

MÓDULO 1

EVALUACION DE ALGORITMOS (CONCEPTOS Y EJEMPLOS)

Introducción

En el desarrollo de aplicaciones, utilizando el computador como herramienta, existen muchas formas de indicarle a la máquina que efectúe correctamente una tarea. Dentro de esas múltiples soluciones, es lógico pensar que habrá una solución mejor que otras, lo cual implica que se debe determinar cuál de ellas es la mejor. Presentamos en este módulo algunas técnicas que permitan identificar la mejor solución entre un grupo de ellas.

Objetivos

1. Aprender los conceptos referentes a evaluar un algoritmo.
2. Conocer la clasificación de los algoritmos en cuanto a su eficiencia.
3. Determinar la eficiencia de un algoritmo en cuanto a tiempo de ejecución.

Preguntas básicas

1. Qué es un contador de frecuencias?
2. Qué es Orden de magnitud?
3. Cuándo un algoritmo tiene orden de magnitud constante?
4. Cuándo un algoritmo tiene orden de magnitud lineal?
5. Cuándo un algoritmo tiene orden de magnitud cuadrático?
6. Cuándo un algoritmo tiene orden de magnitud cúbico?
7. Cuándo un algoritmo tiene orden de magnitud logarítmico?
8. Cuándo un algoritmo tiene orden de magnitud semilogarítmico?

1.1 Definición: La evaluación de algoritmos consiste en medir la eficiencia de un algoritmo en cuanto a consumo de memoria y tiempo de ejecución.

Anteriormente, cuando existían grandes restricciones tecnológicas de memoria, evaluar un algoritmo en cuanto a consumo de recursos de memoria era bastante importante, ya que dependiendo de ella los esfuerzos de programación y de ejecución serían grandes. En la actualidad, con el gran desarrollo tecnológico del Hardware, las restricciones de consumo de memoria han pasado a un segundo plano, por lo tanto, el aspecto de evaluar un algoritmo en cuanto a su consumo de memoria no es relevante.

En cuanto al tiempo de ejecución, a pesar de las altas velocidades de procesamiento que en la actualidad existen, sigue siendo un factor fundamental, especialmente en algoritmos que tienen que ver con el control de procesos en tiempo real y en aquellos cuya frecuencia de ejecución es alta.

Procedamos entonces a ver cómo determinar el tiempo de ejecución de un algoritmo. Básicamente existen dos formas de hacer esta medición: una que llamamos *a posteriori* y otra que llamamos *a priori*.

1.2 Evaluación a posteriori: El proceso de elaboración y ejecución de un algoritmo consta de los siguientes pasos:

- Elaboración del algoritmo.
- Codificación en algún lenguaje de programación.
- Compilación del programa.
- Ejecución del programa en una máquina determinada.

Cuando se ejecuta el programa, los sistemas operativos proporcionan la herramienta de informar cuánto tiempo consumió la ejecución del algoritmo.

Esta forma de medición tiene algunas inconvenientes: al codificar el algoritmo en algún lenguaje de programación estamos realmente midiendo la eficiencia del lenguaje de programación, ya que no es lo mismo un programa codificado en FORTRAN, que en PASCAL o que en C; al ejecutar un programa en una máquina determinada estamos introduciendo otro parámetro, el cual es la eficiencia de la máquina, ya que no es lo mismo ejecutar el programa en un XT, en un 386, en un 486, en un pentium, en un AS400, en un RISC, etc.

En otras palabras con esta forma de medición estamos es evaluando la calidad del compilador y de la máquina, mas no la del algoritmo, que es nuestro interés.

Lo que nos interesa es medir la eficiencia del algoritmo por el hecho de ser ese algoritmo, independiente del lenguaje en el que se haya codificado y de la máquina en la cual se ejecute.

Veamos entonces la forma de medir la ejecución de un algoritmo *a priori*.

1.3 Evaluación a priori: Se necesita definir dos conceptos básicos que son: **contador de frecuencias** y **orden de magnitud**.

1.3.1 Contador de frecuencias: es una expresión algebraica que indica el número de veces que se ejecutan las instrucciones de un algoritmo. Para ilustrar cómo determinar esta expresión consideremos los siguientes ejemplos:

Algoritmo 1

1.	read(a, b, c) _____	1
2.	x = a + b _____	1
3.	y = a + c _____	1
4.	z = b * c _____	1
5.	w = x / y - z _____	1
6.	write(a, b, c, w) _____	1
		6
Contador de frecuencias =		6

En este, el **algoritmo 1**, cada instrucción se ejecuta sólo una vez. El total de veces que se ejecutan todas las instrucciones es 6. Este valor es el contador de frecuencias para el algoritmo 1.

Algoritmo 2

1.	read(n) _____	1
2.	s = 0 _____	1
3.	i = 1 _____	1
4.	while (i <= n) do _____	n + 1
5.	s = s + 1 _____	n
6.	i = i + 1 _____	n
7.	end(while) _____	n
8.	write(n, s) _____	1
		4n + 5
Contador de frecuencias =		4n + 5

En este, el **algoritmo 2**, las instrucciones 1, 2, 3 y 8 se ejecutan sólo una vez, mientras que las instrucciones 5, 6 y 7 se ejecutan **n** veces cada una, ya que pertenecen a un ciclo, cuya variable controladora del ciclo se inicia en uno, tiene un valor final de **n** y se incrementa de a uno, la instrucción 4 se ejecuta una vez más ya que cuando **i** sea mayor que **n** de todas formas hace la pregunta de si la **i** es menor o igual que **n**. Por tanto, el número de veces que se ejecutan las

instrucciones del algoritmo es $4n + 5$. Esta expresión es el contador de frecuencias para el algoritmo 2.

Algoritmo 3

1.	read(n, m)	_____	1
2.	s = 0	_____	1
3.	i = 1	_____	1
4.	while (i <= n) do	_____	n + 1
5.	t = 0	_____	n
6.	j = 1	_____	n
7.	while (j <= m) do	_____	n.(m + 1)
8.	t = t + 1	_____	n.m
9.	j = j + 1	_____	n.m
10.	end(while)	_____	n.m
11.	s = s + t	_____	n
12.	i = i + 1	_____	n
13.	end(while)	_____	n
14.	write(n, m, s)	_____	1

Contador de frecuencias = $4n.m + 7n + 5$

En este, el **algoritmo 3**, las instrucciones 1, 2, 3 y 14 se ejecutan sólo una vez, mientras que las instrucciones de la 4 a la 13 se ejecutan **n** veces cada una, ya que pertenecen a un ciclo, cuya variable controladora del ciclo se inicia en uno, tiene un valor final de **n** y se incrementa de a uno, la instrucción 4 se ejecuta una vez más ya que cuando **i** sea mayor que **n** de todas formas hace la pregunta de si la **i** es menor o igual que **n**. Las instrucciones 7 a 10 se ejecutan **m** veces cada una, pero como se hallan dentro de un ciclo en el cual cada una de las instrucciones se ejecuta **n** veces, el valor **m** se ejecuta **n** veces, es decir, **m.n** veces. La instrucción 7 se ejecuta una vez más, por aquello de que la condición se evalúa siempre. Por tanto, el número de veces que se ejecutan las instrucciones del algoritmo es $4n.m + 7n + 5$. Esta expresión es el contador de frecuencias para el algoritmo 2.

Algoritmo 4

1.	read(n)	_____	1
2.	s = 0	_____	1
3.	i = 1	_____	1
4.	while (i <= n) do	_____	n + 1
5.	t = 0	_____	n
6.	j = 1	_____	n
7.	while (j <= n) do	_____	n.(n + 1)
8.	t = t + 1	_____	n.n
9.	j = j + 1	_____	n.n
10.	end(while)	_____	n.n
11.	s = s + t	_____	n
12.	i = i + 1	_____	n
13.	end(while)	_____	n
14.	write(n, s)	_____	1

Contador de frecuencias = $4n^2 + 7n + 5$

El algoritmo 4 es similar al 3, la única diferencia es que el ciclo interno, instrucciones de la 7 a la 10, se ejecutan **n** veces en vez de **m**.

Algoritmo 5

```
1.  read(n, m) _____ 1
2.  s = 0 _____ 1
3.  i = 1 _____ 1
4.  while (i <= n) do _____ n + 1
5.      s = s + 1 _____ n
6.      i = i + 1 _____ n
7.  end(while) _____ n
8.  write(n, s) _____ 1
9.  s = 0 _____ 1
10. i = 1 _____ 1
11. while (i <= m) do _____ m + 1
12.     t = t + 1 _____ m
13.     i = i + 1 _____ m
14. end(while) _____ m
15. write(m, s) _____ 1
```

$$\text{Contador de frecuencias} = 4n + 4m + 9$$

Veamos ahora cómo obtener los órdenes de magnitud para cada uno de los algoritmos dados.

1.3.2 Orden de magnitud: es el concepto que define la eficiencia del algoritmo en cuanto a tiempo de ejecución. Se obtiene a partir del contador de frecuencias: se eliminan los coeficientes, las constantes y los términos negativos, de los términos resultantes: si son dependientes entre sí, se elige el mayor de ellos y éste será el orden de magnitud de dicho algoritmo, de lo contrario el orden de magnitud será la suma de los términos que quedaron. Si el contador de frecuencias es una constante, como en el caso del algoritmo 1, el orden de magnitud es $O(1)$.

Teniendo presente esto, los órdenes de magnitud de los algoritmos dados son:

Para el algoritmo 1: Orden de magnitud $O(1)$.

Para el algoritmo 2: Orden de magnitud $O(n)$.

Para el algoritmo 3: Orden de magnitud $O(n.m)$.

Para el algoritmo 4: Orden de magnitud $O(n^2)$.

Para el algoritmo 5: Orden de magnitud $O(n+m)$.

Es bueno aclarar lo siguiente: en los ejemplos, el valor de **n** es un dato que se lee dentro del programa. En el caso más amplio **n** podrá ser el número de registros que haya que procesar en un archivo, el número de datos que haya que producir como salida, o quizá otro parámetro diferente. Es decir, hay que poner atención en el análisis que se haga del algoritmo, de cuál es el parámetro que tomamos como **n**.

En el ambiente de sistemas los órdenes de magnitud más frecuentes son:

ORDEN DE MAGNITUD	REPRESENTACION
Constante	$O(1)$
Lineal	$O(n)$
Cuadrático	$O(n^2)$
Cúbico	$O(n^3)$
Logarítmico	$O(\log_2(n))$
Semilogarítmico	$O(n\log_2(n))$
Exponencial	$O(2^n)$

Si quisiéramos clasificar estos órdenes de magnitud en forma ascendente en cuanto a eficiencia tendríamos: los algoritmos más eficientes son aquellos con orden de magnitud constante, luego los de orden de magnitud logarítmico, a continuación los de orden de magnitud lineal, luego los semilogarítmicos, luego los cuadráticos, luego los cúbicos y por último los algoritmos con orden de magnitud exponencial.

Hasta aquí hemos visto algoritmos cuyo orden de magnitud es constante, lineal y cuadrático. No es difícil imaginar un algoritmo cuyo orden de magnitud sea cúbico.

Pasemos a tratar algoritmos con orden de magnitud logarítmico.

1.4 Orden de magnitud logarítmico: Empecemos definiendo lo que es un logaritmo. La definición clásica de logaritmo es: logaritmo de un número x es el exponente al cual hay que elevar otro número llamado base para obtener x .

Con fines didácticos, presentamos otra definición de logaritmo: logaritmo de un número x es el número de veces que hay que dividir x , por otro número llamado base, para obtener como cociente uno (1).

Por ejemplo, si tenemos el número 100 y queremos hallar su logaritmo en base 10, habrá que dividirlo por 10 sucesivamente hasta obtener como cociente uno (1).

$$\begin{array}{r} 100 \big| 10 \\ 0 \quad 10 \big| 10 \\ 0 \quad \quad 1 \end{array}$$

hubo que dividir el 100 dos veces por 10 para obtener cociente 1, por tanto el logaritmo en base 10 de 100 es **2**.

Según la definición tradicional: $10^2 = 100 = 10 \cdot 10$

Es decir, **2** es el exponente al cual hay que elevar 10, la base, para obtener 100.

Si queremos hallar el logaritmo en base 2 de 16 habrá que dividir 16 sucesivamente por 2 hasta obtener un cociente de 1. Veamos:

$$\begin{array}{r} 16 \big| 2 \\ 8 \quad 2 \big| 2 \\ 4 \quad \quad 2 \big| 2 \\ 2 \quad \quad \quad 2 \big| 2 \\ 1 \end{array}$$

es decir, hubo que dividir el 16, cuatro (4) veces por dos (2) para obtener un cociente de uno (1), por tanto el logaritmo en base 2 de 16 es **4**.

Según la definición tradicional: $2^4 = 16 = 2 \cdot 2 \cdot 2 \cdot 2$

Es decir, **4** es el exponente al cual hay que elevar 2, la base, para obtener 16.

Consideremos ahora, algoritmos sencillos cuyo orden de magnitud es logarítmico.

Algoritmo 6

```
1.  n = 32 _____ 1
2.  s = 0 _____ 1
3.  i = 32 _____ 1
4.  while (i > 1) do _____  $\log_2 n + 1$ 
5.      s = s + 1 _____  $\log_2 n$ 
6.      i = i / 2 _____  $\log_2 n$ 
7.  end(while) _____  $\log_2 n$ 
8.  write(n, s) _____ 1
```

Contador de frecuencias = $4\log_2 n + 5$

En este, el algoritmo 6, la variable controladora del ciclo es **i**, y dentro del ciclo, **i** se divide por dos (2). Si hacemos un seguimiento detallado tendremos:

La primer vez que se ejecuta el ciclo la variable **i** sale valiendo 16.

La segunda vez que se ejecuta el ciclo la variable **i** sale valiendo 8.

La tercera vez que se ejecuta el ciclo la variable **i** sale valiendo 4.

La cuarta vez que se ejecuta el ciclo la variable **i** sale valiendo 2.

La quinta vez que se ejecuta el ciclo la variable **i** sale valiendo 1.

y termina el ciclo. Es decir, las instrucciones del ciclo se ejecutan 5 veces.

Cinco (5) es el logaritmo en base dos (2) de **n** ($n == 32$), por consiguiente, cada instrucción del ciclo se ejecuta un número de veces igual al logaritmo en base dos de **n**. El contador de frecuencias y el orden de magnitud de dicho algoritmo se presentan a continuación.

Eliminando constantes y coeficientes el orden de magnitud es $O(\log_2 n)$

Consideremos este nuevo algoritmo

Algoritmo 7

```
1.  n = 32 _____ 1
2.  s = 0 _____ 1
3.  i = 1 _____ 1
4.  while (i < n) do _____  $\log_2 n + 1$ 
5.      s = s + 1 _____  $\log_2 n$ 
6.      i = i * 2 _____  $\log_2 n$ 
7.  end(while) _____  $\log_2 n$ 
8.  write(n, s) _____ 1
```

Contador de frecuencias = $4\log_2 n + 5$

En este algoritmo la variable controladora del ciclo **i** se multiplica por dos dentro del ciclo.

Haciendo un seguimiento tendremos:

En la primera pasada la variable **i** sale valiendo 2.

En la segunda pasada la variable **i** sale valiendo 4.

En la tercera pasada la variable **i** sale valiendo 8.

En la cuarta pasada la variable **i** sale valiendo 16.

En la quinta pasada la variable **i** sale valiendo 32 y el ciclo se termina.

Las instrucciones del ciclo se ejecutan 5 veces, por tanto el orden de magnitud del algoritmo es logarítmico en base dos (2).

En general, para determinar si un ciclo tiene orden de magnitud lineal o logarítmico, debemos fijarnos cómo se está modificando la variable controladora del ciclo: si ésta se modifica mediante sumas o restas el orden de magnitud del ciclo es lineal, si se modifica con multiplicaciones o divisiones el orden de magnitud del ciclo es logarítmico.

Consideremos ahora el siguiente algoritmo:

Algoritmo 8

```

1.  n = 81 _____ 1
2.  s = 0 _____ 1
3.  i = 81 _____ 1
4.  while (i > 1) do _____  $\log_3 n + 1$ 
5.      s = s + 1 _____  $\log_3 n$ 
6.      i = i / 3 _____  $\log_3 n$ 
7.  end(while) _____  $\log_3 n$ 
8.  write(n, s) _____ 1

```

Contador de frecuencias = $4\log_3 n + 5$

Aquí, la variable controladora del ciclo *i* se divide por tres dentro del ciclo.

Haciéndole seguimiento tendremos:

En la primera pasada la variable *i* sale valiendo 27.

En la segunda pasada la variable *i* sale valiendo 9.

En la tercera pasada la variable *i* sale valiendo 3.

En la cuarta pasada la variable *i* sale valiendo 1 y termina el ciclo.

Las instrucciones del ciclo se ejecutaron 4 veces. Cuatro es el logaritmo en base 3 de 81. Por consiguiente el orden de magnitud de dicho algoritmo es logarítmico en base 3.

En general, si tenemos un algoritmo:

Algoritmo 9

```

1.  n = 81 _____ 1
2.  s = 0 _____ 1
3.  i = 81 _____ 1
4.  while (i > 1) do _____  $\log_x n + 1$ 
5.      s = s + 1 _____  $\log_x n$ 
6.      i = i / x _____  $\log_x n$ 
7.  end(while) _____  $\log_x n$ 
8.  write(n, s) _____ 1

```

Contador de frecuencias = $4\log_x n + 5$

Aquí, la variable controladora del ciclo se divide por *x*.

El número de veces que se ejecutan las instrucciones del ciclo es logaritmo en base *x* de *n*, por tanto el orden de magnitud es logarítmico en base *x*.

Para obtener algoritmos con orden de magnitud semilogarítmico basta con que un ciclo logarítmico se ubique dentro de un ciclo con orden de magnitud **O(n)** o viceversa. Por ejemplo:

Algoritmo 10

```

1.  read(n) _____ 1
2.  s = 0 _____ 1

```

3.	i = 1	_____	1
4.	while (i <= n)	do _____	n + 1
5.	t = 0	_____	n
6.	j = n	_____	n
7.	while (j > 1)	do _____	n.(log ₂ n + 1)
8.	t = t + 1	_____	n.log ₂ n
9.	j = j / 2	_____	n.log ₂ n
10.	end(while)	_____	n.log ₂ n
11.	write(t)	_____	n
12.	s = s + t	_____	n
13.	i = i + 1	_____	n
14.	end(while)	_____	n
15.	write(n, s)	_____	1

Contador de frecuencias = $8n + 4n \cdot \log_2 n + 5$

y el orden de magnitud es semilogarítmico.

1.5 Otros: Digamos que la idea es elaborar algoritmos lo más eficiente posible. Consideremos la siguiente tarea: Elaborar algoritmo que lea un entero positivo **n** y que determine la suma de los enteros desde uno hasta **n**. Una solución es la siguiente:

```

read(n)
s = 0
i = 1
while (i <= n) do
    s = s + i
    i = i + 1
end(while)
write(n, s)

```

El anterior algoritmo lee un dato entero **n** (**n > 0**), calcula e imprime la suma de los enteros desde 1 hasta **n**, y el orden de magnitud de él es **O(n)**. Fíjese que es exactamente el mismo algoritmo 2 presentado anteriormente.

Matemáticamente hablando la suma de los enteros desde 1 hasta **n** es:

$$S = \sum_{i=1}^{i=n} i$$

sumatoria, que según los matemáticos es: $\frac{n * (n + 1)}{2}$

Conociendo esta fórmula, podemos elaborar otro algoritmo que ejecute exactamente la misma tarea. Veamos:

```

read(n)
s = n*(n+1)/2
write(n, s)

```

el cual, tiene orden de magnitud **O(1)**.

Es decir, tenemos dos algoritmos completamente diferentes que ejecutan exactamente la misma tarea, uno con orden de magnitud **O(n)** y el otro con orden de magnitud **O(1)**.

Lo anterior no significa que toda tarea podrá ser escrita con algoritmos $O(1)$, no, sino que de acuerdo a los conocimientos que se tengan o a la creatividad de cada cual, se podrán elaborar algoritmos más eficientes.

Este ejemplo, el cual es válido, desde el punto de vista de desarrollo de algoritmos, es aplicable también a situaciones de la vida real. Seguro que alguna vez usted tuvo algún problema, y encontró una solución, y se salió del problema, aplicando la solución que encontró, y sin embargo, al tiempo, dos o tres meses después, pensando en lo que había hecho, encontró que pudo haber tomado una mejor determinación y que le habría ido mejor.

La enseñanza es que cuando se nos presente un problema debemos encontrar como mínimo dos soluciones y luego evaluar cuál es la más apropiada. Ya tenemos herramientas para determinar la solución más eficiente.

Como un ejemplo adicional consideremos el siguiente algoritmo:

Algoritmo 11a

1	read(n)	_____	1
2	s = 0	_____	1
3	for (i = 1; i <= n; i++) do	_____	n + 1
4	t = 0	_____	n
5	for (J = 1; J <= i; J++) do		
6	t = t + 1		
7	end(for)		
8	s = s + t	_____	n
9	end(for)	_____	n
10	write(n, s)	_____	1

Vamos a determinar el contador de frecuencias y el orden de magnitud de dicho algoritmo.

Las instrucciones 1, 2 y 10 se ejecutan sólo una vez.

Las instrucciones 3, 4, 8 y 9 se ejecutan n veces. Es importante, en este punto, decir que las instrucciones de ciclos, For y Do...While, se comportan de una manera análoga a la instrucción while.

Consideremos ahora las instrucciones 5, 6 y 7:

 Cuando la i vale 1 las instrucciones 5, 6 y 7 se ejecutan 1 vez.

 Cuando la i vale 2 las instrucciones 5, 6 y 7 se ejecutan 2 veces.

 Cuando la i vale 3 las instrucciones 5, 6 y 7 se ejecutan 3 veces.

 Cuando la i vale 4 las instrucciones 5, 6 y 7 se ejecutan 4 veces.

"	"	"	"	5	"	"	"	"	"	"	5	"
"	"	"	"	6	"	"	"	"	"	"	6	"
"	"	"	"	7	"	"	"	"	"	"	7	"

 Cuando la i vale n , las instrucciones 5, 6 y 7 se ejecutan n veces.

Es decir, el total de veces que se ejecutan las instrucciones 5, 6 y 7 es:

$$1 + 2 + 3 + \dots + n$$

o sea la sumatoria de i , con i desde 1 hasta n , que como ya habíamos visto es $n*(n+1)/2$. Por tanto el contador de frecuencias de nuestro algoritmo será:

Algoritmo 11b

1	read(n)	_____	1
2	s = 0	_____	1

3	for (i = 1; i <= n; i++) do	_____	n + 1
4	t = 0	_____	n
5	for (J = 1; J <= i; J++) do	_____	n(n+1)/2 + n
6	t = t + 1	_____	n(n+1)/2
7	end(for)	_____	n(n+1)/2
8	s = s + t	_____	n
9	end(for)	_____	n
10	write(n, s)	_____	1

Contador de frecuencias = $3n(n+1)/2 + 5n + 4$
y el orden de magnitud es $O(n^2)$.

Para terminar este módulo, consideremos algunas variaciones al algoritmo 2.

Variación 1:

Algoritmo 12

1.	read(n)	_____	1
2.	s = 0	_____	1
3.	i = 1	_____	1
4.	while (i < n) do	_____	n
5.	s = s + 1	_____	n - 1
6.	i = i + 1	_____	n - 1
7.	end(while)	_____	n - 1
8.	write(n, s)	_____	1

Contador de frecuencias = $4n + 1$
y el orden de magnitud es $O(n)$.

La única diferencia de este algoritmo con el algoritmo 2 es la condición del ciclo en la instrucción 4. En el algoritmo 2 la condición es menor o igual, mientras que aquí la condición de la instrucción 4 es sólo menor que. Esto implica que las instrucciones desde la 4 hasta la 7 se ejecutan una vez menos. Esta es la razón por la que en algunas situaciones aparecen contadores de frecuencias con términos negativos.

Variación 2

Algoritmo 13

1.	read(n)	_____	1
2.	s = 0	_____	1
3.	i = 1	_____	1
4.	while (i <= n) do	_____	n/2 + 1
5.	s = s + 1	_____	n/2
6.	i = i + 2	_____	n/2
7.	end(while)	_____	n/2
8.	write(n, s)	_____	1

Contador de frecuencias = $4n/2 + 5$
y el orden de magnitud es $O(n)$.

La diferencia de este algoritmo con el algoritmo 2 es que en la instrucción 7 del algoritmo 2 la variable controladora del ciclo se modifica incrementándola en 1, mientras que en este algoritmo la variable controladora del ciclo se incrementa en 2. Esto ocasiona que las instrucciones del ciclo se ejecuten sólo la mitad de las veces.

En general, si la variable controladora del ciclo se modifica con sumas o restas, el contador de frecuencias de las instrucciones correspondientes al ciclo será el valor final de la variable

controladora del ciclo menos el valor inicial de la variable controladora del ciclo más uno, dividido por el valor con el cual se modifica la variable controladora del ciclo. El contador de frecuencias de la instrucción correspondiente al ciclo (while, for, do...while) es el mismo valor anterior más uno.

