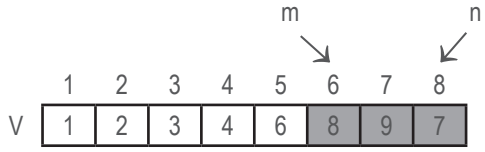
Como la **j** es mayor que la **i** no efectúa el intercambio de las instrucciones 13 a 15, se sale del ciclo de la instrucción 6 y continúa efectuando el intercambio de la instrucción 17.

El vector queda así:

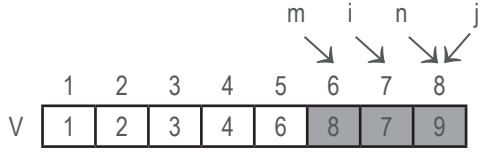


Y sólo falta por ordenar el subvector desde la posición 6 a la 8 ya que el subvector desde la posición 4 hasta la 4 (instrucción 18) es de un solo elemento y no ejecutará esta llamada.

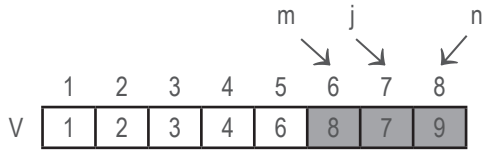
Para ordenar los datos desde la posición 6 hasta la 8 la situación es:



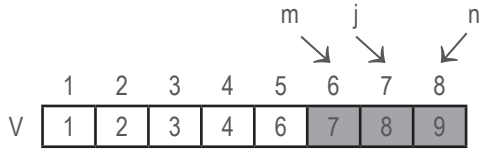
Al ejecutar los ciclos de las instrucciones 7 a 12 los valores de **i** y **j** serán 7 y 8, respectivamente. Por lo tanto, efectúa el intercambio de los datos que están en esas posiciones y la situación es:



Y al volver a ejecutar el ciclo de la instrucción 6 la **i** queda valiendo 10 y la **j** queda valiendo 7. La situación es:



En este punto la situación es que la **j** es mayor que la **i**, y por consiguiente no hace el intercambio de las instrucciones 13 a 15, se sale del ciclo de la instrucción 6 y procede a efectuar el intercambio de la instrucción 17. El vector queda así:



El cual, al efectuar las llamadas para ordenar los datos desde la posición 6 a la 6 (instrucción 18) y desde la posición 8 a la 8 (instrucción 19), no efectúa ninguna tarea y termina el proceso de ordenamiento.

3. Convierta a no recursivo el algoritmo de permutaciones visto en el módulo 15, numeral 15.2, del texto guía.

Solución:

Primero presentemos el algoritmo elaborado en dicho módulo.

```
1.  void  permuta(char  V[], entero  n, entero  i)
2.            entero  k
3.            if  (i == n)  then
4.                        write(V)
5.            else
6.                        for  (k = i; k <= n; k++)  do
7.                                    intercambie(i, k)
8.                                    permuta(V, n, i + 1)
9.   L1:                   end(for)
10.           end(if)
11.  fin(permuta)
```

La versión NO recursiva aplicando los pasos definidos en el numeral 16.3 del texto guía es:

```
1.  void  permuta(char  V[], entero  n, entero  i)
2.              objeto  pila = new pila[100]
3.              tope = 0
4.  L0:         if  (n == i)  then
5.                  write(v)
6.              else
7.                  for  (k = i; k <= n; k++)  do
8.                      intercambie(i, k)
9.                      pila[tope + 1] = v
10.                     pila[tope + 2] = n
11.                     pila[tope + 3] = i
12.                     pila[tope + 4] = k
13.                     pila[tope + 5] = 'L1'
14.                     tope = tope + 5
15.                     v = v
16.                     n = n
17.                     i = i + 1
18.                     goto L0
19. L1:             end(for)
20.             end(if)
21.             if  (tope > 0)  then
22.                 tope = tope – 5
23.                 v = pila[tope + 1]
24.                 n = pila[tope + 2]
25.                 i = pila[tope + 3]
26.                 k = pila[tope + 4]
27.                dr = pila[tope + 5]
28.                goto dr
29.             end(if)
30.  fin(permuta)
```

Esta es la primera versión resultante. Como dijimos anteriormente, un buen análisis del algoritmo permite eliminar instrucciones redundantes y los goto:

Es obvio que las instrucciones 15 y 16 sobran; sólo se guarda una dirección de retorno, (L1), luego las instrucciones 13 y 27 también sobran, basta cambiar la instrucción 28 por goto L1; la variable N conserva el mismo valor a través de todas las llamadas, por lo cual es tontería guardarla en la pila, o sea que las instrucciones 10 y 24 también sobran. Eliminando estas instrucciones nuestro algoritmo queda así:

```
1.  void  permuta(char  V[], entero  n, entero  i)
2.              objeto  pila = new pila[100]
3.              tope = 0
4.  L0:         if  (n == i)  then
```

```
5.                write(V)
6.            else
7.                for  (k = i;  k <= n; k++)  do
8.                     intercambie(i, k)
9.                     pila[tope + 1] = v
10.                    pila[tope + 2] = i
11.                    pila[tope + 3] = k
12.                    tope = tope + 3
13.                    i = i + 1
14.                    goto L0
15.  L1:            end(for)
16.           end(if)
17.           if  (tope > 0)  then
18.               tope = tope – 3
19.               v = pila[tope + 1]
20.                i = pila[tope + 2]
21.                k = pila[tope + 3]
22.               goto L1
23.           end(if)
24.  end(permuta)
```

La eliminación de los goto no es tan sencilla en este caso ya que cada vez que termina una llamada recursiva debe retornar a una instrucción end(for), cuya función es incrementar la variable controladora del ciclo en 1, y si es menor o igual que el valor final de dicha variable repite las instrucciones del ciclo. Lo que hace la instrucción for es, propiamente, asignar un valor inicial a la variable controladora del ciclo y controlar que sea menor o igual que el valor límite definido para entrar a ejecutar las instrucciones del ciclo.

Para poder eliminar los goto, nuestro primer paso será expresar el ciclo for con sus instrucciones elementales. Nuestro algoritmo queda así:

```
1.  void  permuta(char  V[], entero  n, entero  i)
2.          objeto  pila = new pila[100]
3.          tope = 0
4.  L0:   if  (n == i)  then
5.                write(V)
6.          else
7.                k = i
8.  L2:           if  (k <= n)  then
9.                     intercambie(i, k)
10.                    pila[tope + 1] = v
11.                    pila[tope + 2] = i
12.                    pila[tope + 3] = k
13.                    tope = tope + 3
14.                    i = i + 1
```

```
15.                    goto L0
16.  L1:               k = k + 1
17                     goto L2
18                 end(if)
19.            end(if)
20.            if  (tope > 0)  then
21.                tope = tope – 3
22.                v = pila[tope + 1]
23.                 i = pila[tope + 2]
24.                 k = pila[tope + 3]
25.                goto L1
26.            end(if)
27.  fin(permuta)
```

Teniendo el algoritmo con el ciclo que contiene la llamada recursiva expresado en sus instrucciones elementales, hay una técnica con la cual, identificando la instrucción o grupo de instrucciones que se ejecutan secuencialmente como un bloque y utilizando una variable adicional, llamémosla estado, se puede plantear un ciclo con una instrucción case que de acuerdo al valor de estado ejecute algún grupo de instrucciones y asigne a estado el nuevo valor según el conjunto de instrucciones con que continúe la ejecución.

En nuestro algoritmo los bloques de instrucciones secuenciales, "autónomos", que se identifican son los siguientes: instrucciones 4 a 7, las cuales ejecutaremos cuando estado=0; instrucciones 8 a 14, las cuales ejecutaremos cuando estado=1; instrucciones 20 a 24 junto con la instrucción 16 (esta última sólo se ejecuta cuando se hayan ejecutado las instrucciones 20 a 24), las cuales ejecutaremos cuando estado=2.

En cada uno de estos casos codificaremos dichas instrucciones y le asignaremos a la variable estado un nuevo valor dependiendo del bloque de instrucciones que se deban ejecutar a continuación de acuerdo a la lógica del algoritmo.

Con esta técnica, cualquier algoritmo, por complicada maraña de goto's que tenga, siempre se podrá convertir a un algoritmo sin goto's.

Inicialmente asignaremos 0 a la variable estado, y terminaremos el ciclo cuando estado=3. Nuestro algoritmo quedará así:

```
void  permuta(char  V[], entero  n, entero  i)
      objeto  pila = new pila[100]
      tope = 0
      estado = 0
      while  (estado != 3)  do
          casos de estado
              0:   if  (n == i)  then
                       write(V)
```

```
                    estado = 2
                else
                    k = i
                    estado = 1
                end(if)
            1:  if (k <= n) then
                    intercambie(i, k)
                    pila[tope + 1] = v
                    pila[tope + 2] = i
                    pila[tope + 3] = k
                    tope = tope + 3
                    i = i + 1
                    estado = 0
                else
                    estado = 2
                end(if)
            2:  if (tope > 0) then
                    tope = tope – 3
                    v = pila[tope + 1]
                    i = pila[tope + 2]
                    k = pila[tope + 3] + 1
                    estado = 1
                else
                    estado = 3
                end(if)
        fin(casos)
    end(while)
fin(permuta)
```

Udea

Para ser, saber y saber hacer

Educación a distancia