Algoritmo 14

1.	read(n)	_____	1
2.	s = 0	_____	1
3.	i = 1	_____	1
4.	while (i <= n) do	_____	n/x + 1
5.	s = s + 1	_____	n/x
6.	i = i + x	_____	n/x
7.	end(while)	_____	n/x
8.	write(n, s)	_____	1

Contador de frecuencias = $4n/x + 5$
y el orden de magnitud es $O(n)$.

Por último, digamos que cuando dos algoritmos que ejecutan una misma tarea tienen el mismo orden de magnitud, será más eficiente aquel cuyo contador de frecuencias sea menor. En muchas situaciones, un algoritmo es más eficiente que otro, dependiendo del valor de **n**. Por ejemplo, si se tienen dos algoritmos A1 y A2 con orden de magnitud $O(n)$ y sus contadores de frecuencias son $10n + 50$ y $5n + 100$ respectivamente, el algoritmo A1 es más eficiente para valores de **n** entre 1 y 10, para valores de **n** mayores que 10 es más eficiente el algoritmo A2.

EJERCICIOS PROPUESTOS

Determine contador de frecuencias y orden de magnitud para los siguientes algoritmos:

1. **void** p1(entero n)
 s = 0
 for (i = 1; i <= n; i++) do
 for (J = 1; J <= i; J++) do
 for (k = 1; k <= J; k++) do
 s = s + 1
 end(for)
 end(for)
 end(for)
 fin(p1)

2. **void** p2(entero n)
 s = 0
 i = 1
 j = 1
 while ((i <= n) and (j <= n)) do
 s = s + 1
 i = i + 1
 if i > n and j < n then
 i = 1
 j = j + 1
 end(if)
 end(while)

```
        write(s)
    fin(p2)
```

```
3.  void p3(vector V, entero n, real x)
    i = 1
    j = n
    do
        k = (i+j)/2
        if V[k] <= x then
            i = k + 1
        else
            j = k - 1
        end(if)
    while (i <= j)
fin(void).
```

```
4.  void p4(Vector V, entero n)
    i = 1
    while i <= n do
        s = 0
        j = i + 1
        while ((j <= n) and (j <= i + V[i])) do
            s = s + V[i]
            j = j + 1
        end(while)
        write(s)
        i = j
    end(while)
fin(p4)
```

```
5.  void p5()
    read(n)
    s = 0
    i = 2
    while (i <= n) do
        s = s + 1
        i = i * i
    end(while)
    write(n, s)
fin(p5)
```

```
6.  void p6()
    read(n)
    s = 0
    i = 3
    while (i <= n) do
        s = s + 1
        i = i * i
    end(while)
    write(n, s)
fin(p6)
```

MODULO 2

EVALUACIÓN DE ALGORITMOS (ORDENAMIENTO Y BÚSQUEDA)

Introducción

Dos de las principales operaciones que se presentan en el desarrollo de aplicaciones por computador son ordenar los datos con los cuales se trabaja con el fin de que los procesos de búsqueda sean lo más eficiente posible. Tener los datos ordenados (ascendente o descendientemente) hace que el proceso de buscar algún dato en el conjunto de datos sea una tarea fácil y eficiente: imagínese un diccionario o un directorio telefónico con los datos en desorden, buscar el significado de una palabra o el teléfono correspondiente a una persona sería una labor supremamente dispendiosa. Trataremos en este módulo algunos métodos para ordenar colecciones de datos y la forma de efectuar una búsqueda en ese conjunto de datos, además de determinar el orden de magnitud de cada uno de los métodos presentados.

Objetivos

1. Aprender el método de ordenamiento de datos denominado “selección”.
2. Aprender el método de ordenamiento de datos denominado “inserción”.
3. Aprender el método de ordenamiento de datos denominado “burbuja”.
4. Aprender el método de ordenamiento de datos denominado “burbuja mejorado”.
5. Aprender el método de búsqueda denominado “búsqueda secuencial”.
6. Aprender el método de búsqueda denominado “búsqueda binaria”.

Preguntas básicas

1. Cómo se enuncia el método de ordenamiento “selección”?
2. Cuál es el orden de magnitud del método de ordenamiento “selección”?
3. Cuál es el orden de magnitud del método de ordenamiento “inserción”?
4. Cuál es el orden de magnitud del método de ordenamiento “burbuja”?
5. Cuál es el orden de magnitud del método de ordenamiento “burbuja mejorado”?
6. Cuál es la ventaja del método “burbuja mejorado” frente al método “burbuja”?
7. Cuál es el orden de magnitud del método de búsqueda “búsqueda secuencial”?
8. Cuál es el orden de magnitud del método de búsqueda “búsqueda binaria”?
9. Qué se requiere para utilizar el método búsqueda binaria?

2.1 Generalidades: en el desarrollo de este módulo consideraremos que el estudiante está familiarizado con el paradigma objetual. Asumiremos que se tiene una clase Vector cuyos datos privados son V , un arreglo de una dimensión, y n , el tamaño de dicho arreglo. Además, se dispone de un método privado llamado `intercambia(i, k)`, el cual, como su nombre lo dice, intercambia el dato que está en la posición i de V con el dato que se halla en la posición k de V . Se asume también, en los algoritmos aquí presentados que los subíndices del arreglo van desde 1 hasta n , y no, desde 0 hasta $n-1$, como lo hacen las diferentes versiones de C y java.

2.2 Ordenamiento por selección: este es el método de ordenamiento más sencillo que hay. Se puede enunciar: “de los datos que faltan por ordenar, determinar el menor de ellos y colocarlo de primero en ese conjunto de datos”. Con base en este enunciado procedemos a desarrollar nuestro algoritmo. Planteamos un ciclo externo (el que comienza en la instrucción 3) en el cual se define a partir de cuál posición faltan datos por ordenar, la variable controladora de dicho ciclo, la variable i , indica que faltan por ordenar los datos que se hallan a partir de dicha posición. Inicialmente i vale 1 indicando que faltan por ordenar todos los datos que hay desde la posición 1 hasta la posición n . Utilizamos una variable k , la cual indicará la posición en la cual se halla el menor dato entre todos los datos que hay desde i hasta n ; inicialmente a k se le asigna la posición i (instrucción 4) indicando que el menor dato se halla en la posición i del vector V . En el ciclo interno (instrucciones 5 a 9) se determina la posición en la cual se halla el menor dato entre las posiciones i y n . Utilizamos una variable j con la cual se recorren todos los datos desde $i+1$ hasta n , comparando el dato de la posición k con el dato de la posición j . En caso de que el dato de la

posición j sea menor que el dato que se halla en la posición k , se actualiza el valor de k , logrando de esta forma que en la variable k siempre esté la posición en la cual se halla el menor dato. Al terminar este ciclo se procede a intercambiar el dato de la posición i (el primero del conjunto de datos que faltan por ordenar) con el dato de la posición k (el menor del conjunto de datos que faltan por ordenar). En caso de que el valor de k no se hubiera modificado en el ciclo interno (instrucciones 5 a 9) el proceso de intercambio deja el vector tal cual estaba. El ciclo externo (instrucciones 3 a 11) se ejecuta hasta que i sea igual a n . Cuando i sea igual a n significa que sólo falta un dato por ordenar, lo que implica que se termina el proceso, puesto que un solo dato está ordenado por sí mismo.

```

1. void selección()
2.     entero i, j, k
3.     for (i = 1; i < n; i++) do
4.         k = i
5.         for (j = i+1; j <= n; j++) do
6.             if V[j] < V[k] then
7.                 k := j
8.             end(if)
9.         end(for)
10.        intercambia(i, k)
11.    end(for)
12. end(selección)

```

Analicemos ahora la eficiencia de este algoritmo. El contador de frecuencias para las instrucciones 1 y 2 es 1. La instrucción 3 (ciclo con la variable i desde 1 hasta $n-1$, incrementándose de 1 en 1) se ejecuta n veces, por tanto las instrucciones 4 y 10 se ejecutan $n-1$ veces cada una de ellas. En el ciclo interno (instrucciones 5 a 9) la variable j varía desde $i+1$ hasta n : cuando la i vale 1 dichas instrucciones se ejecutan $n-1$ veces, cuando la i vale 2 se ejecutan $n-2$ veces, cuando la i vale 3 se ejecutan $n-3$ veces, y así sucesivamente hasta que la i valga $n-1$: cuando la i vale $n-1$ las instrucciones del ciclo interno se ejecutan una vez. Es decir, el número de veces que se ejecutan las instrucciones del ciclo interno es una sumatoria de todos los enteros desde 1 hasta $n-1$, cuyo total es $n(n-1)/2$. El contador de frecuencias y el orden de magnitud de nuestro algoritmo es:

1. void selección()	_____	1
2. entero i, j, k	_____	1
3. for (i = 1; i < n; i++) do	_____	n
4. k = i	_____	$n-1$
5. for (j = i+1; j <= n; j++) do	_____	$n(n-1)/2 + (n-1)$
6. if V[j] < V[k] then	_____	$n(n-1)/2$
7. k := j	_____	$n(n-1)/2$
8. end(if)	_____	$n(n-1)/2$
9. end(for)	_____	$n(n-1)/2$
10. intercambia(i, k)	_____	$n-1$
11. end(for)	_____	$n-1$
12. end(selección)	_____	1

Contador de frecuencias = $5n(n-1)/2 + 5n - 1$
 y el orden de magnitud es $O(n^2)$

Es bueno hacer notar que la instrucción 7, la cual es una instrucción que se ejecuta sólo cuando la condición que se prueba en la instrucción 6 es verdadera, estamos considerando que siempre que entre al ciclo se ejecuta, es decir, estamos considerando que la condición siempre es verdadera.

En general, cuando se tenga una instrucción de decisión dentro de un ciclo, siempre se debe considerar el peor de los casos.

2.2 Ordenamiento por inserción: la idea básica de este método de ordenamiento es tomar un dato que se encuentre en una posición i del arreglo y buscar dónde insertarlo entre los $i-1$ datos anteriores, los cuales ya se hayan ordenados. A medida que se efectúa el proceso de búsqueda se van moviendo, una posición hacia la derecha, los datos del arreglo que están en las $i-1$ posiciones anteriores, que sean mayores que el dato de la posición i . Teniendo definido el método nuestro algoritmo es el que se presenta a continuación: el ciclo que se plantea en la instrucción 3 recorre el vector desde la posición 2 hasta la posición n ; en la instrucción 4 guardamos el dato de la posición i en una variable auxiliar que llamamos d ; en la instrucción 5 se inicializa una variable j en $i-1$, con la cual efectuamos el proceso de buscar dónde insertar, comparando el dato de la posición j con el dato d , si el dato de la posición j es mayor que el dato d , desplazamos el dato de la posición j hacia la posición $j+1$ y continuamos retrocediendo con j . Este proceso (ciclo de las instrucciones 6 a 9) se efectúa hasta que j sea igual a cero o el dato de la posición j sea menor o igual que el dato d . Al terminar este ciclo, se almacena el dato d en la posición $j+1$.

```
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)
```

Veamos cuál es el orden de magnitud de este algoritmo.

El ciclo de las instrucciones 3 a 11 se ejecuta $n-1$ veces.

El ciclo interno (instrucciones 6 a 9), el número de veces que se ejecuta, depende de si el valor de $V[j]$ es mayor que d o no. El máximo número de veces que se ejecuta es cuando la variable j alcanza el valor de 0: cuando la i valga 2 se ejecuta una vez, cuando la i valga 3 se ejecuta 2 veces, cuando la i valga 4 se ejecuta 3 veces y así sucesivamente, es decir, el total de veces que se ejecuta es la sumatoria de los enteros desde 1 hasta $n - 1$, cuyo resultado es $n(n - 1)/2$. Observe que estamos considerando el peor de los casos. Si por alguna casualidad los datos llegan ordenados el ciclo interno no se ejecuta ni una sola vez y el contador de frecuencias del ciclo interno será cero, sólo la instrucción 6 se ejecutará $n - 1$ veces. El peor caso, también es una casualidad: se presenta sólo si los datos llegan ordenados descendientemente. El caso promedio es que al ciclo interno siempre entre, pero que no lo ejecute en su totalidad todas las veces.

El contador de frecuencias y el orden de magnitud de nuestro algoritmo se presenta a continuación.

1. void inserción()	_____	1
2. entero i, j, d	_____	1
3. for (i = 2; i <= n; i++) do	_____	n
4. d = V[i]	_____	$n - 1$
5. j = i - 1	_____	$n - 1$

6.	while ((j > 0) and (d < V[j])) do	_____	$n(n-1)/2 + (n-1)$
7.	V[j+1] = V[j]	_____	$n(n-1)/2$
8.	j = j - 1	_____	$n(n-1)/2$
9.	end(while)	_____	$n(n-1)/2$
10.	V[j+1] = d	_____	$n-1$
11.	end(for)	_____	$n-1$
12.	end(insertión)	_____	1

Contador de frecuencias = $4n(n-1)/2 + 5n - 2$
y el orden de magnitud es $O(n^2)$.

2.3 Ordenamiento por burbuja: la idea de este método es que en cada pasada que se efectúe sobre un conjunto de datos se lleve el mayor dato a la última posición. Esto implica que en la primera pasada se toman todos los datos y el mayor dato quedará de último; en la segunda pasada se toman todos los datos, menos el último que ya está en su posición, y se repite el proceso de determinar el mayor dato y ubicarlo de último; en la tercera pasada se toman todos los datos, menos los dos últimos que ya están ordenados y se repite nuevamente el proceso de determinar el mayor dato y ubicarlo de último. Lo interesante, es la forma como se determina el mayor dato y se ubica de último: se recorren todos los datos comparando siempre dos datos consecutivos, digamos el de la posición j con el de la posición j+1, si el dato de la posición j es mayor que el dato de la posición j+1 se intercambian dichos datos, de lo contrario se dejan como estaban. Cualquiera que hubiera sido la situación se continúa comparando los dos siguientes datos consecutivos. La primer vez se hace este proceso hasta la posición n del arreglo, la segunda vez se hace hasta la posición n-1, la tercera vez hasta la posición n-2, y así sucesivamente hasta que sólo quede un dato, es decir, que n-i sea igual a 1. El algoritmo para efectuar este proceso se presenta a continuación.

```

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)

```

Veamos ahora cuál es el orden de magnitud de dicho algoritmo.

De acuerdo al proceso definido, se deben realizar una serie de pasadas. Si el vector tiene 8 datos se deben hacer 7 pasadas, si el vector tiene 100 datos se deben efectuar 99 pasadas. En general, para n datos se deben efectuar n - 1 pasadas. El ciclo de la instrucción 3 se utiliza para contar las pasadas, por tanto, dicha instrucción se ejecuta n veces.

El total de comparaciones a efectuar en cada pasada va disminuyendo a medida que se avanza en las pasadas: en la primera pasada se ejecutan n - 1 comparaciones, en la segunda pasada se efectúan n - 2 comparaciones, en la tercera pasada se efectúan n - 3 comparaciones, y así sucesivamente hasta que en la última pasada se efectúa sólo una comparación. Como son n - 1 pasadas el total de comparaciones será la sumatoria de los enteros desde 1 hasta n-1, cuyo valor es $n(n-1)/2$.

El contador de frecuencias y el orden de magnitud de dicho algoritmo se presenta a continuación.

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

Contador de frecuencias = $5n(n-1)/2 + 3n + 1$
y el orden de magnitud es $O(n^2)$.

2.4 Ordenamiento por burbuja mejorado: la idea del método burbuja mejorado es disminuir el número de pasadas. Al aplicar el método de burbuja puede suceder que después de alguna pasada los datos queden ordenados y que, por consiguiente no haya necesidad de efectuar más pasadas. La forma de detectar que los datos ya han quedado ordenados es que en el ciclo interno (instrucciones 4 a 8) no se efectúe ningún intercambio de datos en alguna pasada. Si se hace una pasada y en esa pasada no se hizo ningún intercambio entonces los datos ya están ordenados. Con el fin de que nuestro algoritmo detecte esta situación definimos una nueva variable, la cual llamamos sw. Nuestra variable sw se inicializa en 0 antes de ejecutar el ciclo de comparaciones correspondiente a cada pasada (instrucciones 5 a 10 del algoritmo a continuación) y dentro del ciclo si se efectúa algún intercambio colocamos el valor de sw en 1. Si al salir del ciclo (instrucción 11 del algoritmo a continuación) la variable sw está en 0 quiere decir que no se hizo ningún intercambio y que por lo tanto los datos están ordenados y termina la tarea. El algoritmo para el método burbuja mejorado se presenta a continuación.

1. void burbujaMejorado()	1
2. entero i, j, sw	1
3. for (i = 1; i < n; i++) do	n
4. sw = 0	n-1
5. for (j = 1; j <= n - i; j++) do	$n(n-1)/2 + (n-1)$
6. if (V[j] > V[j+1]) then	$n(n-1)/2$
7. intercambia(j, j+1)	$n(n-1)/2$
8. sw = 1	$n(n-1)/2$
9. end(if)	$n(n-1)/2$
10. end(for)	$n(n-1)/2$
11. if (sw == 0) then return	n-1
12. end(for)	n-1
13. end(burbujaMejorado)	1

Contador de frecuencias = $6n(n-1)/2 + 5n - 1$
y el orden de magnitud es $O(n^2)$.

2.5 Búsqueda secuencial: como es bien sabido los computadores son máquinas cuyo objetivo es almacenar grandes volúmenes de información para trabajar sobre ella de una manera eficiente y segura. Por consiguiente, una de las tareas que con mayor frecuencia se presenta es buscar

algún dato. Consideremos el caso de un vector en el cual tenemos los datos ordenados ascendentemente:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
V	2	4	7	9	11	15	17	28	31	36	43	50	58	62	69

Si nos interesa buscar el dato $X = 85$ y decir en cuál posición del vector se encuentra hay dos formas de hacerlo.

Una, es hacer una búsqueda secuencial. Ésta consiste en comenzar a partir del primer elemento e ir comparando el contenido de esa posición con el dato **X (X=85)**, el proceso terminará cuando se encuentre el dato o cuando se haya recorrido todo el vector y no se encuentre el dato. Un algoritmo que haga este proceso es el siguiente:

Sea n el número de elementos del vector y x el dato a buscar.

1. $i = 1$
2. while $((i \leq n) \text{ and } (V[i] \neq x))$ do
3. $i = i + 1$
4. end(while)
5. if $(i \leq n)$ then return i
6. else write("el dato ", x , "no existe")

En dicho algoritmo tenemos un ciclo (instrucciones 2 a 4), el cual, en el peor de los casos se ejecuta n veces, es decir, cuando el dato x no esté en el vector. Por tanto el orden de magnitud de ese algoritmo es **$O(n)$** .

En general, cuando se efectúa una búsqueda, el peor caso es no encontrar lo que se está buscando. En nuestro ejemplo hubo que hacer 15 comparaciones para poder decir que el 85 no está en el vector. Si el vector hubiera tenido un millón de datos y hubiéramos tenido que buscar un dato que no está, hubiéramos tenido que haber hecho un millón de comparaciones para poder decir que el dato buscado no se encontró.

2.6 Búsqueda binaria: Ahora, si aprovechamos la característica de que los datos en el vector están ordenados podemos plantear otro método de búsqueda:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
V	2	4	7	9	11	15	17	28	31	36	43	50	58	62	69
								M							

Comparamos el dato x con el dato del elemento de la mitad del vector, (llamémoslo **M**). Pueden suceder tres casos: **$V[M] == x$** ; **$V[M] > x$** ; **$V[M] < x$**

Si **$V[M] == x$** se ha terminado la búsqueda.

Si **$V[M] > x$** significa que si el dato x se encuentra en el vector **V**, estará en la primera mitad del vector.

Si **$V[M] < x$** significa que si el dato x se encuentra en el vector **V**, estará en la segunda mitad del vector.

En nuestro ejemplo, para seguir buscando el dato $x = 85$ lo haremos desde la posición 9 hasta la posición 15. Es decir, con una comparación hemos eliminado la mitad de los datos sobre los cuales hay que efectuar la búsqueda. En otras palabras, hemos dividido la muestra por dos (2).

El conjunto de datos sobre el cual hay que efectuar la búsqueda queda así:

9	10	11	12	13	14	15
31	36	43	50	58	62	69
			M			

De la mitad que quedó escogemos nuevamente el elemento de la mitad y repetimos la comparación planteada anteriormente. Con esta comparación eliminamos nuevamente la mitad de los datos que nos quedaban, es decir, dividimos nuevamente por dos. Si continuamos este proceso, con cuatro comparaciones podremos afirmar que el dato $x = 85$ no está en el vector.

Hemos reducido el número de comparaciones de 15 a 4.

Cuatro (4) es el logaritmo en base 2 de 15, aproximándolo por encima.

Si **N** fuera 65000, con 16 comparaciones podremos decir que un dato no se encuentra en un conjunto de 65000 datos. Como puede observarse, la ganancia en cuanto al número de comparaciones es apreciable: de 65000 a 16. Un algoritmo que use esta técnica se presenta a continuación.

1. primero = 1
2. ultimo = n
3. while (primero <= ultimo) do
4. medio = (primero + ultimo) / 2
5. if (V[medio] == x) then return medio
6. if (V[medio] > x) then ultimo = medio - 1
7. else primero = medio + 1
8. end(while)
9. write("el dato", x, "no existe")

Un algoritmo de búsqueda que funcione con esta técnica tiene orden de magnitud logarítmico en base dos, $O(\log_2 n)$, ya que la muestra de datos se divide sucesivamente por dos.

2.7 Comentarios: hemos presentado 4 métodos para ordenar ascendentemente los datos de un arreglo. Los 4 métodos se implementan con algoritmos cuyo orden de magnitud es cuadrático, $O(n^2)$. El método de selección y el método de burbuja siempre tendrán ese orden de magnitud, el burbuja mejorado y el de inserción en el mejor de los casos tendrán orden de magnitud lineal, $O(n)$, sin embargo, en el caso promedio sus órdenes de magnitud serán cuadráticos aunque las instrucciones del ciclo interno no siempre se ejecuten n^2 veces. En realidad existen muchos más métodos de ordenamiento, buscando todos ellos, que esta tarea sea lo más eficiente posible. En nuestro curso veremos otros dos métodos conocidos como quick-sort y sort-merge. Dichos métodos tienen orden de magnitud semilogarítmico, $O(n \cdot \log_2 n)$ y los trataremos en el tema correspondiente a recursión. Adicionalmente existe otro método denominado heap-sort, también con orden de magnitud semilogarítmico el cual se verá en el curso de lógica y representación III en el tema correspondiente a árboles binarios.

EJERCICIOS PROPUESTOS

1. Elabore algoritmo que modifique el ordenamiento por inserción efectuando el proceso de búsqueda utilizando la búsqueda binaria.
2. Haga seguimiento detallado, paso por paso, al proceso de ordenamiento del vector mostrado a continuación usando el método de selección.

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

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

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

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

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

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

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

MODULO 3

MANEJO DINAMICO DE MEMORIA (CONCEPTOS Y CLASE NODO SIMPLE)

Introducción

Hasta ahora hemos trabajado colecciones de datos representándolas en arreglos. Dicha representación presenta algunos problemas como desperdicio de memoria o memoria insuficiente, además de que las operaciones de inserción y borrado se logran con algoritmos cuyo orden de magnitud es lineal, $O(n)$, lo cual se considera ineficiente en tareas que tienen alta frecuencia de ejecución. Veremos aquí una nueva forma para representar colecciones de datos de tal manera que no aparezcan los problemas de memoria insuficiente o desperdicio de memoria, es decir, que los programas utilicen exactamente la memoria que necesitan, y que los algoritmos para los procesos de inserción y borrado sean eficientes, es decir, que tengan orden de magnitud $O(1)$.

Objetivos

1. Aprender el concepto de lista ligada.
2. Conocer el objeto nodo simple y su manipulación.

Preguntas básicas

1. En qué consiste el manejo dinámico de la memoria?
2. Para qué sirve el campo de liga en un nodo?
3. Por qué los procesos de inserción y borrado tienen orden de magnitud lineal en el manejo estático de la memoria?

3.1 Repaso manejo estático de la memoria: Para entrar a hablar del manejo dinámico de memoria repasemos brevemente lo que es el manejo estático de la memoria, es decir, los arreglos. Tratemos el caso de un vector, según la clase definida en el curso de lógica y representación I.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
b	d	f	h	i	m									

Figura 3.1

En el vector de la figura 3.1 tenemos un conjunto de datos ordenados ascendentemente.

Las operaciones básicas que se realizan sobre ese conjunto de datos son insertar un dato y borrar un dato. Analicemos cada una de ellas.

Si se desea insertar el dato '**g**' en el vector de la figura 3.1 el proceso a seguir es:

1. Buscar en cuál posición insertarlo. (Método ***dondeInsertar(d)***)
2. Insertarlo en la posición correspondiente. (Método ***insertarEnPosicion(i,d)***)

Del primer paso se obtiene que la posición en la cual hay que insertar la '**g**' es la posición 4 del vector. El segundo paso deberá mover los datos desde la posición 4 hasta la posición 6, una posición hacia la derecha y luego asignar a la posición 4 el dato '**g**'.

Si el dato a insertar hubiera sido la letra '**a**' hubiéramos tenido que mover todos los datos del vector.

Generalizando, se considera el peor de los casos, que es cuando hay que insertar el dato al principio del vector. Hay que mover **n** datos en el vector, lo cual implica que el algoritmo de inserción tendrá orden de magnitud **$O(n)$** .

Si deseamos borrar un dato de un vector los pasos a seguir son:

1. Buscar en cuál posición se halla el dato a borrar. (Método **retornaPosición(d)**).
2. Borrar el dato del vector. (Método **borra(d)**)

En caso de que el dato a borrar fuera la letra 'f', el primer paso me retorna 3, o sea, la posición en la cual se halla la letra 'f'. El segundo paso moverá los datos desde la posición 4 hasta la 6 una posición hacia la izquierda.

Si el dato a borrar hubiera estado en la posición 1 del vector y el vector tiene **n** datos, entonces habrá que mover **n-1** datos hacia la izquierda, y el orden de magnitud de dicho algoritmo es **O(n)**.

De lo expuesto anteriormente, los algoritmos de inserción y borrado tienen orden de magnitud **O(n)**, y esto en el manejo de grandes volúmenes de información, con operaciones de alta frecuencia, como son insertar y borrar se considera ineficiente.

Por lo tanto se ha buscado una forma alterna de representación en la cual las operaciones de inserción y borrado sean eficientes, es decir, tengan orden de magnitud **O(1)**.

3.2 Concepto: Consideremos los siguientes dos vectores que llamamos **DATO Y LIGA**.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
DATO	d	h	b		m	f		i						

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
LIGA	6	8	1		┐	2		5						

Figura 3.2

En el vector **DATO** almacenamos los datos en posiciones aleatorias. Físicamente los datos se hallan en desorden. Nos interesa tenerlos ordenados lógicamente. El símbolo \perp representa nulo.

Para ello debemos conocer en cuál posición del vector se halla el primer dato. En nuestro ejemplo, el primer dato se halla en la posición 3 del vector. Utilizaremos una variable, llamémosla **primero**, cuyo valor es 3. Es decir, el hecho de que **primero** valga 3 significa: el primer dato se halla en la posición 3 del vector **DATO**.

Nos interesa saber en cuál posición se halla el siguiente dato, para ello utilizamos la posición 3 del vector **LIGA**.

El siguiente dato, que es la 'd', se halla en la posición 1 del vector **DATO**, por tanto en la posición 3 del vector **LIGA** tendremos un 1.

El hecho de que en la posición 3 del vector **LIGA** haya un 1 significa que el siguiente dato se halla en la posición 1 del vector **DATO**.

Para conocer el siguiente a la 'd' utilizamos la posición 1 del vector **LIGA**. Allí encontramos un 6, lo que significa que el siguiente dato a la 'd' se encuentra en la posición 6 del vector **DATO**.

En general, para un dato que se halle en la posición **i** del vector **DATO**, la correspondiente posición **i** del vector **LIGA** indica en cuál posición del vector **DATO** se halla el siguiente dato.

Los textos de computadores acostumbran presentar la situación descrita anteriormente, de la siguiente forma:

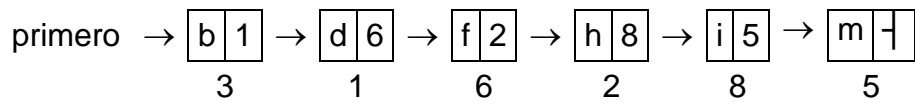


Figura 3.3

donde cada cuadrado contiene dos campos: uno para el dato y otro para la liga. Técnicamente, cada cuadrado se denomina **nodo**.

3.3 Clase nodo simple. Como nuestro objetivo es trabajar programación orientada a objetos, y para ello hacemos uso de clases con sus respectivos métodos, comencemos definiendo la clase en la cual manipulemos el nodo que acabamos de definir.

Dicha clase la denominaremos **nodoSimple**

Clase **nodoSimple**

Privado

Objeto dato

nodoSimple liga

Público

nodoSimple() // constructor

Objeto retornaDato()

nodoSimple retornaLiga()

void asignaDato(Objeto d)

void asignaLiga(nodoSimple x)

fin(clase nodoSimple)

Los algoritmos correspondientes a los métodos definidos son:

nodoSimple(objeto d) //constructor

dato = d

liga = null

fin(nodoSimple)

objeto retornaDato()

return dato

fin(retornaDato)

nodoSimple retornaLiga()

return liga

fin(retornaLiga)

void asignaDato(objeto d)

dato = d

fin(asignaDato)

void asignaLiga(nodoSimple x)

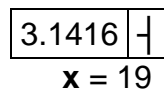
liga = x

fin(asignaLiga)

Para hacer uso de esta clase consideremos el siguiente ejemplo:

1. nodoSimple x
2. d = 3.1416
3. x = new nodoSimple(d)

al ejecutar la instrucción 3 el programa se comunica con el sistema operativo y éste le asigna una posición de memoria, a nuestro programa, el cual identifica esta posición de memoria con la variable **x**. El resultado obtenido es:



en el cual, 19 es la dirección del nodo en la cual se almacenan los campos de dato y liga. Dicha dirección la suministra el sistema operativo, de una forma transparente para el usuario.

Si queremos acceder los campos del nodo **x** lo haremos así:

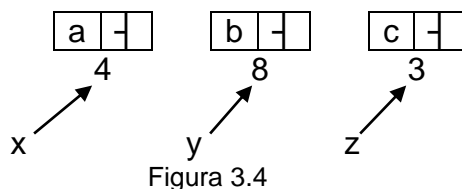
```
d = x.retornaDato()           // d queda valiendo 3.1416
z = x.retornaLiga()         // z queda valiendo nulo
```

Un ejemplo más completo de la utilización de la clase `nodoSimple` presentamos a continuación.

Supongamos que tenemos el conjunto de datos **a**, **b**, **c**, **f** y que se procesan con el siguiente algoritmo:

1. `nodoSimple x, y, z, p`
2. `read(d)` // lee la letra 'a'
3. `x = new nodoSimple(d)` // digamos que envió el nodo 4
4. `read(d)` // lee la letra 'b'
5. `y = new nodoSimple(d)` // digamos que envió el nodo 8
6. `read(d)` // lee la letra 'c'
7. `z = new nodoSimple(d)` // digamos que envió el nodo 3

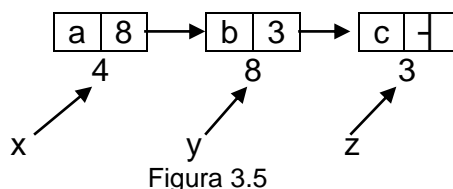
El resultado gráfico al ejecutar estas instrucciones es:



Ahora, si ejecutamos las instrucciones:

8. `x.asignaLiga(y)`
9. `y.asignaLiga(z)`

los nodos quedarán conectados así:



Si ejecutamos las siguientes instrucciones:

10. `read(d)` // lee la letra 'f'
11. `y = new nodoSimple(d)` // digamos que envió el nodo 7

nuestros nodos quedan:

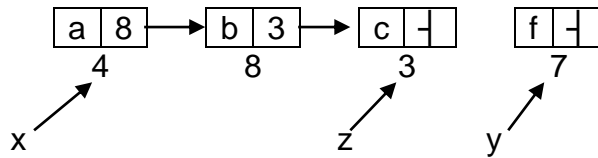


Figura 3.6

Ahora, ejecutemos estas instrucciones:

12. `p = x.retornaLiga()`
13. `write(p.retornaDato())`

la variable `p` queda valiendo 8 y escribe la letra `b`.

y si ejecutamos esta:

14. `x.retornaLiga().asignaLiga(y)`

nuestros nodos quedan:

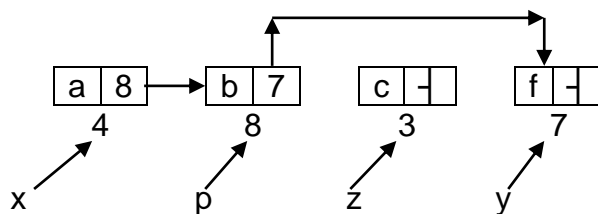


Figura 3.7

Observe que cuando se ejecuta `x.retornaLiga()` se está obteniendo el contenido del campo de liga del nodo `x`, el cual es otro nodo, el nodo 8 en nuestro ejemplo. Entonces, el nodo 8 invoca el método `asignaLiga()` y le estamos enviando como parámetro el nodo `y`, que ya en este momento vale 7. Podemos reorganizar el dibujo de nuestros nodos así:

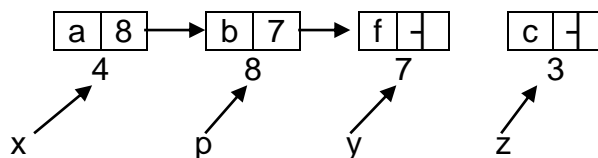


Figura 3.8

Observe que esta figura es exactamente la misma que la figura 3.7, sólo que se han reacomodado para verla mejor.

Ahora, si ejecutamos las siguientes instrucciones:

15. `y.asignaLiga(z)`
16. `y = null`
17. `z = null`
18. `p = x`

nuestra lista queda así:

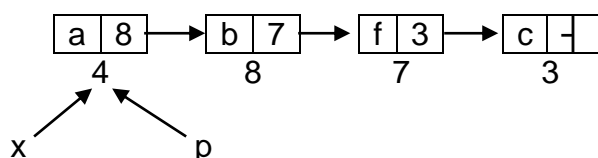


Figura 3.9

Y ahora, ejecutemos estas instrucciones:

```
19. while (p != null) do
20.     write(p.retornaDato())
21.     p = p.retornaLiga()
22. end(while)
```



en la instrucción 20, la primer vez que entra al ciclo escribe la 'a', puesto que p vale 4. En la instrucción 21 modifica el valor de p: a p le asigna lo que hay en el campo de liga del nodo 4, es decir 8. Nuestra lista está así:

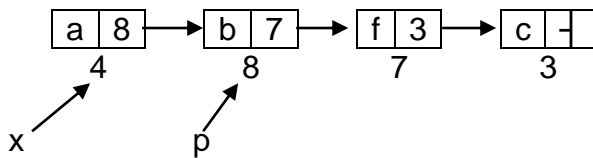


Figura 3.10

La segunda vez que entra al ciclo escribe la letra 'b' y la p queda valiendo 7. Nuestra lista está así:

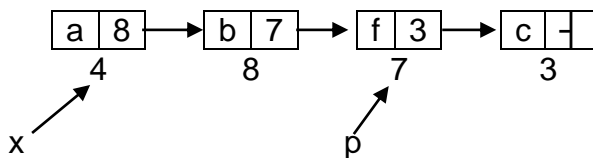


Figura 3.11

La tercera vez que entra al ciclo escribe la letra 'f' y la p queda valiendo 3:

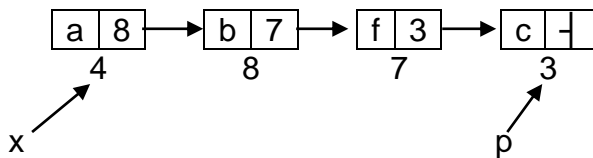


Figura 3.12

La cuarta vez que entra al ciclo escribe la letra 'c' y la p queda valiendo null, por consiguiente, termina el ciclo. Este ciclo, instrucciones 19 a 22, tiene orden de magnitud $O(n)$ siendo n el número de nodos de la lista.

Ahora, si por último ejecutamos estas instrucciones:

```
23. x.asignaLiga(x.retornaLiga().retornaLiga())
```

la lista queda:

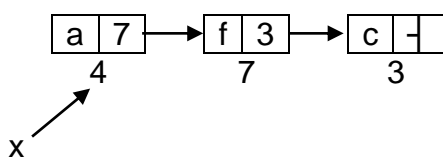


Figura 3.13

Observe que no hemos dibujado el nodo 8, ya que no está conectado con los otros, se ha desconectado de la lista con la instrucción 23. Lo que hace la instrucción 23 se puede escribir también con las siguientes instrucciones:

```
23a. y = x.retornaLiga()
```

23b. x.asignaLiga(y.retornaLiga())

O

23a. y = x.retornaLiga()

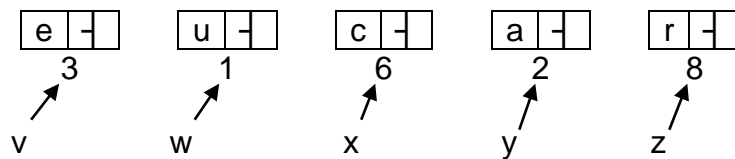
23b. y = y.retornaLiga()

23c. x.asignaLiga(y)

Es supremamente importante que entienda bien este ejemplo de aplicación de la clase nodoSimple, puesto que es la base para trabajar la clase lista ligada del siguiente módulo.

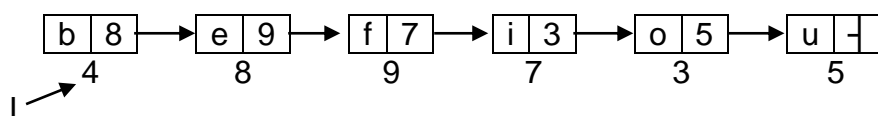
EJERCICIOS PROPUESTOS

1. Dados los siguientes nodos:



- Escriba las instrucciones para conectarlos, de tal manera que al recorrerlos queden escritos ascendentemente.
- Redibuje la lista de tal forma que los datos se vean ordenados ascendentemente.
- 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.
- 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.
- 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".

2. Dados los siguientes nodos conectados y la variable L que apunta hacia el nodo 4



- Escriba algoritmo que imprima el dato del nodo 7.
- Escriba algoritmo que desconecte el nodo 9.
- Escriba algoritmo que inserte un nuevo nodo con dato "q" a continuación del nodo 3.
- Escriba algoritmo que escriba los datos de todos los nodos dibujados.

MÓDULO 4

MANEJO DINÁMICO DE MEMORIA (CLASE LISTA SIMPLEMENTE LIGADA)

Introducción

En el módulo anterior tratamos la clase nodo simple. Con base en ella podemos trabajar colecciones de datos, representando cada uno de ellos en un nodo y tener conectados todos los nodos correspondientes a dicha colección. Trataremos en este módulo la manipulación de estos conjuntos de nodos conectados, los cuales se conocen como listas simplemente ligadas.

Objetivos

1. Definir la clase simplemente ligada.
2. Definir e implementar las operaciones con objetos de la clase lista simplemente ligada.
3. Aprender a construir listas simplemente ligadas.

Preguntas básicas

1. Qué es una lista simplemente ligada?
2. Cuáles son las operaciones básicas en listas simplemente ligadas?
3. De qué operaciones consta el proceso de inserción en una lista simplemente ligada?
4. De qué operaciones consta el proceso de borrado en una lista simplemente ligada?
5. De cuáles formas se puede construir una lista simplemente ligada?

4.1 Clase Lista Simplemente Ligada: Es un conjunto de nodos conectados cuyo elemento básico son objetos de la clase nodoSimple. Con el fin de poder operar sobre este conjunto de nodos es necesario conocer el primer nodo del conjunto de nodos que están conectados y en muchas situaciones el último nodo del conjunto. Con base en esto vamos a definir una clase llamada **LSL**, la cual tendrá dos datos privados de la clase nodoSimple, que llamaremos **primero** y **ultimo**: primero apuntará hacia el primer nodo de la lista y ultimo apuntará hacia el último nodo de la lista. Además, definiremos las operaciones que podremos efectuar sobre objetos de dicha clase.

Clase LSL

Privado

nodoSimple primero, ultimo

Público

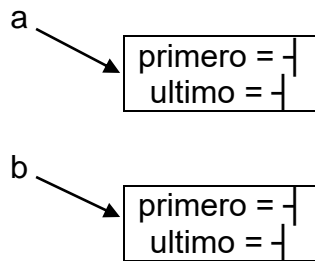
```
LSL() // constructor
boolean esVacía()
nodoSimple primerNodo()
nodoSimple ultimoNodo()
nodoSimple anterior(nodoSimple x)
boolean finDeRecorrido(nodoSimple x)
void recorre()
nodoSimple buscaDondeInsertar(Objeto d)
void insertar(Objeto d, nodoSimple y)
void conectar(nodoSimple x, nodoSimple y)
nodoSimple buscarDato(Objeto d, nodoSimple y)
void borrar(nodoSimple x, nodoSimple y)
void desconectar(nodoSimple x, nodoSimple y)
void ordenaAscendentemente()
```

fin(clase LSL)

Explicaremos brevemente cada uno de los métodos definidos. Comencemos con el constructor. Cuando se ejecuta el constructor lo único que se hace es crear una nueva instancia de la clase LSL con sus correspondientes datos privados en null. Ejemplo: si tenemos estas instrucciones:

1. LSL a, b
2. a = new LSL()
3. b = new LSL()

lo que se obtiene es:



La función `esVacia()` retorna verdadero si la lista que invoca el método está vacía, falso de lo contrario. Una lista está vacía cuando la variable `primero` es null.

Para explicar qué es lo que hace cada uno de los métodos definidos en la clase LSL consideremos el siguiente objeto, perteneciente a la clase LSL y que llamamos **a**:

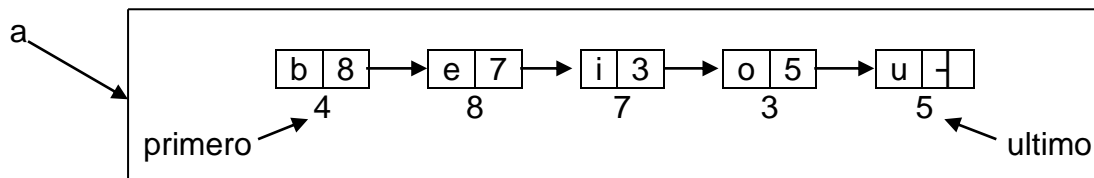


Figura 4.1

La función `primerNodo()` retorna el nodo que está de primero en la lista. Si efectuamos la instrucción `p = a.primerNodo()` entonces **p** quedará valiendo 4.

La función `ultimoNodo()` retorna el nodo que está de último en la lista. Si ejecutamos la instrucción `p = a.ultimoNodo()` entonces **p** quedará valiendo 5.

La función `finDeRecorrido(p)` retorna verdadero si el nodo **p** enviado como parámetro es null, falso de lo contrario.

La función `anterior(x)` retorna el nodo anterior al nodo enviado como parámetro. Si tenemos que `x = 7` y ejecutamos la instrucción `p = a.anterior(x)` entonces **p** quedará valiendo 8.

El método `recorre()` simplemente, como su nombre lo dice, recorre y escribe los datos de una lista simplemente ligada. Si ejecutamos la instrucción `a.recorre()` el resultado que se obtiene es la escritura de b, e, i, o, u.

La función `buscaDondeInsertar(d)` retorna el nodo a continuación del cual se debe insertar un nuevo nodo con dato **d** en una lista simplemente ligada en la cual los datos están ordenados ascendentemente y deben continuar cumpliendo esta característica después de insertarlo. Presentemos algunos ejemplos:

- | | |
|--|--|
| <code>y = a.buscaDondeInsertar('f')</code> | entonces y queda valiendo 8. |
| <code>y = a.buscaDondeInsertar('z')</code> | entonces y queda valiendo 5 |
| <code>y = a.buscaDondeInsertar('a')</code> | entonces y queda valiendo null. |

El método insertar(d, y) consigue un nuevo nodoSimple, lo carga con el dato d e invoca el método conectar con el fin de conectar el nuevo nodo (llamémoslo x) a continuación del nodo y. Si ejecutamos las siguientes instrucciones:

```
d = 'f'
y = a.buscaDondeInsertar(d)
a.insertar(d, y)
```

el objeto a quedará así:

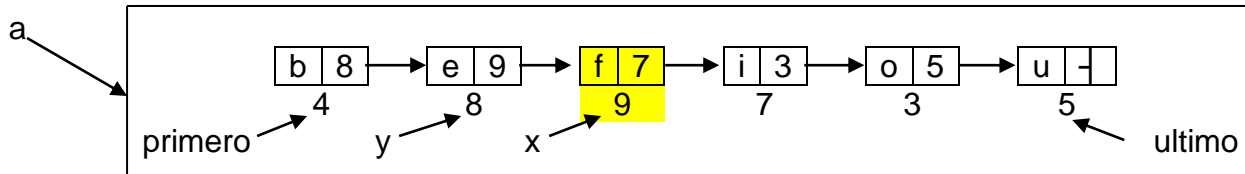


Figura 4.2

El método conectar simplemente conecta el nodo x a continuación del nodo y, tal como se ve en la figura 4.2.

La función buscarDato(d, y), como su nombre lo dice, busca el dato d en la lista que invoca el método: Si lo encuentra, retorna el nodo en el cual lo encontró, de lo contrario, retorna null. En el parámetro y, el cual debe ser un parámetro por referencia, retorna el nodo anterior al nodo en el cual encontró el dato d. Algunos ejemplos, considerando la lista de la figura 4.2 son:

```
x = a.buscarDato('f', y) entonces x queda valiendo 9 y y queda valiendo 8.
x = a.buscarDato('u', y) entonces x queda valiendo 5 y y queda valiendo 3.
x = a.buscarDato('b', y) entonces x queda valiendo 4 y y queda valiendo null.
x = a.buscarDato('m', y) entonces x queda valiendo null y y queda valiendo ultimo.
```

El método borrar(x, y) controla que el parámetro x sea diferente de null: si x es null produce el mensaje de que el dato d (el dato buscado con el método buscarDato) no se halla en la lista y retorna; si x es diferente de null, invoca el método desconectar(x, y).

El método desconectar(x, y) simplemente desconecta el nodo x de la lista que invoca el método. Para desconectar un nodo de una lista se necesita conocer cuál es el nodo anterior. Por tanto, el nodo y, el segundo parámetro, es el nodo anterior a x. Si ejecutamos las siguientes instrucciones sobre la lista de la figura 4.2

```
d = 'i'
x = a.buscarDato(d, y) // x queda valiendo 7 y y queda valiendo 9
a.borrar(x, y) // desconecta el nodo x
```

Ésta queda así:

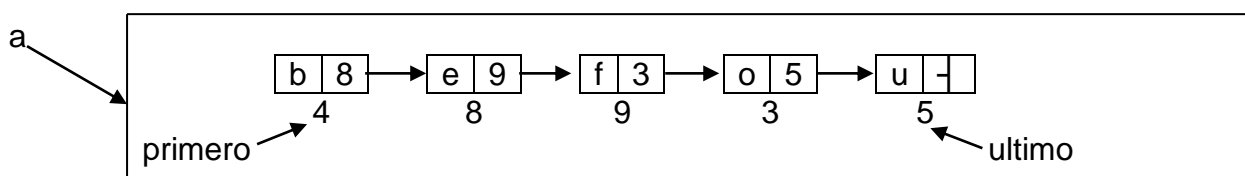


Figura 4.3

El método ordenaAscendentemente(), como su nombre lo dice, reorganiza los nodos de una lista simplemente ligada, de tal manera que los datos en ella queden ordenados ascendentemente.

Si tenemos la siguiente lista

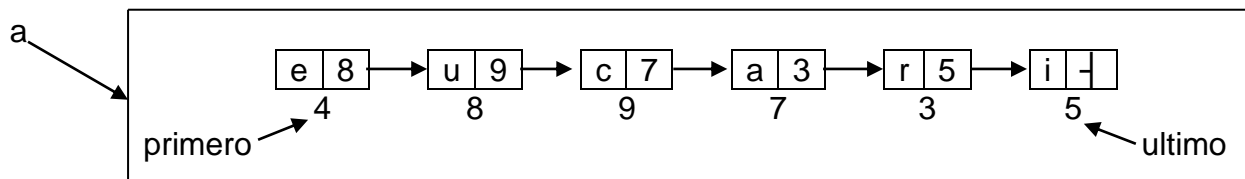


Figura 4.4

y ejecutamos la instrucción a.ordenarAscendentemente() la lista quedará así:

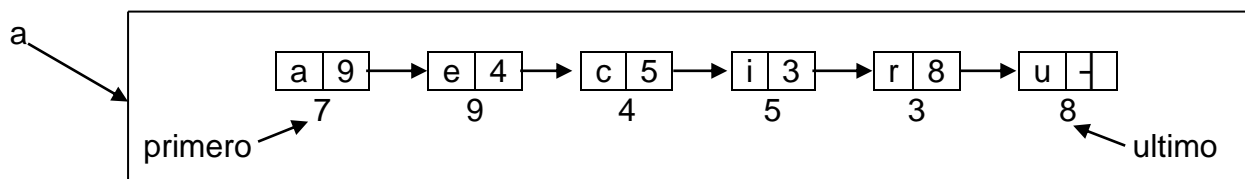


Figura 4.5

Observe que los datos no se han movido, son los nodos los que han cambiado de posición y por consiguiente primero y ultimo han variado.

Asegúrese de entender bien qué es lo que hace cada método. Si Ud. Conoce bien lo que hace cada método, podrá usarlos apropiadamente, de lo contrario no. Como ingeniero de sistemas Ud., debe saber hacer uso de los métodos de una clase y además cómo implementarlos.

Pasemos a ver cómo se implementa cada uno de los métodos que hemos definido.

4.2 Implementación de los métodos de la clase LSL.

El constructor:

```
LSL()
    primero = ultimo = null
fin(LSL)
```

Simplemente inicializa los datos privados primero y ultimo en null.

```
boolean esVacia()
    return primero == null
fin(esVacia)
```

Recuerde que tanto en C como en java la instrucción

```
return x == y
```

es equivalente a:

```
if (x == y)
    return true
else
    return false
```

```

nodoSimple primerNodo()
    return primero
fin(primerNodo)

nodoSimple ultimoNodo()
    return ultimo
fin(ultimo)

boolean finDeRecorrido(nodoSimple x)
    return x == null
fin(finDeRecorrido)

1. void recorre()
2.     nodoSimple p
3.     p = primerNodo()
4.     while no finDeRecorrido(p) do
5.         write(p.retornaDato())
6.         p = p.retornaLiga()
7.     end(while)
8. fin(recorre)

```

Para recorrer y escribir los datos de una lista simplemente se requiere una variable auxiliar, la cual llamamos **p**. Dicha variable se inicializa con el primer nodo de la lista ligada (instrucción 3) y luego planteamos un ciclo (instrucciones 4 a 7), el cual se ejecuta mientras **p** sea diferente de null (método finDeRecorrido()). Cuando **p** sea null se sale del ciclo y termina la tarea. Observe que si por alguna razón el objeto que invoca este método tiene la lista vacía, nuestro algoritmo funciona correctamente. Simplemente no escribe nada. Es decir, con la instrucción 4 se controlan dos situaciones: lista vacía y fin de recorrido.

4.3 Proceso de inserción de un dato en una lista ligada. Para insertar un dato **d** en una lista ligada hemos definido tres métodos en nuestra clase LSL: buscarDondeInsertar(d), insertar(d, y) y conectar(x, y). Analicemos cada uno de ellos. Es importante recordar que en este proceso de inserción se requiere que los datos de la lista se hallen ordenados en forma ascendente.

Consideremos la lista de la figura 4.6 y que se desea insertar el dato 'g'. Lo primero que se debe hacer es determinar en qué sitio debe quedar la letra 'g' para que se cumpla que los datos continúen ordenados en forma ascendente. Por observación, nos damos cuenta que la 'g' debe quedar entre la 'f' y la 'o', es decir, a continuación del nodo 9.

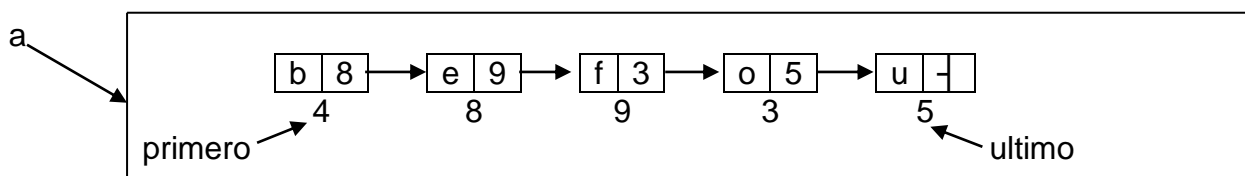


Figura 4.6

Nuestro primer método, buscarDondeInsertar(d) efectúa la tarea que determina a continuación de cuál nodo debe quedar el dato a insertar. El algoritmo para esta tarea es:

```

1. nodoSimple buscaDondeInsertar(objeto d)
2.     nodoSimple p, y


```



```

3.      p = primerNodo()
4.      y = anterior(p)
5.      while (no finDeRecorrido(p) and p.retornaDato() < d) do
6.          y = p
7.          p = p.retornaLiga()
8.      end(while)
9.      return y
10. end(buscaDondeInsertar)

```

En este algoritmo se requieren dos variables auxiliares, las cuales llamamos **p** y **y**. Con la variable **p** se recorre la lista y a medida que la vamos recorriendo se va comparando del dato de **p** (**p.retornaDato()**) con el dato **d** que se desea insertar. Mientras que el dato de **p** sea menor que **d** se avanza en la lista con **p** y con **y**. La variable **y** siempre estará apuntando hacia el anterior a **p**. Como inicialmente **p** es el primer nodo, inicialmente **y** es null. Fíjese que si el dato **d** a insertar es menor que el dato del primer nodo de la lista ligada nuestro algoritmo retorna null. El hecho de que nuestro algoritmo retorne null significa que el dato a insertar va a quedar de primero en la lista. 

Veamos ahora cómo es el algoritmo para nuestro método **insertar(d, y)**.

```

1. void insertar(Objeto d, nodosimple y)
2.     nodoSimple x
3.     x = new nodoSimple(d)
4.     conectar(x, y)
5. end(insertar)

```

Nuestro método **insertar** simplemente consigue un nuevo nodo, lo carga con el dato **d** e invoca el método **conectar**. El parámetro **y** se refiere al nodo a continuación del cual habrá que conectar el nuevo nodo **x**.

Ocupémonos del método **conectar(x, y)**.

Caso 1: al ejecutar **y = buscaDondeInsertar(d)** con **d = 'g'** en el objeto de la figura 4.6 y el método **insertar(d,y)** estamos en la situación mostrada en la figura 4.7.

Conectar el nodo **x** a continuación del nodo **y** implica que cuando lleguemos al nodo **y** debemos trasladarnos hacia el nodo **x**, o sea que el campo de liga del nodo **y** debe quedar valiendo 6, y cuando estemos en el nodo **x** debemos trasladarnos hacia el nodo 3, es decir, que el campo de liga del nodo **x** debe quedar valiendo 3. Es decir, hay que modificar dos campos de liga: el campo de liga

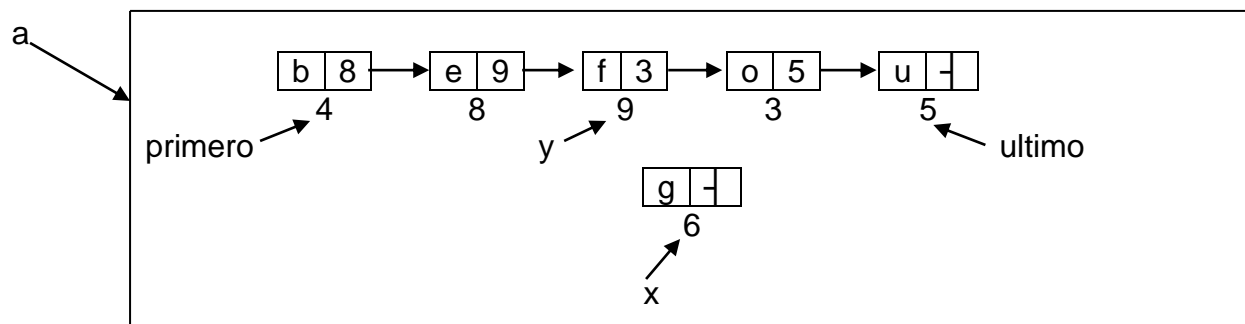


Figura 4.7

del nodo **y** y el campo de liga del nodo **x**. Para lograr esto debemos ejecutar las siguientes instrucciones:

```

x.asignaLiga(y.retornaLiga()) // Modifica el campo de liga del nodo x
y.asignaLiga(x)               // Modifica el campo de liga del nodo y

```

y la lista queda:

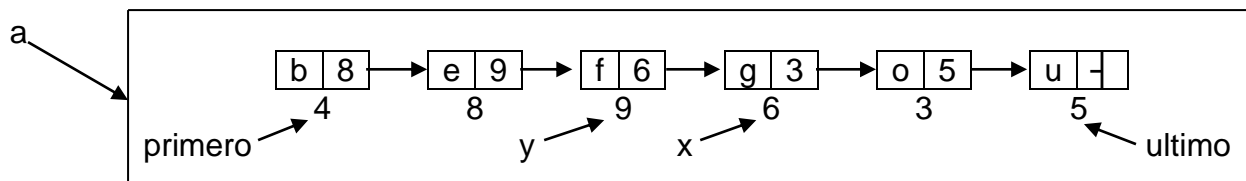


Figura 4.8

Que como se podrá observar, se ha insertado el nodo **x** a continuación del nodo **y**.

Caso 2: Consideremos que el dato a insertar es el dato $d = 'z'$. Al ejecutar $y = \text{buscaDondeInsertar}(d)$ en el objeto de la figura 4.6 y el método $\text{insertar}(d, y)$ estamos en la situación mostrada en la figura 4.9.

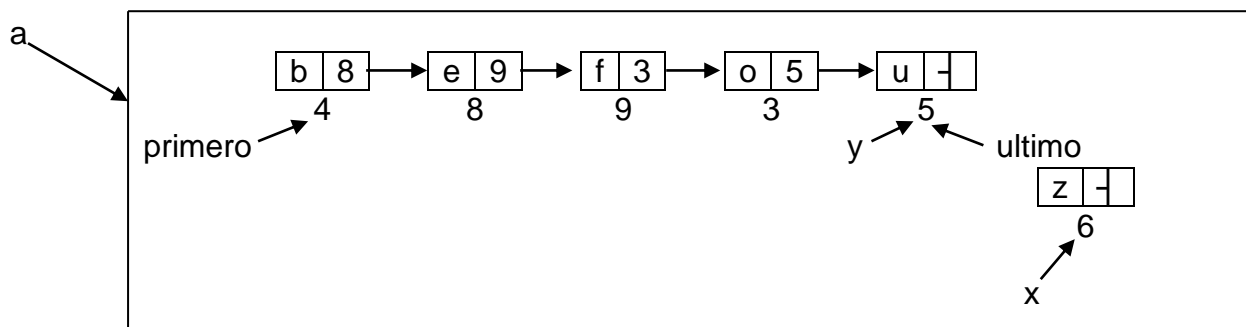


Figura 4.9

Conectar x a continuación de y se logra con las mismas instrucciones del caso 1, teniendo en cuenta que como el dato que se inserta queda de último hay que actualizar la variable **ultimo**. Lo anterior significa que las instrucciones para conectar x a continuación de y serán:

```

x.asignaLiga(y.retornaLiga()) // Modifica el campo de liga del nodo x
y.asignaLiga(x)               // Modifica el campo de liga del nodo y
if (y == ultimo) then
    ultimo = x
end(if)

```

y la lista queda:

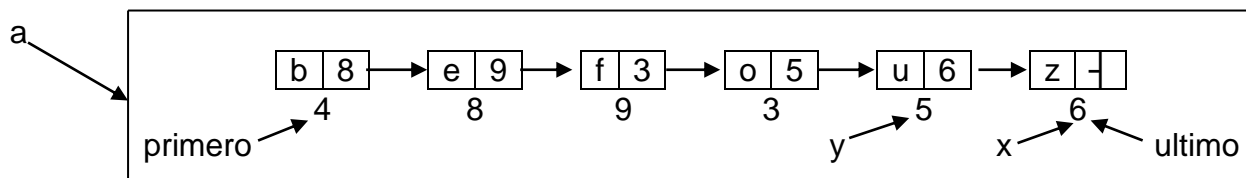


Figura 4.10

Caso 3: Consideremos que el dato a insertar es el dato $d = 'a'$. Al ejecutar $y = \text{buscaDondeInsertar}(d)$ en el objeto de la figura 4.6 y el método $\text{insertar}(d, y)$ estamos en la situación mostrada en la figura 4.11.

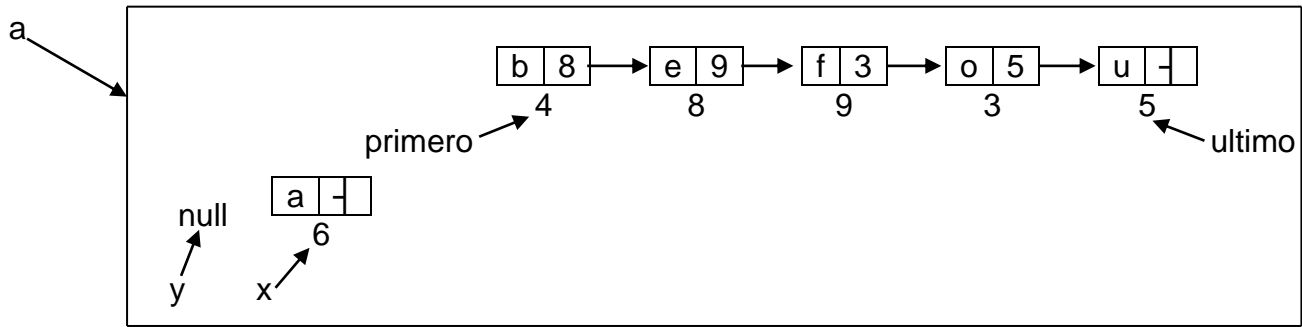


Figura 4.11

En esta situación, cuando la **y** es null, es decir que el dato a insertar quedará de primero, hay que modificar el campo de liga de **x** y la variable primero:

```
x.asignaLiga(primero)
primero = x
```

pero hay que tener en cuenta que la lista pudo haber estado vacía. Si esa fuera la situación habría que actualizar también la variable ultimo. Considerando estas situaciones las instrucciones completas para conectar un nodo **x** al principio de la lista es:

```
x.asignaLiga(primero)
if (primero == null) then
    ultimo = null
end(if)
primero = x
```

El algoritmo completo para el método conectar simplemente consiste en agrupar las instrucciones descritas para cada uno de los casos en un solo algoritmo. Dicho algoritmo se presenta a continuación: en instrucciones 2 a 7 se consideran los casos 1 y 2, mientras que en instrucciones 9 a 13 se considera el caso 3.

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

4.4 Proceso de borrado de un dato en una lista ligada. Borrar un dato de una lista ligada consiste en ubicar el nodo en el cual se halla el dato y desconectarlo de la lista. Para ello hemos definido tres métodos que hemos llamado buscarDato(d, y), borrar(x, y) y desconectar(x, y).

Si tenemos la siguiente lista:

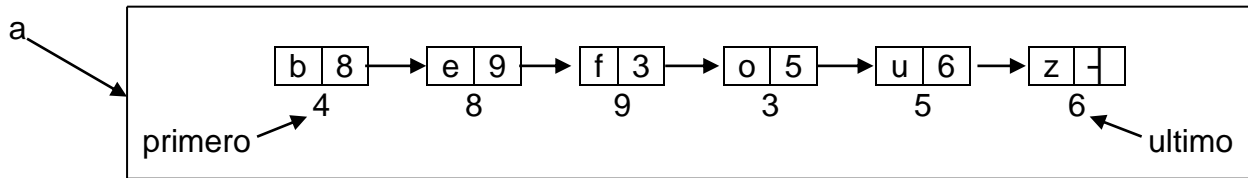


Figura 4.12

y queremos borrar el nodo que contiene la letra 'f' lo primero que debemos hacer es determinar en cuál nodo se halla la letra 'f' y cuál es su nodo anterior. Esta tarea se efectúa con el método buscarDato(d, y): ejecutamos la instrucción

```
x = a.buscarDato(d, y)
```

y estamos en esta situación:

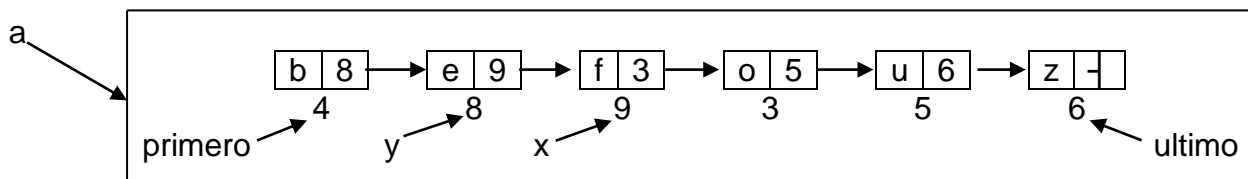


Figura 4.13

El algoritmo correspondiente al método buscarDato(d, y) es similar al algoritmo buscaDondeInsertar(d) y lo presentamos a continuación.

```

1. nodoSimple buscarDato(objeto d, nodoSimple y)
2.     nodoSimple x
3.     x = primerNodo()
4.     y = anterior(x)
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)

```

Utilizamos la variable **x** para recorrer la lista e ir comparando el dato del nodo **x** con **d**. De este ciclo, instrucciones 5 a 8, se sale cuando encuentre el dato o cuando haya terminado de recorrer la lista. Observe que cuando el dato que se busca no se halla en la lista el parámetro **y** queda valiendo ultimo. Además, cuando el dato que se busca es el primer dato de la lista la variable **y** queda valiendo null.

Nuestro método borrar(x, y) simplemente consiste en controlar que el nodo **x** sea diferente de null. Si el nodo **x** es null significa que el dato no se encontró en la lista, por consiguiente no hay ningún nodo para desconectar entonces produce el mensaje correspondiente y retorna al programa llamante puesto que no hay nada más que hacer. Si **x** es diferente de null se invoca el método desconectar. El algoritmo correspondiente a dicho método se presenta a continuación.

```

void borrar(nodoSimple x, nodoSimple y)
    if x == null then
        write("dato no existe")
        return
    end(if)
    desconectar(x, y)

```

fin(borrar)

Analicemos ahora las instrucciones correspondientes al método desconectar(x, y).

Caso 1: es la situación mostrada en la figura 4.13. Lo único que hay que hacer para desconectar el nodo **x** es modificar el campo de liga del nodo **y**, asignándole lo que hay en el campo de liga del nodo **x**. Esto se logra con la siguiente instrucción:

y.asignaLiga(x.retornaLiga())

Caso 2: Si el dato a borrar es d = 'z', al ejecutar la instrucción

x = a.buscarDato(d, y)

la situación es:

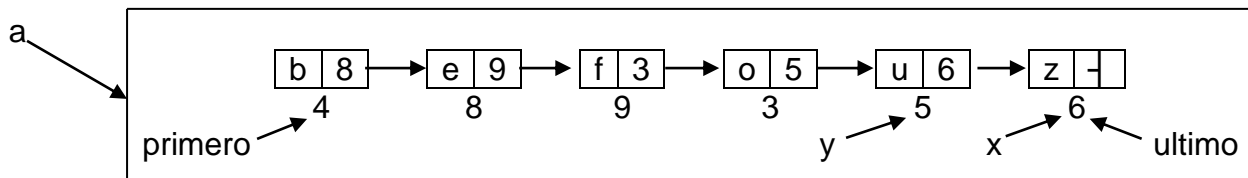


Figura 4.14

Por consiguiente, cuando se desconecte el nodo **x** la variable **ultimo** debe quedar valiendo **y** y las instrucciones serán:

```
y.asignaLiga(x.retornaLiga())  
if (x == ultimo) then  
    ultimo = y  
end(if)
```

Caso 3: si el dato a borrar es d = 'b' al ejecutar la instrucción

x = a.buscarDato(d, y)

la situación es:

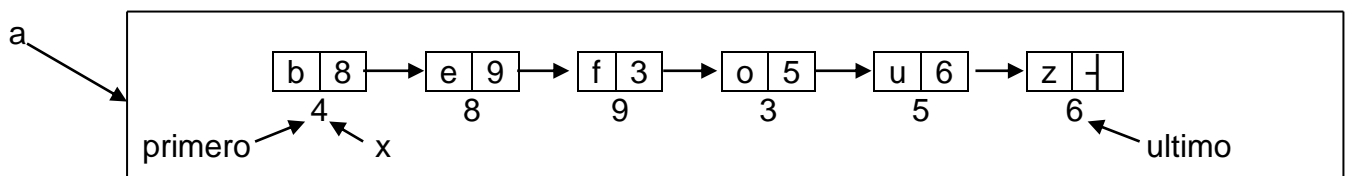


Figura 4.14

Con la variable **y** valiendo null.

En esta situación lo que hay que hacer es actualizar la variable **primero**: a **primero** se le asigna lo que haya en el campo de liga de **primero**. Hay que tener en cuenta que puede suceder que la lista sólo hubiera tenido un nodo (figura 4.15), en este caso la variable **primero** quedará valiendo null, por tanto la variable **ultimo** también deberá quedar valiendo null. Las instrucciones para resolver este caso son:

```
primero = primero.retornaLiga()  
if (primero == null) then  
    ultimo = null  
end(if)
```

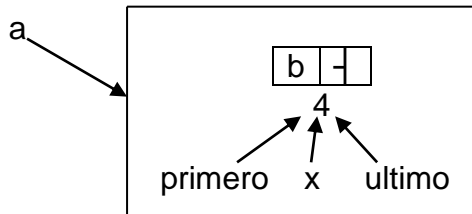


Figura 4.15

Agrupando las instrucciones correspondientes a los tres casos en un solo algoritmo obtenemos:

```

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

```

En instrucciones 2 a 6 se resuelve para los casos 1 y 2, mientras que en instrucciones 8 a 11 se resuelve para el caso 3.

Los algoritmos para conectar y desconectar son algoritmos con orden de magnitud **O(1)**, lo cual mejora sustancialmente estas operaciones, con respecto a la representación de datos en un vector.

4.5 Ordenamiento de datos en una lista simplemente ligada: presentamos un algoritmo que utiliza el método *selección*, el cual como se vio en el módulo 2 se enuncia: “*de los datos que faltan por ordenar determinar cuál es el menor y colocarlo de primero en ese conjunto de datos*”.

```

1. void ordenaAscendentemente()
2.   nodoSimple p, ap, menor, amenor, q, aq
3.   p = primerNodo()
4.   ap = anterior(p)
5.   while (p <> ultimoNodo())
6.     menor = p
7.     amenor = ap
8.     q = p.retornaLiga()
9.     aq = p
10.    while (no finDeRecorrido(q))
11.      if (q.retornaDato() < menor.retornaDato())
12.        menor = q
13.        amenor = aq
14.      end(if)
15.      aq = q
16.      q = q.retornaLiga()
17.    end(while)
18.    if (menor == p)

```

```

19.         ap = p
20.         p = p.retornaLiga()
21.     else
22.         desconectar(menor, amenor)
23.         conectar(menor, ap)
24.         ap = menor
25.     end(if)
26. end(while)
27. fin(ordenaAscendentemente)

```

Con base en este enunciado definimos las variables necesarias para elaborar el algoritmo. Se requiere una variable que indique a partir de cuál nodo faltan datos por ordenar. Dicha variable la llamamos **p**. Inicialmente **p** apunta hacia el primer nodo ya que al principio faltan todos los datos por ordenar. Se requiere otra variable para recorrer la lista e ir comparando los datos para determinar en cuál nodo se halla el menor dato. Esta variable la llamamos **q**. Y por último requerimos de otra variable que apunte hacia el nodo que tiene el menor dato. Esta variable la llamamos **menor**.

Nuestro algoritmo consistirá entonces en determinar cuál es el nodo que tiene el menor dato, entre los datos que faltan por ordenar, desconectamos el nodo de ese sitio y lo conectamos al principio de ese conjunto de datos.

Como ya hemos visto, para efectuar las operaciones de conexión y desconexión se requieren conocer los registros anteriores a los nodos con los cuales se desean ejecutar dichas operaciones, por tanto, definimos otras tres variables que llamaremos **ap**, **aq** y **amenor**, las cuales apuntan hacia los nodos anteriores a **p**, **q** y **menor** respectivamente.

En la instrucción 5 definimos el ciclo principal del ciclo. El algoritmo terminará cuando **p** esté apuntando hacia el último nodo de la lista, puesto que cuando **p** esté apuntando hacia ese nodo, significa que sólo falta un dato por ordenar, y un solo dato está ordenado.

En las instrucciones 6 y 7 asignamos los valores iniciales de **menor** y **amenor**. Inicialmente consideramos que el menor dato es el primero del conjunto de datos que faltan por ordenar, es decir, **p**.

En las instrucciones 8 y 9 se asignan los valores iniciales de **q** y **aq**.

En las instrucciones 10 a 17 se efectúa el ciclo para determinar cuál es el nodo que contiene el menor dato. Se compara el dato del nodo **q** con el dato del nodo **menor**. Si el dato de **q** es menor que el dato de **menor** actualizamos **menor** y su anterior. Cualquiera que hubiera sido el resultado avanzamos con **q** y su anterior.

En la instrucción 18 confrontamos en cuál posición está el nodo que tiene el menor dato. Si el menor dato estaba de primero en ese conjunto de datos, es decir, en **p**, simplemente avanzamos con **p** y su anterior (instrucciones 19 y 20), de lo contrario desconectamos el nodo **menor**, cuyo anterior es **amenor** y lo conectamos a continuación de **ap** (instrucciones 22 y 23). De haber ocurrido esto último, el anterior a **p**, es decir **ap**, ya es **menor** (instrucción 24).

4.6 Método anterior(x): ya hemos definido que el nuestro método denominado anterior(x) retornará el nodo anterior al nodo **x** enviado como parámetro. Para ello bastará con recorrer la lista ligada utilizando dos variables auxiliares: una (llamémosla **p**), que se va comparando con **x** y otra que siempre apuntará hacia el nodo anterior a **x** (llamémosla **y**). Cuando **p** sea igual a **x**, simplemente se retorna **y**. Inicialmente será el primer nodo y **y** será null. Nuestro algoritmo es el siguiente:

```

nodoSimple anterior(x)
    nodoSimple p, y
    p = primerNodo()
    y = null
    while (p != x) do
        y = p
        p = p.retornaLiga()
    end(while)
    return y
fin(anterior)

```

4.7 Construcción de Listas Ligadas. Pasemos ahora a considerar cómo construir listas ligadas. Básicamente hay tres formas de construir una lista ligada. Una, que los datos queden ordenados ascendentemente a medida que la lista se va construyendo; otra, insertando nodos siempre al final de la lista, y una tercera, insertando los nodos siempre al principio de la lista.

Para la primera forma de construcción, que los datos queden ordenados ascendentemente a medida que se va construyendo la lista, un algoritmo general es:

```

LSL a
a = new LSL()
mientras haya datos por leer haga
    lea(d)
    y = a.buscaDondeInsertar(d)
    a.insertar(d, y)
fin(mientras)

```

Es decir, basta con plantear un ciclo para lectura de datos e invocar los métodos para buscar dónde insertar e insertar desarrollados previamente.

Para la segunda forma de construcción, insertando nodos siempre al final de la lista, un algoritmo es:

```

LSL a
nodoSimple y
a = new LSL()
mientras haya datos por leer haga
    lea(d)
    y = a.ultimoNodo()
    a.insertar(d, y)
fin(mientras)

```

Nuevamente, se plantea un ciclo para la lectura de datos y cada vez que se lea un dato se determina el ultimo y se invoca el método para insertar.

Para la tercera forma de construcción, insertando nodos siempre al principio de la lista, nuestro algoritmo es:

```

LSL a
a = new LSL()
mientras haya datos por leer haga
    lea(d)
    a.insertar(d, null)

```


fin(mientras)

Aquí, basta con plantear el ciclo para lectura de datos e invocar el método insertar enviando como segundo parámetro null. Recuerde que nuestro método insertar(d, y) inserta un nodo con dato **d** a continuación de **y**: si **y** es null significa que el dato **d** quedará de primero.

EJERCICIOS PROPUESTOS

1. Elaborar algoritmo que lea un entero n y que construya una lista ligada, de a dígito por nodo.
2. Elaborar algoritmo que borre de una lista ligada un dato dado, todas las veces que lo encuentre.
3. Se tiene una lista ligada, con un dato numérico en cada nodo. Elabore un algoritmo que determine e imprima el promedio de datos de la lista.
4. Elabore un algoritmo que retorne el nodo que contiene el dato mayor en una lista simplemente ligada.
5. Elabore un algoritmo que retorne el nodo que contiene el menor dato en una lista simplemente ligada.
6. Elabore un algoritmo que intercambie el primer registro con el último en una lista simplemente ligada.
7. Elabore un algoritmo que sume los datos impares en una lista simplemente ligada.
8. Elabore un algoritmo que sume los datos de los registros que están en las posiciones pares de la lista ligada.
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 algoritmo que construya una tercera lista ligada con los datos de A que están en B.
10. Elabore un algoritmo que borre de una lista ligada todos los nodos que tengan dato par sabiendo que cada nodo contiene un dato numérico.
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 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.
12. Elabore un algoritmo que intercambie los registros de una lista ligada así: el primero con el segundo, el tercero con el cuarto, el quinto con el sexto y así sucesivamente.
13. Elabore algoritmo que intercambie los registros de una lista ligada así: el primero con el último, el segundo con el penúltimo, el tercero con el antepenúltimo, y así sucesivamente.
14. Elabore algoritmo que intercambie los registros de una lista 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.
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:

→

0	e	1
---	---	---

 →

0	u	6
---	---	---

 →

0	c	2
---	---	---

 →

0	a	8
---	---	---

 →

0	r	4
---	---	---

 →

0	i	7
---	---	---

 →

0	s	-
---	---	---

Al ejecutar su algoritmo la lista debe quedar:

→

3	e	1
---	---	---

 →

7	u	6
---	---	---

 →

2	c	2
---	---	---

 →

1	a	8
---	---	---

 →

5	r	4
---	---	---

 →

4	i	7
---	---	---

 →

6	s	-
---	---	---

16. Se tienen tres listas simplemente ligadas. Elabore un algoritmo que construya una cuarta lista con los datos comunes a las tres listas.

MÓDULO 5

MANEJO DINÁMICO DE MEMORIA (APLICACIÓN LISTA SIMPLEMENTE LIGADA)

Introducción

Es a veces necesario dentro de algunas aplicaciones manejar números enteros cuya capacidad supera la que manejan los tipos predefinidos por los lenguajes de programación. Por ejemplo, si se desea conocer exactamente la distancia al sol en milímetros, ninguna máquina de las actuales soportaría esa cantidad de dígitos, por tanto debemos buscar una forma de representación en la cual podamos obtener dicha cifra y además hacer operaciones aritméticas con dichos números. Presentamos en este módulo una forma de hacerlo utilizando listas ligadas.

Objetivos

1. Definir la forma de representar enteros usando la clase LSL.
2. Desarrollar algoritmos para manipular números enteros bajo esta representación.

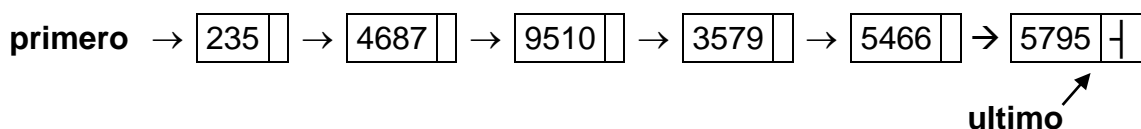
Preguntas básicas

1. Cuántos dígitos se pueden almacenar por nodo para representar enteros en listas simplemente ligadas?
2. Qué cuidado se debe tener para escribir un entero representado como lista ligada?
3. Por qué definimos la clase `altaPrecision` derivada de la clase LSL?

5.1 Representación: Primero que todo definamos la forma como representaremos números enteros de esta magnitud utilizando listas ligadas. Si deseamos representar el siguiente número:

23546879510357954665795

usaremos una lista ligada almacenando de a cuatro dígitos por nodo, conformando los grupos de 4 dígitos de derecha a izquierda. Nuestra lista queda así:



En general, se pueden almacenar 1, 2, 3 o cualquier otra cantidad de dígitos por nodo, sin embargo, hemos elegido 4 porque permite definir el dato almacenado en cada nodo como entero y manipularlo eficientemente.

Teniendo definida la representación, el siguiente paso será desarrollar los programas tendientes a manipular los números enteros de alta precisión bajo dicha representación. Es decir, algoritmos para sumar, restar, etc., números enteros bajo esta representación.

5.2 Clase `altaPrecision`: Como hemos decidido representar números enteros con muchos dígitos como lista ligada, definiremos una clase llamada **`altaPrecisión`**, la cual será derivada de nuestra clase lista simplemente ligada (**LSL**), lo cual implica que todo objeto definido de **`altaPrecisión`** podrá hacer uso de todos los métodos definidos para la clase **LSL**. Como queremos que los objetos de la clase **`altaPrecisión`** puedan acceder también los datos privados de la clase **LSL**, definimos dichos datos como **protegidos**, en vez de privados. Los métodos a desarrollar serán:

- Leer número,
- Escribir número.
- Sumar dos números.
- Restar dos números.
- Multiplicar dos números.
- Dividir dos números.

Definamos entonces nuestra clase **altaPrecisión**.

Clase **altaPrecisión** derivada de **LSL**

Público

```
void leeNúmero()  
void escribeNúmero()  
altaPrecision sume(altaPrecision b)  
altaPrecisión reste(altaPrecision b)  
altaPrecisión multiplique(altaPrecision b)  
altaPrecisión divide(altaPrecision b)  
void evalúe(entero d1, entero d2, entero acarreo)
```

fin(altaPrecision)

El método **leeNúmero()** aceptará como entrada una hilera de caracteres, la procesará y construirá una lista simplemente ligada, almacenando de a cuatro dígitos por nodo. En el módulo correspondiente a manejo de caracteres presentaremos este algoritmo.

El método **escribeNúmero()** imprimirá el número representado en una lista.

Los métodos para sumar, restar, multiplicar y dividir, crearán una nueva lista en la cual se tendrá el resultado correspondiente a efectuar cada operación con el objeto de llamada y el objeto **b** entrado como parámetro.

5.3 Suma de enteros de alta precisión: teniendo definida la clase `altaPrecision` podremos escribir un programa para manejar números de alta precisión, haciendo uso de los métodos definidos para esta clase. Un programa podría ser:

1. `altaPrecision a, b, c`
2. `a = new altaPrecision()`
3. `b = new altaPrecision()`
4. `a.leeNumero()`
5. `b.leeNumero()`
6. `c = a.sume(b)`
7. `a.escribeNumero()`
8. `b.escribeNumero()`
9. `c.escribeNumero()`

En este algoritmo definimos tres variables de la clase `altaPrecision`: se crean dos objetos vacíos, instrucciones 2 y 3; se construye la lista correspondiente a cada uno de esos números, instrucciones 4 y 5; se crea un nuevo número, el cual es la suma de los números leídos, usando el método `suma` de finido en nuestra clase, y por último se escriben los números leídos y el resultado de la suma de ellos, instrucciones 7 a 9.

Ocupémonos ahora del algoritmo de sumar.

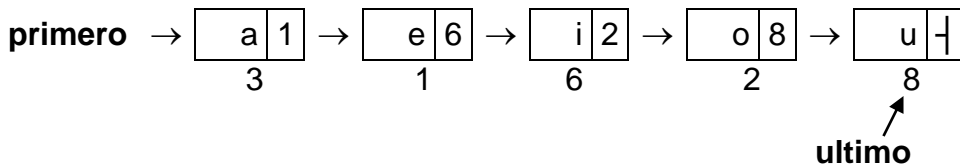
Si deseamos sumar 4358795 con 62759, en nuestra aritmética decimal, la suma se efectúa así:

```
  4358795  
    62759  
-----  
  4421554
```

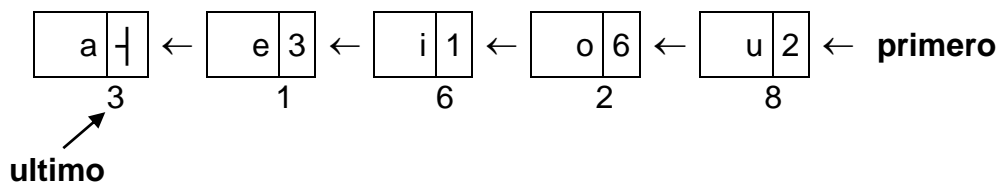
es decir, comenzamos sumando primero los dígitos menos significativos (los dígitos más a la derecha) y nos movemos con ambos números de derecha a izquierda teniendo en cuenta si se lleva algún acarreo o no. Lo anterior implica que para poder efectuar la suma con nuestra

representación, debemos ubicarnos en el último nodo de cada lista y luego devolvemos en ellas simultáneamente efectuando la suma del número contenido en cada nodo, teniendo en cuenta el acarreo. Pero, como ya hemos visto anteriormente, las lista ligadas que tenemos definidas sólo se pueden recorrer en un solo sentido: de izquierda a derecha.

Teniendo en cuenta esta restricción debemos definir un mecanismo que nos permita recorrer las listas ligadas en sentido inverso, es decir, de derecha a izquierda. Para lograr esto, lo que haremos será reversar los apuntadores de la lista, es decir, el campo de liga de cada nodo apuntará hacia el nodo anterior y no hacia el siguiente. Llamemos este método **reversaLista()**. El efecto de este algoritmo será: si tenemos la siguiente lista



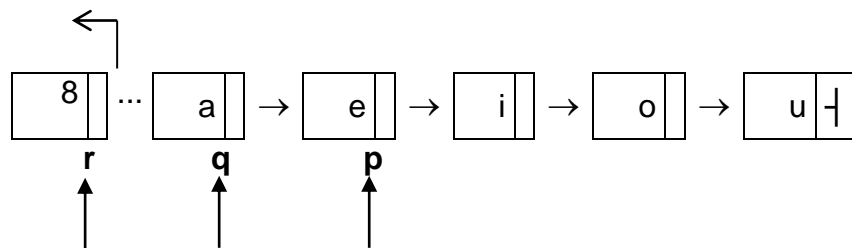
y le aplicamos el algoritmo **reversaLista()**, la lista quedará:



y la podremos recorrer de derecha a izquierda.

Ocupémonos de elaborar el algoritmo correspondiente a dicho método. Para lograr este objetivo sólo necesitamos conocer la dirección de tres nodos consecutivos: llamémoslos **r**, **q**, y **p**, donde **r** apunta hacia el nodo anterior a **q**, **q** apunta hacia el nodo anterior a **p**, y **p** es la variable con la cual iremos recorriendo la lista.

Esquemáticamente tendríamos la siguiente situación:



y el proceso de reversar apuntadores se efectuaría con las siguientes instrucciones:

```

q.asignaLiga(r)
r = q
q = p
p = p.retornaLiga()
  
```

En las cuales, la primera instrucción reversa el apuntador de **q** y en las tres siguientes se avanza con **r**, **q** y **p**.

Entendiendo las anteriores instrucciones, el siguiente paso es incrustarlas dentro de un ciclo de recorrido de la lista y habremos reversado los apuntadores de la lista.

Nuestro algoritmo será:

```

void reversaLista()
    nodoSimple  p, q, r
    p = primerNodo()
    ultimo = p
    q = anterior(p)
    while (no finDeRecorrido(p)) do
        r = q
        q = p
        p = p.retornaLiga()
        q.asignaLiga(r)
    end(while)
    primero = q
fin(subprograma)

```

Este algoritmo pertenecerá a la clase **LSL** ya que esta operación, que se requiere con objetos de la clase **altaPrecision**, también podrá ser útil en muchas otras situaciones.

Veamos cómo es nuestro algoritmo para el método de suma.

```

1. altaPrecision  sume(altaPrecision b)
2.     nodoSimple  p, q, acarreo
3.     acarreo = new  nodoSimple(0)
4.     reversaLista()
5.     b.reversaLista()
6.     c = new  altaPrecision()
7.     p = primerNodo()
8.     q = b.primerNodo()
9.     while (no finDeRecorrido(p)  and  no b.finDeRecorrido(q)) do
10.         c.evaluate(p.retornaDato(), q.retornaDato(), acarreo)
11.         p = p.retornaLiga()
12.         q = q.retornaLiga()
13.     end(while)
14.     while (no finDeRecorrido(p))
15.         c.evaluate(p.retornaDato(), 0, acarreo)
16.         p = p.retornaLiga()
17.     end(while)
18.     while (no finDeRecorrido(p))
19.         c.evaluate(0, q.retornaDato(), acarreo)
20.         q = q.retornaLiga()
21.     end(while)
22.     if (acarreo.retornaDato() != 0) then
23.         c.insertar(acarreo.retornaDato(), null)
24.     end(if)
25.     reversaLista()
26.     b.reversaLista()
27.     return c
28. fin(sume)

```

Instrucción 4 reversa los apuntadores del objeto que invocó el método suma.

Instrucción 5 reversa los apuntadores del objeto enviado como parámetro.

El ciclo de instrucciones 9 a 13 es el ciclo principal del algoritmo: se recorren las dos listas simultáneamente sumando los contenidos de los datos de los nodos **p**, **q** y **acarreo**.

El ciclo de instrucciones 14 a 17 termina de recorrer la lista del objeto que invocó el método encaso de que haya terminado de recorrer primero la lista b.

El ciclo de instrucciones 18 a 21 termina de recorrer la lista del objeto enviado como parámetro en caso de que haya terminado de recorrer primero la lista que invocó el método.

En caso de que ambas listas se hubieran terminado de recorrer simultáneamente no ejecuta ninguno de estos dos ciclos.

En la instrucción 22 se controla el hecho de que haya terminado de recorrer ambas listas y que el acarreo sea diferente de cero.

En las instrucciones 25 y 26 se enderezan los apuntadores de las listas, los cuales habían sido reversados en las instrucciones 4 y 5.

Adicionalmente, nuestro algoritmo de suma utiliza un método denominado **evaluate**, el cual recibe como parámetros los datos a sumar, efectúa la suma de ellos, separa los últimos cuatro dígitos e inserta un nodo al principio de la lista correspondiente al objeto que está creando, es decir, el objeto **c**. El método insertar utilizado en la instrucción 23 y en el método evaluate es el definido para la clase LSL.

```
1. void evaluate(entero d1, entero d2, nodoSimple acarreo)
2.     entero s
3.     s = d1 + d2 + acarreo.retornaDato()
4.     acarreo = s / 10000
5.     s = s % 10000
6.     insertar(s, null)
7. fin(evaluate)
```

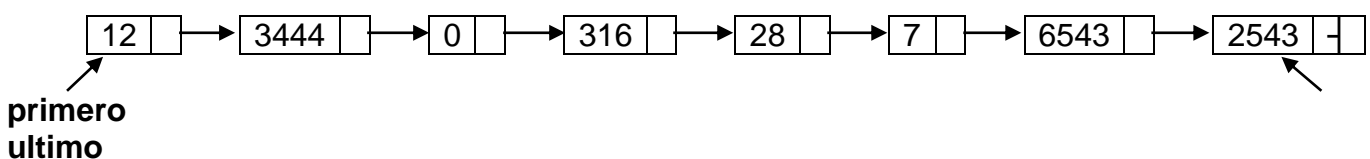
Es conveniente resaltar que el tercer parámetro, acarreo, debe ser un parámetro por referencia. Esa es la razón por la cual lo hemos incluido dentro de un objeto de la clase nodoSimple.

Recuerde que el operador % retorna el residuo de una división entera.

Teniendo ya estos subprogramas elaborados podemos escribir un algoritmo que lea tres números de alta precisión y los sume.

```
altaPrecision a, b, c, d, e
a.leeNumero()
b.leeNumero()
c.leeNumero()
d = a.sume(b)
e = d.sume(c)
a.escribeNumero()
b.escribeNumero()
c.escribeNumero()
e.escribeNumero()
```

5.4 Escritura de un número de alta precisión: presentamos ahora un algoritmo que es invocado por un objeto de la clase altaPrecision e imprime el número representado en él. Recuerde que tomamos la decisión de almacenar cuatro dígitos por nodo. Si el número que se está representando es 123444000003160028000765432543 la lista ligada que lo representa es:



Aquellos nodos en los cuales se representan números menores que 1000 hay que considerarlos como casos especiales a la hora de imprimir. Si consideramos el nodo en el cual está almacenado 316 y le decimos que escriba el dato de ese nodo el programa escribirá 316, y

nosotros necesitamos que escriba 0316; si consideramos el nodo en el cual está almacenado 28 y le decimos que escriba el dato de ese nodo el programa escribirá 28, y nosotros necesitamos que escriba 0028; si consideramos el nodo en el cual está almacenado 7 y le decimos que escriba el dato de ese nodo el programa escribirá 7, y nosotros necesitamos que escriba 0007. La única situación en la cual se pueden omitir los ceros a la izquierda es cuando se escriba el dato del primer nodo.

Un algoritmo que considera estas situaciones se presenta a continuación.

```
1. void escribeAltaPrecision()
2.     nodoSimple p
3.     if (esVacia()) then return
4.     p = primerNodo()
5.     write(p.retornaDato())
6.     p = p.retornaLiga()
7.     while (no finDeRecorrido(p)) do
8.         if (p.retornaDato() < 10) then
9.             write("000")
10.        else
11.            if (p.retornaDato() < 100) then
12.                write("00")
13.            else
14.                if (p.retornaDato() < 1000) then
15.                    write("0")
16.                end(if)
17.            end(if)
18.        end(if)
19.        write(p.retornaDato())
20.        p = p.retornaLiga()
21.    end(while)
22. fin(escribeAltaPrecision)
```

EJERCICIOS PROPUESTOS

1. Elabore algoritmo que reste dos números de alta precisión y deje el resultado en un nuevo objeto.
2. Elabore un algoritmo que sume dos números de alta precisión dejando el resultado en el objeto que invocó el método.
3. Elabore un algoritmo que multiplique dos números de alta precisión dejando el resultado en un nuevo objeto.
4. Elabore un algoritmo que divida dos números de alta precisión dejando el resultado en un nuevo objeto.
5. Elabore algoritmo que calcule y retorne el factorial de un número de alta precisión. El resultado debe ser otro objeto de la clase alta precisión.

MODULO 6

MANEJO DINÁMICO DE MEMORIA (VARIACIONES EN LISTAS SIMPLEMENTE LIGADAS)

Introducción

Hasta aquí hemos trabajado con una clase de lista que hemos denominado lista simplemente ligada, para la cual se ha definido una clase que hemos llamado LSL. Hay situaciones en las cuales es más beneficioso hacerle unas pequeñas modificaciones a las listas con el fin de que algunas tareas sean más versátiles y eficientes. Trataremos en este módulo las variaciones que se pueden hacer.

Objetivos

1. Conocer los diferentes tipos de listas simplemente ligadas que se pueden construir.
2. Conocer y tratar las implicaciones de las diferentes variaciones.

Preguntas básicas

- 1.Cuál es la principal diferencia entre una lista simplemente ligada y una lista simplemente ligada circular?
2. Qué implicación tiene la circularidad de una lista en el algoritmo de recorrido?
3. Qué es un nodo cabeza?
4. Qué ventajas se obtienen al tener listas con nodo cabeza?

6.1 Diferentes tipos de listas ligadas y sus características: comencemos presentando gráficamente los diferentes tipos de lista que se pueden construir:

Listas simplemente Ligadas.

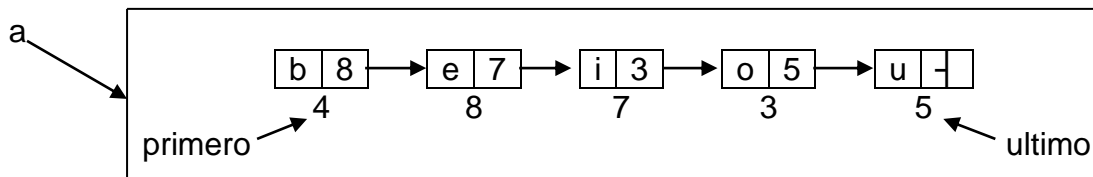


Figura 6.1

Listas simplemente ligadas circulares.

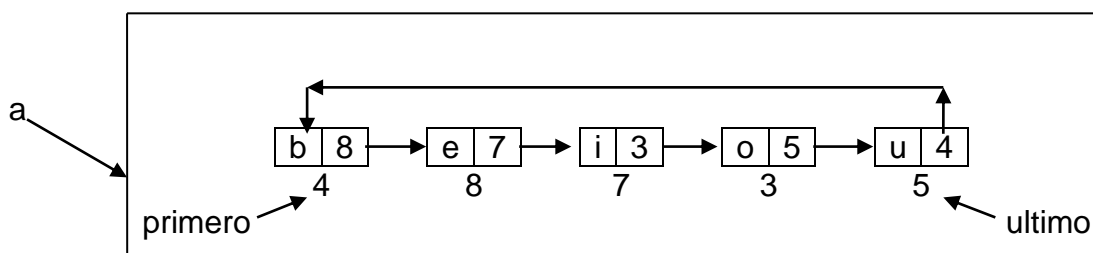


Figura 6.2

Listas simplemente ligadas circulares con nodo cabeza.

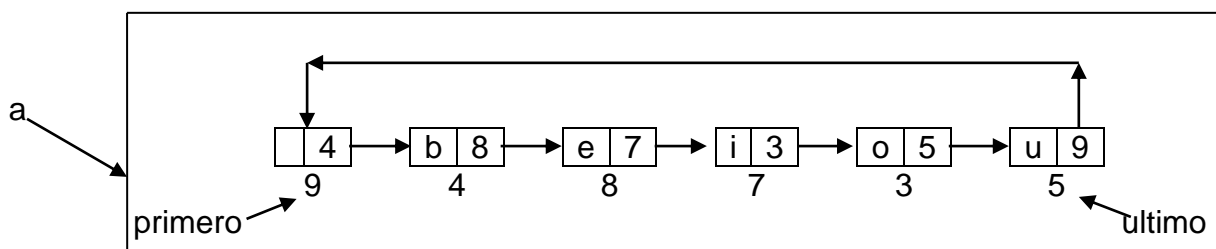


Figura 6.3

Listas simplemente ligadas con nodo cabeza.

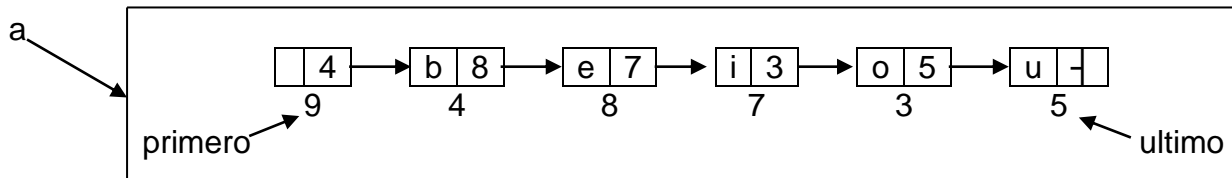


Figura 6.4

Las listas como la de la figura 6.1 son las que hemos venido trabajando hasta ahora. El campo de liga del último nodo es null.

La lista de la figura 6.3 es la misma que la de la figura 6.1, con la diferencia de que el campo de liga del último nodo vale 4, es decir, apunta hacia el primero. A las listas que tienen esta característica se les conoce como listas simplemente ligadas circulares.

La lista de la figura 6.3 es la misma que la de la figura 6.1, con la diferencia de que es circular y de que tiene un registro adicional al principio de la lista. A las listas que tienen estas dos modificaciones se les conoce como listas simplemente ligadas circulares con nodo cabeza.

La lista de la figura 6.4 es la misma que la de la figura 6.1, con la diferencia de que tiene un registro adicional al principio de la lista. A una lista con esta característica se le conoce como lista simplemente ligada con nodo cabeza.

Veamos ahora las implicaciones que tiene cada una de estas modificaciones en los algoritmos que hay que desarrollar.

6.2 Listas simplemente ligadas circulares: comencemos considerando el algoritmo que desarrollamos para el método `recorre` en la clase LSL.

```
1. void  recorrer()
2.      nodoSimple  p
3.      p = primerNodo()
4.      while (no finDeRecorrido(p)) do
5.          write(p.retornaDato())
6.          p = p.retornaLiga()
7.      end(while)
8. fin(recorre)
```

El algoritmo `recorre` e imprime el contenido de cada nodo de la lista. Aquí se controla la situación de lista vacía, **`primero == null`**, con la misma instrucción con que se controla la situación de terminación de recorrido de la lista, **`p == null`**. Es decir, con una sola instrucción, la 4, **`while (no finDeRecorrido(p))`** se están controlando dos situaciones.

Resumiendo, cuando tenemos una lista simplemente ligada, las condiciones son:

lista vacía: `primero == null`
fin de recorrido: `p == null`

Consideremos ahora el caso de las listas simplemente ligadas circulares.

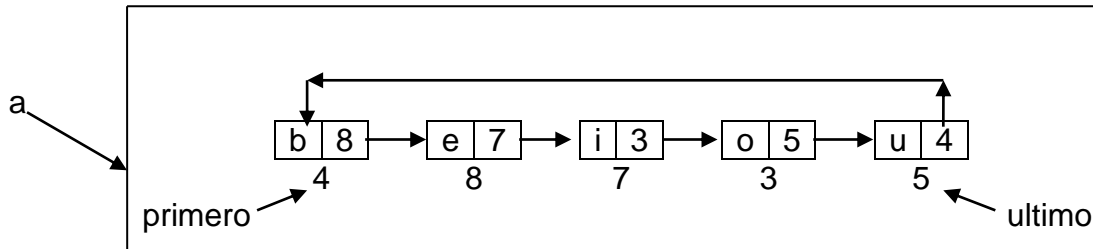


Figura 6.5

Para recorrer la lista de la figura 6.5 también se usa una variable auxiliar la cual llamamos **p**. Inicialmente **p** debe valer primero, se escribe el dato de **p** y se avanza con **p** hasta que **p** vuelva a valer primero. Un algoritmo como el que utilizamos para recorrer una lista simplemente ligada no nos sirve en este caso puesto que la condición de fin de recorrido ($p == \text{null}$) ya no es válida, además no se controla la situación de lista vacía: $\text{primero} == \text{null}$.

Plantear el ciclo de recorrido con la instrucción **while ($p \neq \text{primero}$)**, no sería correcto porque la primera instrucción es asignarle a la variable **p** el contenido de la variable **primero**, y entonces preguntar si **p** es diferente de **primero** daría como resultado falso y nunca entraría al ciclo. Por consiguiente debemos plantear un ciclo que permita entrar primero al ciclo y después preguntar por la condición de si **p** es diferente de **primero**. Para ello disponemos del ciclo **Do ... While (condición)**. Nuestro algoritmo es:

```

1. void  recorreListaCircular()
2.      nodoSimple  p
3.      p = primerNodo()
4.      do
5.          write(p.retornaDato())
6.          p = p.retornaLiga()
7.      while (no finDeRecorrido(p))
8. fin(recorreListaCircular)

```

Utilizamos aquí los métodos `primerNodo()` y `finDeRecorrido(p)`. El método `primerNodo()` puede ser perfectamente el definido para la clase LSL, pero el de `finDeRecorrido(p)` deberá ser diferente al definido en LSL. En este caso el método de `finDeRecorrido(p)` deberá ser: `return (primero == p)`. Además, lamentablemente este algoritmo no controla la situación de lista vacía. En caso de que este algoritmo sea invocado por un objeto cuya lista está vacía, cancela: cuando se ejecuta la instrucción 3 `p` queda valiendo `null` puesto que la lista está vacía, entra al ciclo planteado en la instrucción 4 y al ejecutar la instrucción 5 cancela, ya que intentará escribir el dato de un objeto nulo.

Recuerde que es supremamente importante controlar todas las situaciones que se puedan presentar. Aquí, para controlar la situación de lista vacía debemos agregar instrucciones a nuestro algoritmo. Esto incluye complejidad a nuestros algoritmos. **Para que se convenza intente elaborar el algoritmo de suma desarrollado en el módulo 5, con los números de alta precisión representados en listas simplemente ligadas circulares.**

Resumiendo, cuando tenemos una lista simplemente ligada, las condiciones son:

lista vacía: `primero == null`
fin de recorrido: `p == primero`

6.3 Lista simplemente ligada circular con nodo cabeza: consideremos ahora el caso de las listas simplemente ligadas circulares con nodo cabeza.

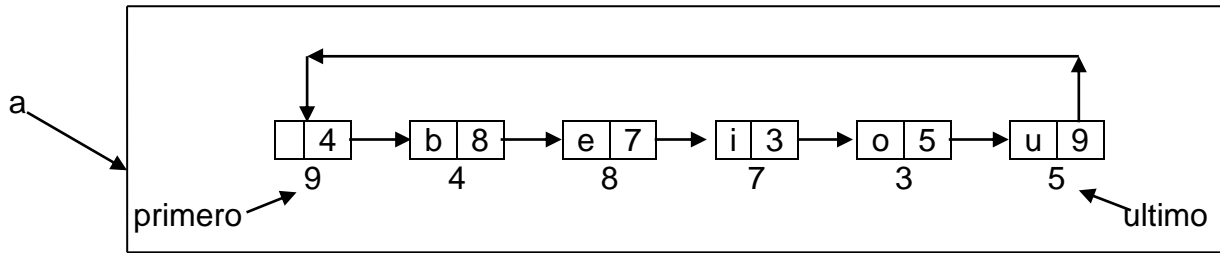
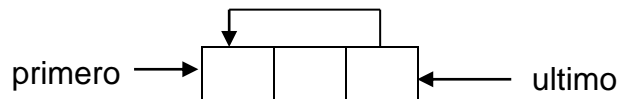


Figura 6.6

Como se puede observar, la lista tiene un nodo adicional al principio de la lista: el nodo 9. El nodo cabeza es un nodo que, por lo general, no tendrá información correspondiente al objeto que se esté representando en la lista ligada, **siempre** será el primer nodo de la lista ligada y facilita todas las operaciones sobre la lista: construir la lista, recorrer la lista, insertar un nodo y borrar un nodo. Realmente, si un nodo adicional presenta tantas ventajas, no debemos preocuparnos mucho por el hecho de consumir una posición más de memoria.

Comencemos considerando la situación de lista vacía, la cual es así:



El campo de liga del nodo cabeza apunta hacia sí mismo.

Vamos a definir una clase **Lista Simplemente Ligada Circular Con Nodo Cabeza (LSLCCRC)**, la cual será derivada de la clase **Lista Simplemente Ligada (LSL)**, por consiguiente podremos hacer uso de los métodos desarrollados en la clase **LSL**.

Clase **LSLCCRC** derivada de **LSL**

Público

```

    LSLCCRC()
    nodoSimple nodoCabeza()
    nodoSimple primerNodo()
    boolean finDeRecorrido(nodoSimple x)
    boolean esVacia()

```

fin(clase LSLCCRC)

Observe que estos métodos tienen los mismos nombres que los definidos en la clase base **LSL**. En tiempo de ejecución el computador distinguirá cuál utilizar dependiendo del objeto que invoque el método. Si el método es invocado por un objeto de la clase **LSL** ejecutará el método correspondiente a la clase **LSL**, pero si el método es invocado por un objeto de la clase **LSLCCRC** ejecutará el método correspondiente a la clase **LSLCCRC**.

LSLCCRC()

```

    primero = new nodoSimple(null)
    primero.asignaLiga(primero)
    ultimo = primero

```

fin(LSLCCRC)

nodoSimple nodoCabeza()

```

    return primero

```

fin(primerNodo)

nodoSimple primerNodo()

```

        return primero.retornaLiga()
    fin(primerNodo)

boolean finDeRecorrido(nodoSimple x)
    return x == primero
fin(finDeRecorrido)

boolean esVacía()
    return primero == primero.retornaLiga()
fin(esVacía)

```

El algoritmo correspondiente al método recorrer será el mismo que el definido para la clase LSL. Si el método recorrer() es invocado por un objeto de la clase LSL, cuando se ejecute el algoritmo invocará los métodos primerNodo() y finDeRecorrido(p) correspondientes a la clase LSL, pero si el método recorrer() es invocado por un objeto de la clase LSLCCRC, cuando se ejecute el algoritmo invocará los métodos primerNodo() y finDeRecorrido(p) correspondientes a la clase LSLCCRC.

Para recorrer la lista se comienza en el segundo nodo, ya que al ejecutar el método primerNodo() a **p** se le asigna **primero.retornaLiga()**. La lista se termina de recorrer cuando **p** sea igual a **primero**.

Si la lista está vacía, al ejecutar primerNodo() **p** queda valiendo **primero** y no entra al ciclo de recorrido. Nuevamente, con una sola condición estamos controlando dos situaciones: lista vacía y fin de recorrido.

Los métodos para conectar y desconectar un nodo de la lista se simplifican puesto que el parámetro **y** nunca será null, por consiguiente no se requiere controlar esta situación en dichos algoritmos. Éstos quedan así:

```

16.void conectar(nodoSimple x, nodoSimple y)
17.    x.asignaLiga(y.retornaLiga())
18.    y.asignaLiga(x)
19.    if (y == ultimo) then
20.        ultimo = x
21.    end(if)
22. fin(conectar)

14.void desconectar(nodoSimple x, nodoSimple y)
15.    y.asignaLiga(x.retornaLiga())
16.    if (x == ultimo) then
17.        ultimo = y
18.    end(if)
19. fin(desconectar)

```

En aras de la reutilización perfectamente podemos dejar la clase LSLCCRC tal como la hemos definido y los únicos métodos con los cuales se haría el polimorfismo dinámico son los que definimos en la clase LSLCCRC. De los métodos desarrollados para la clase LSL habría que modificar el algoritmo correspondiente a anterior(x), de tal manera que funciones apropiadamente para ambas clases, puesto que así como está sólo funciona correctamente para la clase LSL. La elaboración de este algoritmo se deja como ejercicio al estudiante.

6.4 Lista simplemente ligada con nodo cabeza: Tal como vimos en el apartado 6.1 también se puede tener lista simplemente ligada con nodo cabeza.

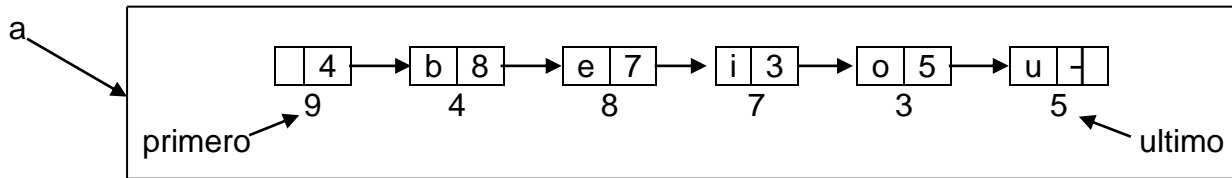


Figura 6.7

En este caso se tiene nodo cabeza, mas no la circularidad. La lista vacía es simplemente el nodo cabeza con su campo de liga en null. Las operaciones de inserción y borrado pueden quedar exactamente como en la clase LSLCCRC, pero la condición de fin de recorrido es como en la clase LSL.

El caso es, que de acuerdo al problema en particular que se quiera desarrollar utilizando listas ligadas, se tomará la decisión de cuál tipo de representación utilizar.

EJERCICIOS PROPUESTOS

1. Elabore algoritmos para los diferentes métodos correspondientes al proceso de inserción en lista simplemente ligadas circulares.
2. Elabore algoritmos para los diferentes métodos correspondientes al proceso de inserción en lista simplemente ligadas circulares con nodo cabeza.
3. Elabore algoritmos para los diferentes métodos correspondientes al proceso de inserción en lista simplemente ligadas con nodo cabeza.
4. Elabore algoritmos para los diferentes métodos correspondientes al proceso de borrado en lista simplemente ligadas circulares.
5. Elabore algoritmos para los diferentes métodos correspondientes al proceso de borrado en lista simplemente ligadas circulares con nodo cabeza.
6. Elabore algoritmos para los diferentes métodos correspondientes al proceso de borrado en lista simplemente ligadas con nodo cabeza.
7. Elabore 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 LSLCCRC.
8. Desarrolle cada uno de los ejercicios propuestos en el módulo 4 para cada uno de los tipos de listas definidos aquí: lista simplemente ligada circular, lista simplemente ligada circular con nodo cabeza y lista simplemente ligada con nodo cabeza.

MODULO 7

MANEJO DINÁMICO DE MEMORIA (NODO DOBLE)

Introducción

Hasta aquí hemos tratado con listas ligadas que sólo se pueden recorrer en un solo sentido: de izquierda a derecha. Existen cierto tipo de problemas que exigen que las listas ligadas se puedan recorrer en ambas direcciones: de izquierda a derecha y de derecha a izquierda. Con este propósito se han construido las listas doblemente ligadas. Para ello debemos definir una nueva clase de nodo, el cual llamaremos nodo doble.

Objetivos

1. Aprender la clase nodo doble.
2. Comprender las ventajas de la clase nodo doble.
3. Manipular correctamente los campos de liga de un nodo doble.

Preguntas básicas

1. Hacia cuál nodo apunta el campo de liga izquierda de un nodo doble?
2. Hacia cuál nodo apunta el campo de liga derecha de un nodo doble?
3. Cuáles son las ventajas de tener un nodo con dos campos de liga?
4. En qué consiste la propiedad fundamental de las listas doblemente ligadas?

7.1 Definición y características: Un nodo doble es un registro que tiene dos campos de liga: uno que llamaremos **Li** (Liga izquierda) y otro que llamaremos **Ld** (Liga derecha). Un dibujo para representar dicho nodo es:

Liga izquierda	Area de datos	Liga derecha
-------------------	------------------	-----------------

donde

Li: apunta hacia el nodo anterior.

Ld: apunta hacia el nodo siguiente.

Con base en esto vamos a definir una clase que llamaremos nodoDoble.

Clase **nodoDoble**

Privado

Objeto dato

nodoDoble Li, Ld

Público

nodoDoble(objeto d) // constructor

void asignaDato(objeto d)

void asignaLd(nodoDoble x)

void asignaLi(nodoDoble x)

objeto retornaDato()

nodoDoble retornaLd()

nodoDoble retornaLi()

fin(clase nodoDoble)

Los algoritmos correspondientes a cada uno de estos métodos son los siguientes:

```
nodoDoble(objeto d) // Constructor
    dato = d
    Ld = null
```

```

    Li = null
fin(nodoDoble)

void asignaDato(objeto d)
    dato = d
fin(asignaDato)

void asignaLd(nodoDoble x)
    Ld = x
fin(asignaLd)

void asignaLi(nodoDoble x)
    Li = x
fin(asignaLi)

objeto retornaDato()
    return dato
fin(retornaDato)

nodoDoble retornaLi()
    return Li
fin(retornaLi)

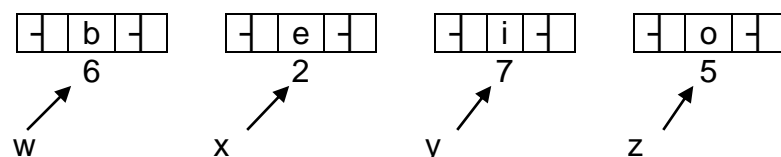
nodoDoble retornaLd()
    return Ld
fin(retornaLd)

```

Teniendo definida esta clase, procedamos a desarrollar un algoritmo en el cual hagamos uso de ella y de sus métodos. Consideremos que se tiene el siguiente conjunto de datos: b, e, i, o, u y que nuestro algoritmo los leerá en ese orden.

1. nodoDoble w, x, y, z
2. read(d)
3. w = new nodoDoble(d)
4. read(d)
5. x = new nodoDoble(d)
6. read(d)
7. y = new nodoDoble(d)
8. read(d)
9. z = new nodoDoble(d)

al ejecutar estas instrucciones el escenario que se obtiene es el siguiente:

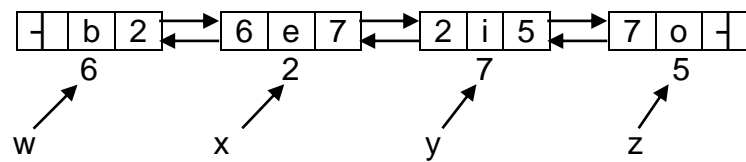


Continuemos con nuestro algoritmo con estas instrucciones:

10. w.asignaLd(x)
11. x.asignaLi(w)
12. x.asignaLd(y)
13. y.asignaLi(x)
14. y.asignaLd(z)

15. z.asignaLi(y)

Al ejecutar estas instrucciones nuestro escenario es:

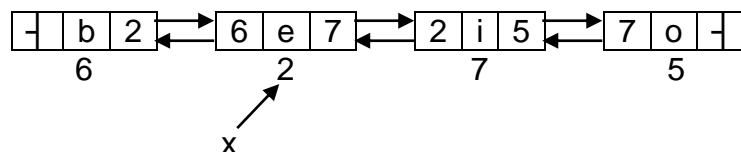


Fíjese que el campo **Ld** del nodo **w** quedó valiendo 2, en virtud de la instrucción 10, indicando que el nodo siguiente a **w** es el 2; el campo **Li** de **x** (el nodo 2) quedó valiendo 6, en virtud de la instrucción 11, indicando que el nodo anterior a **x** es el 6; el campo **Ld** del nodo **x** quedó valiendo 7, en virtud de la instrucción 12, indicando que el nodo siguiente a **x** es el 7. De una forma similar se han actualizado los campos **Li** y **Ld** de los nodos **y** y **z**. Asegúrese de entender bien el escenario actual.

Continuemos ejecutando las siguientes instrucciones:

- 16. w = null
- 17. y = null
- 18. z = null

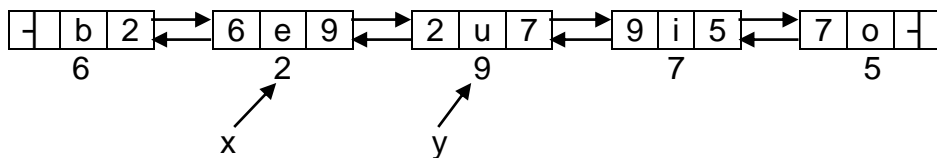
ya nuestra situación es:



Ahora ejecutemos estas:

- 19. read(d)
- 20. y = new nodoDoble(d)
- 21. y.asignaLd(x.retornaLd())
- 22. y.asignaLi(x)
- 23. y.retornaLd().asignaLi(y)
- 24. x.asignaLd(y)

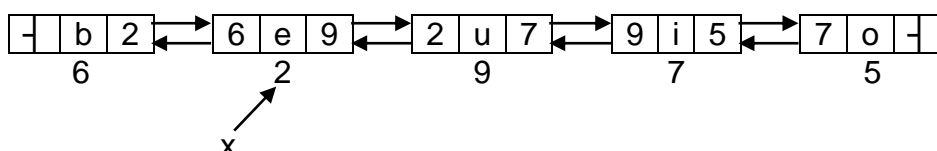
la situación queda:



Ahora, al ejecutar esta instrucción

- 25. y = null

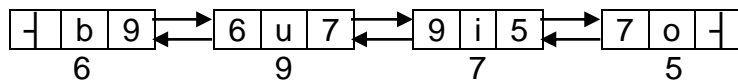
Las cosas quedan:



Por último, ejecutemos estas otras:

```
26. x.retornaLi().asignaLd(x.retornaLd())
27. x.retornaLd().asignaLi(x.retornaLi())
```

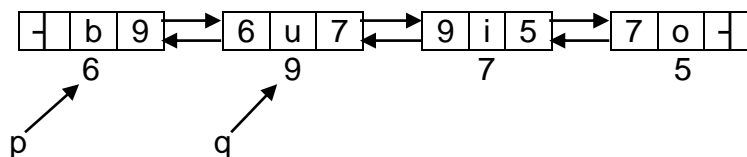
y nuestros nodos quedan:



Fíjese que el resultado de ejecutar las instrucciones 26 y 27 fue eliminar el nodo x. Unas instrucciones que harían la misma tarea que las instrucciones 26 y 27 son:

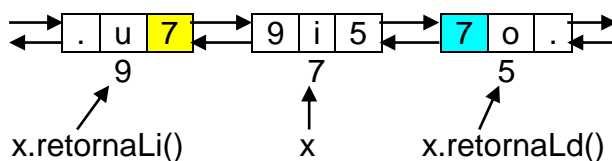
1. p = x.retornaLi()
2. q = x.retornaLd()
3. p.asignaLd(q)
4. q.asignaLi(p)

Si lo hubiéramos hecho de este modo la lista quedaba así:



Es supremamente importante entender que las instrucciones 26 y 27 ejecutan la misma tarea que las instrucciones 1, 2, 3 y 4 mostradas a continuación de ellas.

7.2 Propiedad fundamental de las listas doblemente ligadas. En el siguiente módulo veremos la clase lista doblemente ligada, la cual se construye con objetos de la clase nodo doble. Es importante ver aquí lo que se conoce como la propiedad fundamental de las listas doblemente ligadas ya que está basada en las características de los campos de liga de los nodos dobles. Consideremos los siguientes 3 registros consecutivos:



El campo de liga izquierda del nodo x vale 9, indicando que el nodo anterior es el nodo 9, por tanto el nodo 9 es el nodo x.retornaLi().

El campo de liga derecha del nodo x vale 5, indicando que el nodo siguiente es el nodo 5, por tanto, el nodo 5 es el nodo x.retornaLd().

El nodo x.retornaLi() tiene un campo de liga derecha, el cual vale 7, es decir, el mismo nodo x.

El nodo x.retornaLd() tiene un campo de liga izquierda, el cual también vale 7, es decir, el nodo x.

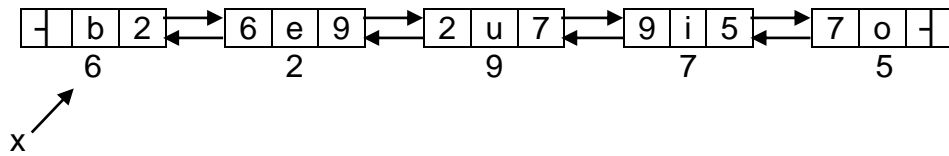
Dado lo anterior, se tiene que:

$$x.retornaLi().retornaLd() == x == x.retornaLd().retornaLi()$$

Esta característica se conoce como la propiedad fundamental de las listas doblemente ligadas, y es de suprema importancia entenderla bien con el fin de manipular apropiadamente los objetos de la clase lista doblemente ligadas que se verá en el siguiente módulo.

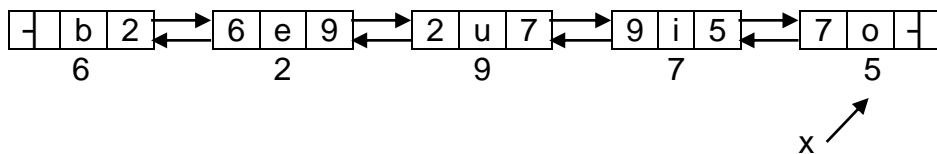
EJERCICIOS PROPUESTOS

1. Dados los siguiente nodos conectados y la variable **x** apuntando hacia el nodo 6



- Escriba algoritmo que imprima el dato del nodo 7.
- Escriba algoritmo que desconecte el nodo 9.
- Escriba algoritmo que inserte un nuevo nodo con dato "q" a continuación del nodo 5.
- Escriba algoritmo que escriba los datos de todos los nodos dibujados.

2. Dados los siguiente nodos conectados y la variable **x** apuntando hacia el nodo 5



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

MÓDULO 8

MANEJO DINÁMICO DE MEMORIA (LISTAS DOBLEMENTE LIGADAS)

Introducción

Ya hemos visto listas que sólo se pueden recorrer en un solo sentido ya que están construidas con objetos de la clase `nodoSimple`. En el módulo anterior vimos una nueva clase de nodo que llamamos `nodoDoble`. Veremos en este módulo listas que se construyen con objetos de esta última clase: listas doblemente ligadas.

Objetivos

1. Construir listas doblemente ligadas.
2. Definir objetos de clase lista doblemente ligada.
3. Manipular objetos de la clase lista doblemente ligada.
4. Aprender las variaciones de la clase lista doblemente ligada.

Preguntas básicas

- 1.Cuál es la principal diferencia entre objetos de la clase lista simplemente ligada y objetos de la clase lista doblemente ligada?
2. De cuántas formas se puede recorrer una lista doblemente ligada?
3. Qué situaciones diferentes se pueden presentar al conectar un nuevo nodo en una lista doblemente ligada?
4. Qué situaciones diferentes se pueden presentar al desconectar un nodo en una lista doblemente ligada?
- 5.Cuál es el orden de magnitud de los procesos de conectar y desconectar nodos en una lista doblemente ligada?

8.1 Clase lista doblemente ligada: comenzamos definiendo la clase lista doblemente ligada. La llamaremos **LDL**.

Clase LDL

Privado

`nodoDoble primero, ultimo`

Público

```
LDL() // constructor
nodoDoble primerNodo()
nodoDoble ultimoNodo()
boolean finDeRecorrido(nodoDoble x)
boolean esVacía()
void recorrelzqDer()
void recorredelzq()
nodoDoble buscaDondeInsertar(objeto d)
void insertar(objeto d, nodoDoble y)
void conectar(nodoDoble x, nodoDoble y)
nodoDoble buscarDato(objeto d)
void borrar(objeto d)
void desconectar(nodoDoble x)
```

`fin(clase LDL)`

Los métodos que definimos acá son los mismos que hemos definido para la clase LSL, ya que las operaciones básicas que se pueden efectuar con objetos de una lista son los mismos. La diferencia está en que su implementación es distinta.

Los algoritmos para los métodos definidos son:

```
LDL() // Constructor
```

```
    primero = ultimo = null  
Fin(LDL)
```

```
nodoDoble primerNodo()  
    return primero  
fin(primerNodo)
```

```
nodoDoble ultimoNodo()  
    return ultimo  
fin(ultimoNodo)
```

```
boolean finDeRecorrido(nodoDoble x)  
    return x == null  
fin(finDeRecorrido)
```

```
boolean esVacia()  
    return primero == null  
fin(esVacia)
```

```
void recorrelzqDer()  
    nodoDoble p  
    p = primerNodo()  
    while (no finDeRecorrido(p)) do  
        write(p.retornaDato())  
        p = p.retornaLd()  
    end(while)  
fin(recorrelzqDer)
```

```
void recorredrIzq()  
    nodoDoble p  
    p = ultimoNodo()  
    while (no finDeRecorrido(p)) do  
        write(p.retornaDato())  
        p = p.retornaLi()  
    end(while)  
fin(recorredrIzq)
```

Los algoritmos para recorrer una lista doblemente ligada son casi idénticos al algoritmo para recorrer una lista simplemente ligada. Sólo se debe tener en cuenta el nodo con el cual se inicia el recorrido y el campo de liga que se debe utilizar cuando se vaya a avanzar sobre la lista: para movernos de izquierda a derecha utilizamos el campo liga derecha de **p** y para movernos de derecha a izquierda utilizamos el campo liga izquierda de **p**.

8.2 Proceso de inserción. Recordemos que nuestro proceso de inserción lo tratamos con tres métodos: buscar dónde insertar, insertar y conectar. Los algoritmos son:

```
nodoDoble buscaDondeInsertar(objeto d)  
    nodoDoble p, y  
    p = primerNodo()  
    y = anterior(p)  
    while (no finDeRecorrido(p) and p.retornaDato() < d) do  
        y = p  
        p = p.retornaLd()  
    end(while)
```

```

return y
fin(buscaDondeInsertar)

```

El algoritmo para el método buscar dónde insertar es casi idéntico al de la clase **LSL**. La única diferencia es la instrucción con la que avanzamos con **p**. Nuestro método retorna la dirección del nodo a continuación del cual hay que insertar un nuevo nodo con dato **d**.

```

void insertar(objeto d, nodoDoble y)
    nodoDoble x, p
    x = nuevo nodoDoble(d)
    conectar(x, y)
fin(insertar)

```

Nuestro método insertar simplemente consigue un nuevo nodoDoble, lo carga con el dato **d** e invoca el método conectar. En el algoritmo para conectar un nodo doble en una lista doblemente ligada se presentan tres situaciones: que haya que conectarlo al principio de la lista, que haya que conectarlo al final de la lista o que haya que conectarlo en un sitio intermedio.

8.2.1 Inserción en sitio intermedio. Consideremos la lista de la figura 8.1 y que se desea insertar la letra 'e'.

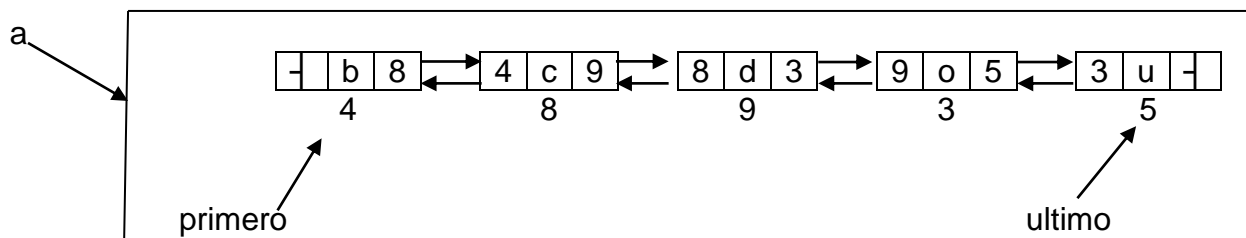


Figura 8.1

Al ejecutar la instrucción

```
y = buscaDondeInsertar('e')
```

la variable **y** queda valiendo 9, y al ejecutar la instrucción **insertar('e', y)** quedamos en la situación correspondiente a la figura 8.2 considerando que el nodo obtenido fue el 6.

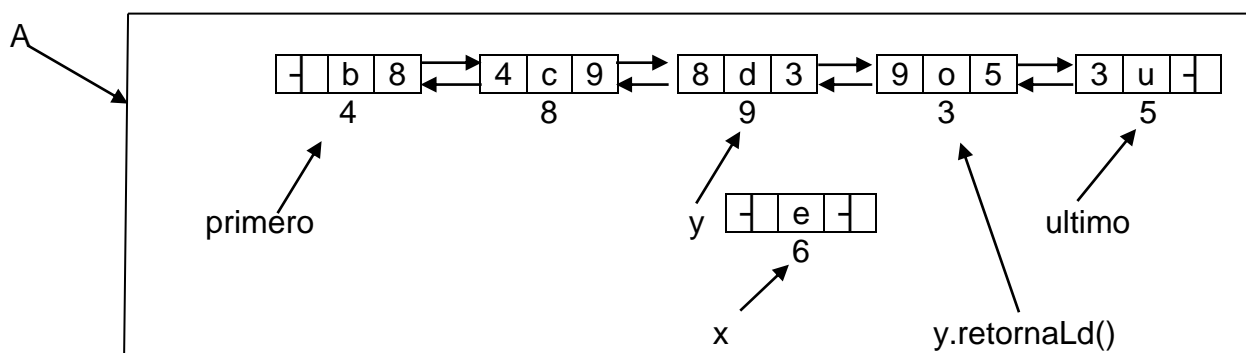


Figura 8.2

Para conectar **x** a continuación de **y** se deben modificar cuatro campos de liga: los campos liga izquierda y liga derecha del nodo **x**, el campo liga derecha del nodo **y** y el campo liga izquierda del nodo **y.retornaLd()**.

El campo liga izquierda del nodo **x** debe quedar valiendo 9: **x.asignaLi(y)**.

El campo liga derecha del nodo **x** debe quedar valiendo 3: **x.asignaLd(y.retornaLd())**.

El campo liga derecha del nodo **y** debe quedar valiendo 6: **y.asignaLd(x)**. Esta instrucción conecta el nodo **x** para cuando se esté haciendo el recorrido de izquierda a derecha.

El campo liga izquierda del nodo **x.retornaLd()** (nodo 3) debe quedar valiendo 6: **x.retornaLd().asignaLi(x)**. Esta instrucción conecta el nodo **x** para cuando la lista se esté recorriendo de derecha a izquierda.

El conjunto de instrucciones es entonces:

```
x.asignaLi(y)
x.asignaLd(y.retornaLd())
y.asignaLd(x)
x.retornaLd().asignaLi(x)
```

Al ejecutar estas instrucciones la lista queda así:

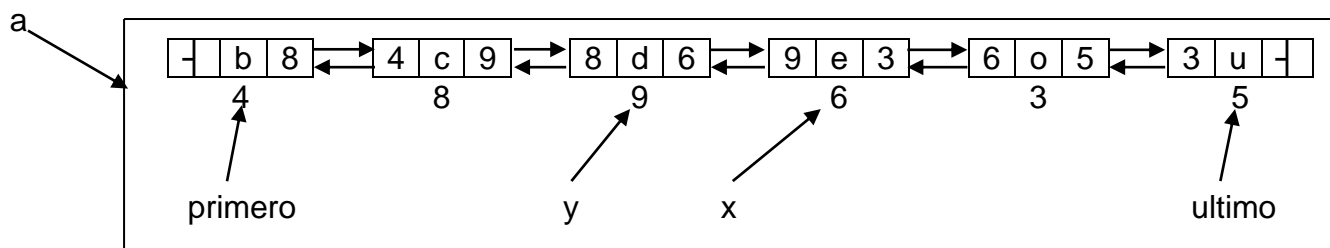


Figura 8.3

8.2.2 Inserción al final: si volvemos nuevamente a la figura 8.1 y lo que queremos es insertar la letra 'z', y ejecutamos **y = buscaDondelInsertar('z')**, **y** queda valiendo 5, y luego ejecutamos **insertar('z', y)** estamos en una situación como en la figura 8.4

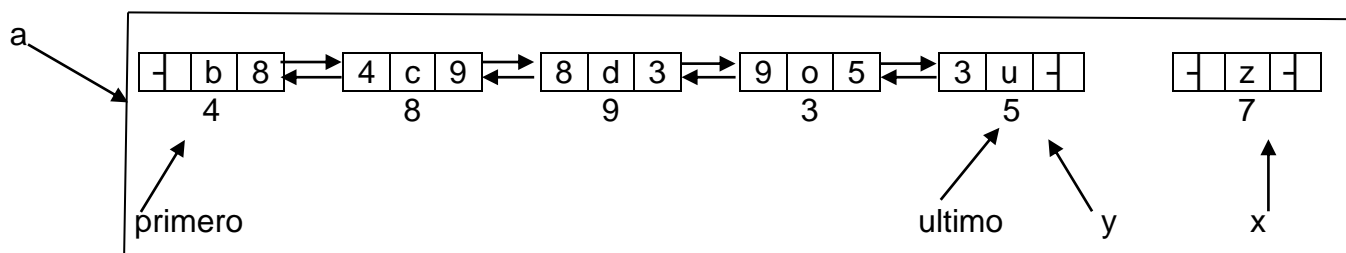


Figura 8.4

Aquí, lo único que se debe hacer es modificar el campo liga derecha de **y**, el campo liga izquierda de **x** y actualizar **último**. Estas instrucciones son:

```
y.asignaLd(x)
x.asignaLi(y)
ultimo = x
```

las cuales, al ejecutarlas conforman la lista como en la figura 8.5

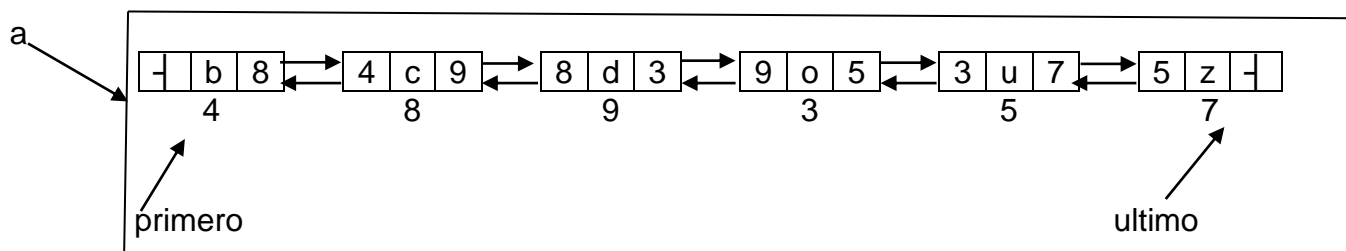


Figura 8.4

8.2.3 Inserción al principio: retomemos nuevamente la lista de la figura 8.1 y que se desea insertar la letra 'a'. Al ejecutar `y = buscaDondeInsertar('a')`, la variable **y** que da valiendo null, luego ejecutamos `insertar('a', y)` y estamos en una situación como en la figura 8.5

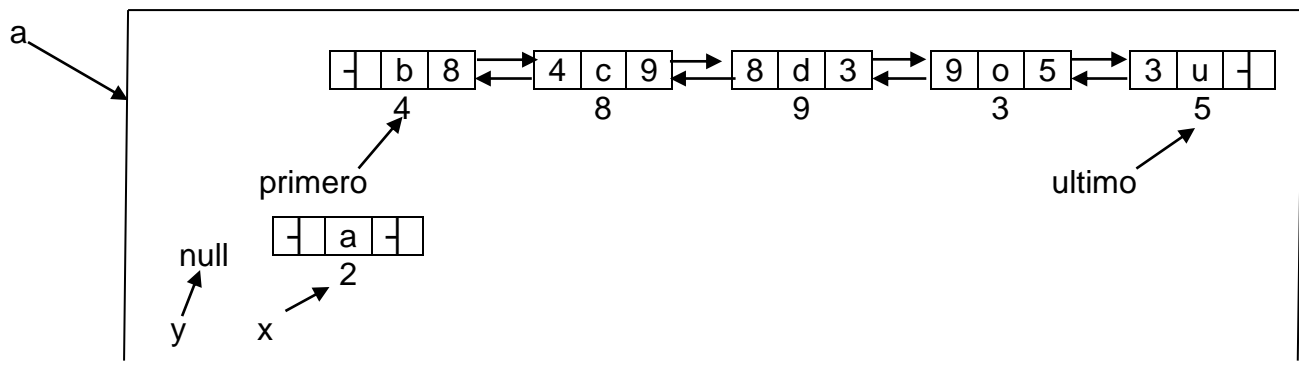


Figura 8.5

En esta situación lo que hay que hacer es modificar el campo liga izquierda de **primero**, el campo liga derecha de **x** y actualizar **primero** con **x**. Sin embargo, hay que tener cuidado con el hecho de que la lista pudo haber estado vacía, ya que si esa hubiera sido la situación, también hay que actualizar la variable **último** con **x**.

El conjunto de instrucciones correspondientes a esta situación es:

```

x.asignaLd(primero)
if (primero != null) then
    primero.asignaLi(x)
else
    ultimo = x
end(if)
primero = x

```

Agrupando los tres casos en un solo algoritmo, el cual corresponda al método que hemos llamado conectar tenemos.

```

void conectar(nodoDoble x, nodoDoble y)
    casos
        y == null:           // El nodo se conecta al principio
            x.asignaLd(primero)
            if (primero <> null) then
                primero.asignaLi(x)
            else
                ultimo = x
            end(if)
            primero = x

        y.retornaLd() == null: // El nodo se conecta al final
            y.asignaLd(x)
            x.asignaLi(y)

        else:               // El nodo se conecta en un sitio intermedio
            x.asignaLd(y.retornaLd())
            x.asignaLi(y)
            x.retornaLd().asignaLi(x)
            y.asignaLd(x)

    fin(casos)
fin(conectar)

```

8.3 Proceso de borrado. Consideremos ahora la tarea de eliminar un nodo de una lista doblemente ligada. Para este proceso lo primero es buscar el nodo que contiene el dato a borrar. Dicho método lo hemos llamado buscarDato(d). El algoritmo se presenta a continuación.

```

nodoDoble  buscarDato(objeto d)
    nodoSimple x
    x = primerNodo()
    while (no finDeRecorrido(x) and x.retornaDato() != d) do
        x = x.retornaLd()
    end(while)
    return x
fin(buscarDato)

```

Como la lista es doblemente ligada basta con determinar el nodo que contiene el dato **d**, ya que los campos liga izquierda y liga derecha dan la información correspondiente al nodo anterior y al nodo siguiente, nodos necesarios para desconectar el nodo que contiene el dato **d**. Si el dato **d** no se halla en la lista nuestro método buscarDato(d) retorna null. Luego de hallar el nodo que contiene el dato **d** se ejecuta el método que hemos llamado borrar, el cual, simplemente controla que el dato exista. En caso de que exista invoca el método desconectar, de lo contrario produce el mensaje apropiado y retorna al programa llamante. Nuestro algoritmo para el método borrar es:

```

void borrar(nodoDoble x)
    if (x == null) then
        write("dato a borrar no existe")
        return
    end(if)
    desconectar(x)
fin(borrar)

```

Consideremos ahora la tarea para desconectar un nodo de una lista doblemente ligada. Se presentan aquí también tres situaciones: desconectar un nodo intermedio, desconectar el último nodo de la lista y desconectar el nodo que está de primero en la lista.

8.3.1 Desconecta nodo intermedio. Consideremos la lista de la figura 8.6 y que se desea borrar la letra 'd'.

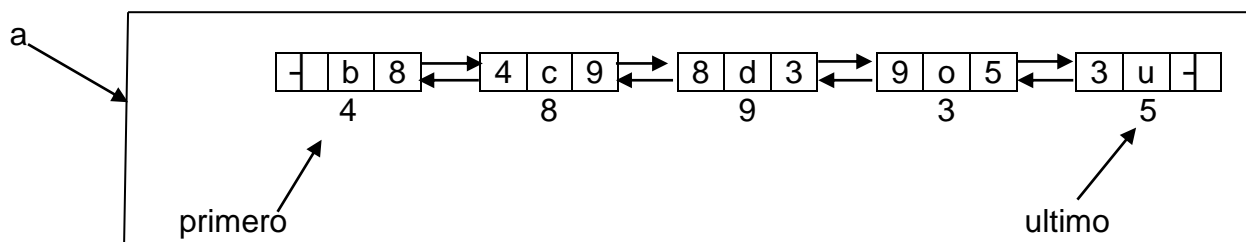


Figura 8.6

Al ejecutar la instrucción `x = buscarDato('d')` la variable `x` queda valiendo 9, y estamos en una situación como la mostrada en la figura 8.7.

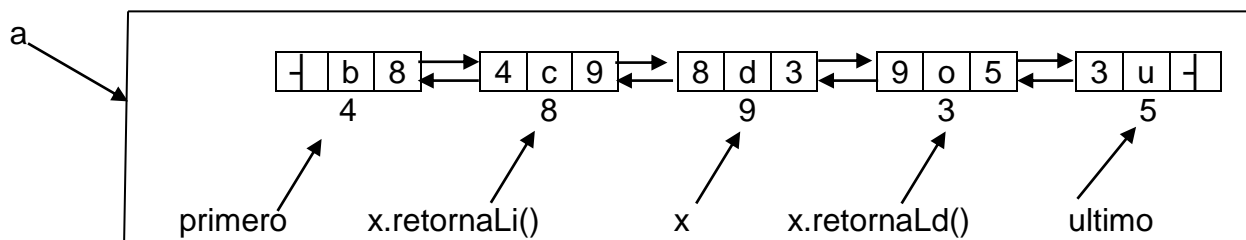


Figura 8.7

Hay que borrar el nodo **x == 9**, debemos tener en cuenta que la lista es doblemente ligada, por consiguiente el nodo anterior es el nodo **x.retornaLi()** y el nodo siguiente es el nodo **x.retornaLd()**.

Para desconectar el nodo **x** en sentido de izquierda a derecha lo que hay que hacer es que cuando estemos en el nodo **x.retornaLi()** debemos pasarnos hacia el nodo **x.retornaLd()**, o sea brincarnos el nodo **x**. Por consiguiente, debemos actualizar el campo de liga derecha del nodo **x.retornaLi()**. La instrucción para efectuar esto es: **x.retornaLi().asignaLd(x.retornaLd())**, la cual se lee: el objeto **x.retornaLi()** invoca el método **asignaLd()** y le envía como parámetro el objeto **x.retornaLd()**.

Si estamos recorriendo de derecha a izquierda, cuando estemos ubicados en el nodo **x.retornaLd()**, debemos pasarnos hacia el nodo **x.retornaLi()**, o sea desconectar el nodo **x**. La instrucción para efectuar esto es: **x.retornaLd().asignaLi(x.retornaLi())**, la cual se lee: el objeto **x.retornaLd()** invoca el método **asignaLi()** y le envía como parámetro el objeto **x.retornaLi()**.

Al ejecutar estas dos instrucciones la lista queda como en la figura 8.8

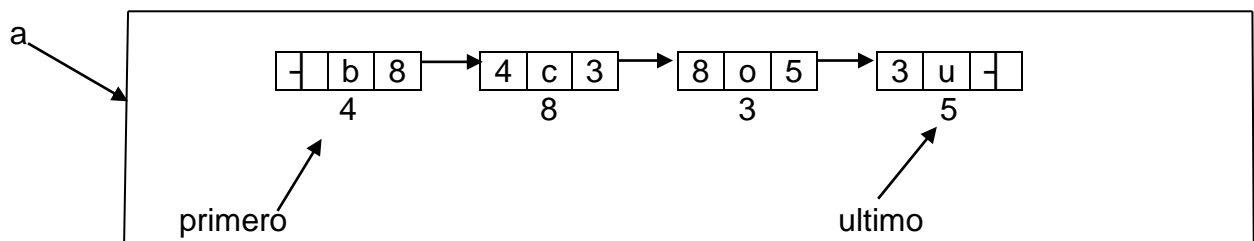


Figura 8.8

8.3.2 Desconecta último nodo. Si hacemos nuevamente referencia a la figura 8.6 y deseamos borrar la 'u', al ejecutar la instrucción **x = buscarDato('u')** quedamos en una situación como la mostrada en la figura 8.9

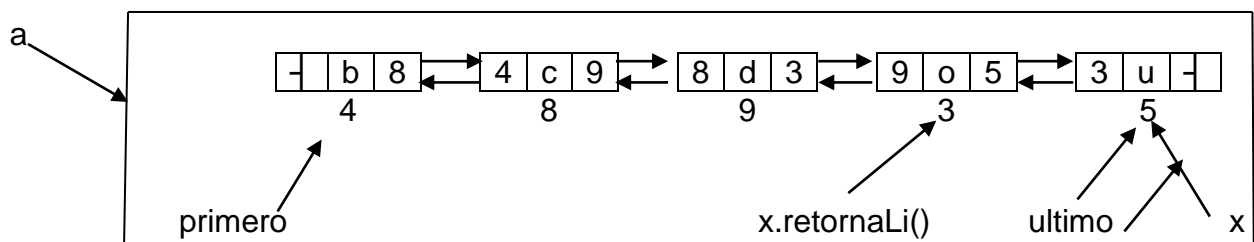


Figura 8.9

En este caso lo único que hay que hacer es modificar el campo liga derecha del nodo **x.retornaLi()**, el cual debe quedar valiendo null, y actualizar **ultimo**. Las instrucciones para estas dos modificaciones son:

```
x.retornaLi().asignaLd(null)  
ultimo = x.retornaLi()
```

Al ejecutar estas dos instrucciones la lista queda como en la figura 8.10.

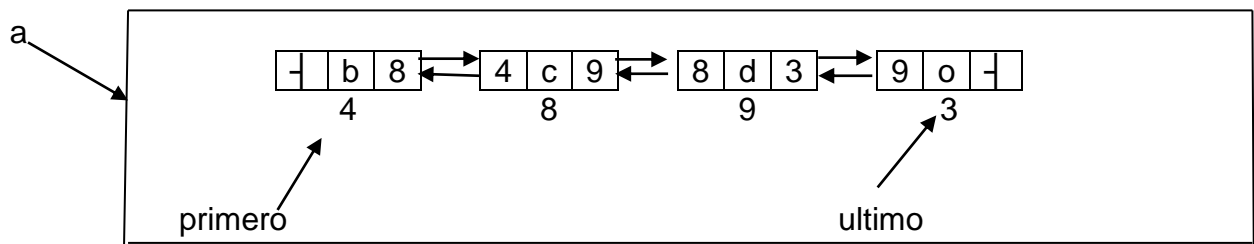


Figura 8.10

8.3.3 Desconecta primer nodo. Remitámonos nuevamente a la figura 8.6, y que se desea borrar la 'b'. Al ejecutar la instrucción `x = buscarDato('b')` estamos en una situación como en la figura 8.11.

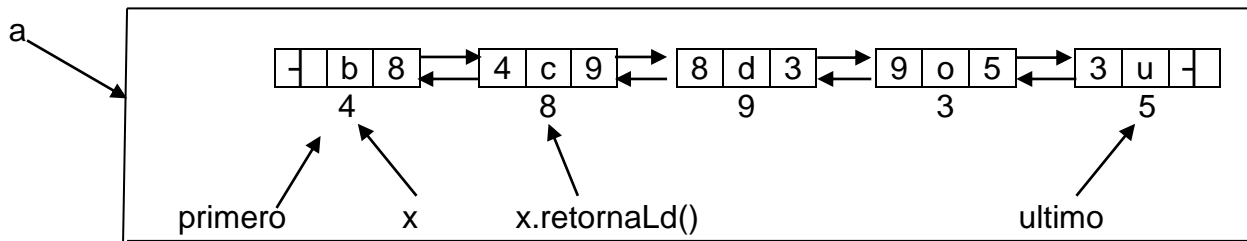


Figura 8.11

Lo único que hay que hacer aquí es actualizar el campo liga izquierda del nodo `x.retornaLd()`, el cual debe quedar en null, y el contenido de la variable `primero`. Para ello debemos ejecutar las instrucciones

```
primero = x.retornaLd()
primero.asignaLi(null)
```

Sin embargo, debemos considerar el hecho de que la lista en la cual hay que borrar el primer nodo sólo tiene un nodo, en cuyo caso también hay que actualizar el contenido de la variable `ultimo`. Es decir, el conjunto de instrucciones debe ser:

```
primero = x.retornaLd()
if (primero == null) then
    ultimo = null
else
    primero.asignaLi(null)
end(if)
```

Agrupando las tres situaciones en un solo algoritmo obtenemos nuestro algoritmo para el método desconectar.

```
void desconectar(nodoDoble x)
    casos
        x.retornaLi() == null:      // El nodo a desconectar es el primero
            primero = x.retornaLd()
            if (primero == null) then
                ultimo = null
            else
                primero.asignaLi(null)
            end(if)

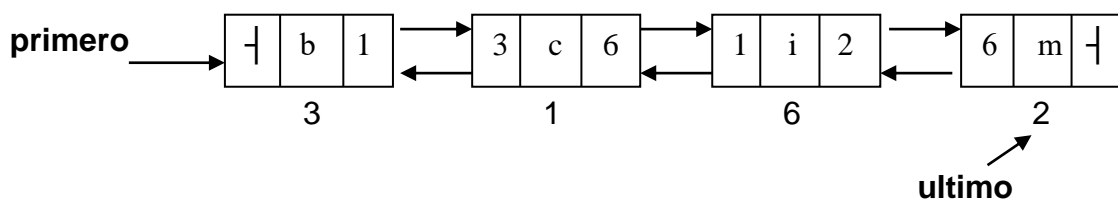
        x.retornaLd() == null:    // El nodo a borrar es el último
            ultimo = x.retornaLi()
            ultimo.asignaLd(null)

        else:                      // El nodo a borrar es intermedio
            x.retornaLd().asignaLi(x.retornaLi())
            x.retornaLi().asignaLd(x.retornaLd())

    fin(casos)
fin(borrar)
```

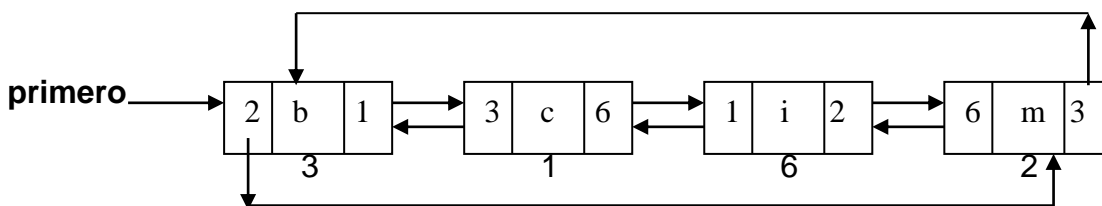
8.4 Variaciones en listas doblemente ligadas. Al igual que las listas simplemente ligadas, con las listas doblemente ligadas se pueden presentar una serie de variaciones:

8.4.1 Listas doblemente ligadas.



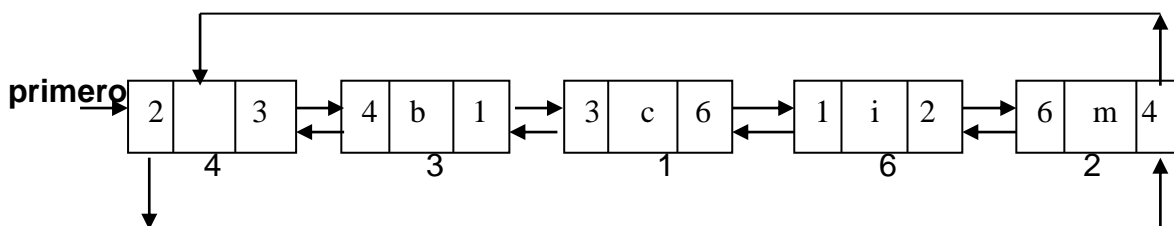
El campo de liga izquierda del primer nodo es **null** porque el primer nodo no tiene anterior, y el campo de liga derecha del último nodo también vale **null** porque el último nodo no tiene siguiente.

8.4.2 Listas doblemente ligadas circulares:

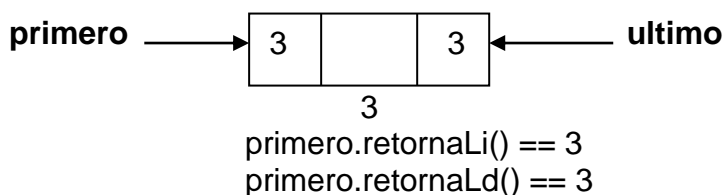


En las listas doblemente ligadas circulares el campo de liga derecha del último nodo apunta hacia el primer nodo de la lista y el campo de liga izquierda del primer nodo apunta hacia el último nodo de la lista.

8.4.3 Listas doblemente ligadas circulares con nodo cabeza:

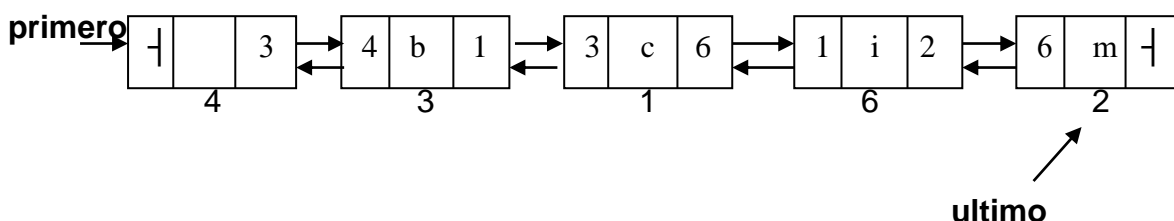


En las lista doblemente ligadas circulares con nodo cabeza se tiene la ventaja de que la representación de la lista vacía tendrá como mínimo el nodo cabeza:



Esta es la configuración de la lista vacía cuando se tienen listas doblemente ligadas circulares con nodo cabeza.

8.4.4 Listas doblemente ligadas no circulares con nodo cabeza.



Como en el caso de las listas simplemente ligadas consideraremos la clase **Lista Doblemente Ligada Circular Con Nodo Cabeza (LDLCCRC)** como derivada de la clase **Lista Doblemente Ligada (LDL)**.

Clase **LDLCCRC** derivada de **LDL**

```
Público
    LDLCCRC()                // constructor
    nodoDoble cabeza()
    nodoDoble primerNodo()
    boolean finDeRecorrido(nodoDoble x)
    boolean esVacía()
fin(clase LDLCCRC)
```

Los algoritmos correspondientes a los métodos anteriores son:

LDLCCRC()

```
    primero = new nodoDoble(null)
    primero.asignaLi(primero)
    primero.asignaLd(primero)
fin(LDLCCRC)
```

```
nodoDoble cabeza()
    return primero
fin(primerNodo)
```

```
nodoDoble primerNodo()
    return primero.retornaLd()
fin(primerNodo)
```

```
nodoDoble ultimoNodo()
    return ultimo
fin(ultimoNodo)
```

```
boolean finDeRecorrido(nodoDoble p)
    return p == primero
fin(finDeRecorrido)
```

```
boolean esVacía()
    return primero == primero.retornaLd()
fin(esVacía)
```

EJERCICIOS PROPUESTOS

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.
2. 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 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.
3. Elabore un algoritmo que intercambie los registros de una lista ligada así: el primero con el segundo, el tercero con el cuarto, el quinto con el sexto y así sucesivamente.
4. Elabore algoritmo que intercambie los registros de una lista ligada así: el primero con el último, el segundo con el penúltimo, el tercero con el antepenúltimo, y así sucesivamente.

5. Elabore algoritmo que intercambie los registros de una lista 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.
6. Elabore algoritmos para los diferentes métodos correspondientes al proceso de inserción en lista doblemente ligadas circulares.
7. Elabore algoritmos para los diferentes métodos correspondientes al proceso de inserción en lista doblemente ligadas circulares con nodo cabeza.
8. Elabore algoritmos para los diferentes métodos correspondientes al proceso de inserción en lista doblemente ligadas con nodo cabeza
9. Elabore algoritmos para los diferentes métodos correspondientes al proceso de borrado en lista doblemente ligadas circulares.
10. Elabore algoritmos para los diferentes métodos correspondientes al proceso de borrado en lista doblemente ligadas circulares con nodo cabeza.
11. Elabore algoritmos para los diferentes métodos correspondientes al proceso de borrado en lista doblemente ligadas con nodo cabeza.

MODULO 9

HILERAS: DEFINICIÓN, OPERACIONES, APLICACIONES.

Introducción

En los primeros lenguajes de programación las hileras aparecían sólo como constantes que se utilizaban únicamente como entrada y salida. No había operaciones sobre ellas. A medida que se desarrollaron los lenguajes de programación se presentó la necesidad de construir reconocedores de léxico, interpretación de hileras, modificaciones, y en general operaciones sobre ellas, máxime cuando comenzaron a aparecer los procesadores de texto. Aquí veremos cómo manipular estos datos no numéricos.

Objetivos

1. Definir una clase hilera.
2. Definir e interpretar las operaciones a efectuar sobre hileras.
3. Aplicar las operaciones definidas para objetos de la clase hilera.

Preguntas básicas

1. Qué es una hilera.
2. Cuáles son las operaciones sobre hileras.
3. Qué retorna la función longitud de la clase hilera.
4. En qué consiste la operación subHilera(i, j)
5. En qué consiste la operación concat(s)
6. En qué consiste la operación inserte(s, i)
7. En qué consiste la operación borre(i, j)
8. En qué consiste la operación replace(i, j, s)
9. Qué hace la función posición

9.1 Definición. Una hilera es un objeto de datos compuesto, construido con base en el tipo primitivo char.

9.2 Operaciones con hileras. Consideremos ahora las operaciones básicas que se pueden ejecutar con hileras. Definamos una clase hilera con sus operaciones:

Clase hilera

Público:

```
int longitud()
hilera subHilera(int i, int j)
hilera concat(hilera s)
void inserte(hilera s, int i)
void borre(int i, int j)
void replace(int i, int j, hilera s)
int posicion(hilera s)
fin(hilera)
```

9.2.1. Asignación: En el lado izquierdo tendremos el nombre de la variable a la cual se le asigna una hilera dada. En el lado derecho se tendrá, o una hilera o una variable tipo hilera. Por ejemplo, si S, T, U y V son variables tipo hilera podemos tener las siguientes instrucciones:

```
S = "abc"
T = "def"
U = T
V = ""
```

En el primer ejemplo la variable S contendrá la hilera "abc".

En el segundo ejemplo la variable T contendrá la hilera "def"

En el tercer ejemplo la variable U contendrá la hilera "def"

En el cuarto ejemplo la variable V contendrá la hilera vacía.

9.2.2. Función longitud.

Forma general: longitud(S)

Esta función retorna el número de caracteres que tiene la hilera **S**. Por ejemplo, si tenemos S = "mazamorra" y ejecutamos la instrucción:

```
n = S.longitud()
```

La variable **n** quedará valiendo 9.

9.2.3. Función Subhilera.

Forma general: T = S.subHilera(i, j) con $1 \leq i \leq i+j-1 \leq n$

siendo **n** la longitud de la hilera S

Esta función, a partir de la posición **i** de la hilera **S** extrae **j** caracteres creando una nueva hilera y dejando intacta la hilera original **S**. Por ejemplo, si tenemos **S** = "mazamorra" y ejecutamos la instrucción:

```
T = S.subHilera(4, 5)
```

La variable T quedará conteniendo "amorr".

9.2.4. Función concatenar:

Forma general: U = S.concat(T)

Esta función, crea una copia de la hilera S y a continuación agrega una copia de la hilera T. Las hileras S y T permanecen intactas. Por ejemplo, si S = "nacio" y T = "nal" y ejecutamos la instrucción:

```
U = S.concat(T)
```

La variable U quedará conteniendo la hilera "nacional".

9.2.5. Método insertar:

Forma general: S.inserte(T, i)

Este método inserta la hilera T a partir del carácter de la posición **i** de la hilera S. Por ejemplo, si tenemos S = "nal" y T = "ciona" y ejecutamos la instrucción:

```
S.inserte(T, 3)
```

la hilera S quedara valiendo "nacional"

Es bueno resaltar que en este caso la hilera S queda modificada mientras que la hilera T permanece intacta.

9.2.6. Método borrar:

Forma general: `S.borre(i, j)`

Este método, a partir de la posición **i** de la hilera **S** borra **j** caracteres. Por ejemplo, si **S** = "amotinar" y ejecutamos la instrucción:

`S.borre(4, 4)`

la hilera **S** queda valiendo "amor"

Es bueno anotar también aquí, que la hilera **S** queda modificada.

9.2.7. Método reemplazar:

Forma general: `S.replace(i, j, T)`

Este método: a partir de la posición **i** de la hilera **S** reemplaza **j** caracteres por una copia de la hilera **T**. Por ejemplo, si la hilera **S** = "abcdef" y la hilera **T** = "xyz" y ejecutamos la instrucción:

`S.replace(3, 2, T)`

la hilera **S** queda valiendo "abxyze"

Este método modifica la hilera **S** mientras que la hilera **T** permanece intacta.

9.2.8 Función posición:

Forma general: `m = S.posicion(T)`

Esta función determina a partir de cuál posición de la hilera **S** se encuentra la hilera **T**, si es que encuentra la hilera **T** en la hilera **S**. En caso de no encontrar la hilera **T** en la hilera **S** retornará cero. Por ejemplo, si **S** = "mazamorra" y **T** = "amor" y ejecutamos la instrucción:

`m = posicion(T)`

la variable **m** quedará valiendo 4.

Las operaciones descritas aquí nos proporcionan las facilidades para manipular hileras en cualquier lenguaje de programación. Es importante agregar que lenguajes de programación como C++ y Java traen definida una clase hilera (String) en la cual se implementan una gran cantidad de variaciones correspondientes a estas operaciones definidas aquí.

9.3 Aplicación clase hilera.

Pasemos a considerar ejemplos de aplicación con las funciones anteriormente definidas.

Empecemos considerando un ejercicio en el cual nos solicitan determinar **cuál es la frecuencia de cada una de las letras del alfabeto español en un texto dado**, es decir, cuántas veces aparece la "a", cuántas veces aparece la "b" y así sucesivamente.

Llamemos **texto** la variable en la cual hay que hacer dicha evaluación.

Para desarrollar dicho algoritmo utilicemos una variable llamada **alfabeto**, la cual es una variable tipo hilera y que definiremos así:

alfabeto = "abcdefghijklmnñopqrstuvwxyz"

Utilizaremos un vector de 27 posiciones, que llamaremos **frecuencia**, en cada una de las cuales guardaremos el número de veces que se encuentre alguna letra del alfabeto definido. Es decir, a la letra "a" le corresponde la posición 1 del vector de frecuencias, a la letra "b" la posición 2, a la letra "c" la posición 3 y así sucesivamente.

Nuestro algoritmo consistirá en recorrer los caracteres de la variable **texto**, buscando cada carácter en la variable **alfabeto** utilizando la función **posicion**, cuando la encuentre sumaremos 1 a la posición correspondiente a esa letra en el vector **frecuencia**.

Nuestro algoritmo es:

```
1. void frecuenciaLetras(hilera alfabeto)
2.   int m, i, j, n
3.   hilera car
4.   m = alfabeto.longitud()
5.   for (i = 1; i <= m; i++) do
6.     frecuencia[i] = 0
7.   end(for)
8.   n = longitud()
9.   for (i = 1; i <= n; i++) do
10.    car = subHilera(i, 1)
11.    j = alfabeto.posicion(car)
12.    if (j != 0) then
13.      frecuencia[j] = frecuencia[j] + 1
14.    end(if)
15.  end(for)
16.  for (i = 1; i <= m; i++) do
17.    letra = alfabeto.subHilera(i, 1)
18.    write(letra, frecuencia[i])
19.  end(for)
20. fin(FrecuenciaLetras)
```

Consideremos ahora un algoritmo en el cual nos interesa determinar la frecuencia de cada palabra que aparezca en un texto.

Para desarrollar dicho algoritmo debemos, primero que todo, identificar cada una de las palabras del texto. Para lograr esta identificación hay que definir cuáles caracteres se utilizan como separadores de palabras. Estos caracteres pueden ser cualquier símbolo diferente de letra, es decir, la coma, el punto, el punto y coma, los dos puntos, el espacio en blanco, etc.

Nuestro algoritmo manejará las siguientes variables:

texto: variable en la cual se halla almacenado el texto a procesar.

palabras: variable tipo vector en la cual almacenaremos cada palabra que se identifique en el texto.

frecuencia: variable tipo vector en la cual almacenaremos el número de veces que se encuentre cada palabra del texto. Una palabra que se encuentre en la posición **i** del vector **palabra**, en la posición **i** del vector **frecuencia** se hallará el número de veces que se ha encontrado.

PalabraHallada: variable en la cual almacenaremos cada palabra que se identifique en el texto.

k: variable para contar cuántas palabras diferentes hay en el texto. La inicializamos en 1. El total de palabras diferentes encontradas será $k - 1$.

n: variable que guarda el número de caracteres en el texto.

alfabeto: es una hilera enviada como parámetro, la cual contiene los símbolos con los cuales se construyen las palabras.

Nuestro algoritmo es el siguiente:

```
1. void frecuenciaPalabras(hilera alfabeto)
2.   int i, k, j, p, n, frecuencia[1000]
3.   hilera car, palabras[1000]
4.   k = 1
5.   for (i = 1; i <= 1000; i++) do
6.     frecuencia[i] = 0
7.   end(for)
8.   n = longitud()
9.   i = 1
10.  while i <= n do
11.    car = subHilera(i, 1)
12.    p = alfabeto.posicion(car)
13.    while (i < n and p = 0) do
14.      i = i + 1
15.      car = subHilera(i, 1)
16.      p = alfabeto.posicion(car)
17.    end(while)
18.    j = i
19.    while (i < n and p <> 0) do
20.      i = i + 1
21.      car = subHilera(i, 1)
22.      p = alfabeto.posicion(car)
23.    end(while)
24.    palabraHallada = subHilera(j, i - j)
25.    palabras[k] = palabraHallada
26.    j = 1
27.    while (palabras[j] <> palabraHallada) do
28.      j = j + 1
29.    end(while)
30.    if (j == k) then
31.      frecuencia[k] = 1
32.      k = k + 1
33.    else
34.      frecuencia[j] = frecuencia[j] + 1
35.    end(if)
36.  end(while)
37.  k = k - 1
38.  for (i = 1; i <= k; i++) do
39.    write(palabras[i], frecuencia[i])
40.  end(for)
41. fin(frecuenciaPalabras)
```

En instrucciones 2 y 3 se definen las variables con las cuales se va a trabajar.

En instrucciones 5 a 7 se inicializan los contadores de palabras en cero.

En Instrucción 8 se determina la longitud del texto a analizar (el que invocó el método).

En instrucción 9 se inicializa la variable *i* en 1. La variable *i* la utilizamos para recorrer el texto, carácter por carácter, e ir identificando las diferentes palabras en él.

Instrucciones 10 a 36 conforman el ciclo principal del algoritmo.

En instrucciones 11 a 17 se omiten los caracteres que no conforman una palabra válida. Los caracteres que conforman una palabra válida son los que pertenecen al alfabeto. Cada carácter del texto se busca en la hilera alfabeto (instrucciones 12 y 16): si no lo encuentra significa que no conforma palabra, por tanto, se desecha. Del ciclo de las instrucciones 13 a 17 se sale cuando encuentre un carácter que pertenezca al alfabeto. Esto significa que a partir de esa posición comienza una palabra. Dicha posición la guardaremos en una variable que llamamos *j* (instrucción 18).

En el ciclo de las instrucciones 19 a 23 se determina hasta cuál posición llega una palabra: de este ciclo se sale cuando encuentre un carácter que no pertenezca al alfabeto, por consiguiente la palabra será la subhilera desde la posición *j* hasta la posición *i* – 1, la cual se extrae con la instrucción 24.

En la instrucción 25 se almacena la palabra hallada en la posición *k* del vector palabras. Con el fin de determinar si la palabra almacenada en la posición *k* del vector es primer vez que aparece o ya estaba, la buscamos en el vector de palabras (instrucciones 26 a 29) con la certeza de que la encontraremos: si se encuentra en la posición *k* significa que es la primer vez que aparece, por tanto, le asignamos 1 al contador de esa palabra (instrucción 31) e incrementamos la variable *k* en 1, si encontró la palabra en una posición diferente a la posición *k* significa que dicha palabra ya se había encontrado, por tanto incrementamos su respectivo contador en 1 (instrucción 34).

Por último, en las instrucciones 37 a 40 se escriben las diferentes palabras encontradas con su respectivo contador.

Planteemos ahora un algoritmo que considero interesante: reemplazar una palabra por otra en un texto dado.

Utilizaremos tres variables:

texto: Variable que contiene el texto donde hay que hacer el reemplazo.

vieja: Variable que contiene la hilera que hay que reemplazar.

nueva: variable que contiene la hilera que reemplazará la hilera vieja.

Para entender el desarrollo de este algoritmo debemos entender lo siguiente:

Si tenemos un texto *S* = “aabc**bc**abcde” y *T* = “abc” y ejecutamos la instrucción

$$m = S.\text{posicion}(T)$$

La variable **m** queda valiendo 2. Esto significa que a partir de la posición 2 de la hilera *S* se encontró la hilera *T*, es decir nuestro algoritmo retorna el sitio donde encuentra la primera ocurrencia de la hilera que se está buscando. Fíjese que en nuestra hilera *S* la hilera “abc” se halla en la posición 2 y en la 6.

Si ejecutamos la instrucción:

$$m = S.\text{subHilera}(5, 6).\text{posicion}(T) \quad \textbf{(fórmula 1)}$$

(Fíjese que se está buscando la hilera *T* a partir de la posición 5 de *S*)

La variable **m** también queda valiendo 2. Por qué?

porque la hilera **T** la buscará en la hilera *S.subHilera*(5, 6), la cual es: “babcde”

La pregunta es: está realmente la segunda ocurrencia de la 0hilera T en la posición 2 de S?. La respuesta es no.

Si planteamos una utilización de la función posicion como en la fórmula 1 y queremos determinar en cuál posición de S comienza la subhilera T, al resultado obtenido habrá que sumarle $i - 1$ siendo i el primer parámetro de la función subHilera. En nuestro caso $i = 5$.

Llamemos **posini** la variable a partir de la cual se inicia la búsqueda de una hilera en una subhilera obtenida con la función subHilera.

Nuestro algoritmo es:

```
1. void reemplazaViejaPorNueva(hilera vieja, hilera nueva)
2.   v = vieja.longitud()
3.   n = nueva.longitud()
4.   t = longitud()
5.   posini = 1
6.   sw = 1
7.   while (sw == 1) do
8.     p = subHilera(posini, t - posini + 1)
9.     posvieja = p.posicion(vieja)
10.    if (posvieja > 0) then
11.      posreal = posvieja + posini - 1
12.      replace(posreal, v, nueva)
13.      posini = posreal + n + 1
14.      t = longitud()
15.      if (posini > t + v + 1) then
16.        sw = 0
17.      end(if)
18.    else
19.      sw = 0
20.    end(if)
21.  end(while)
22.end(reemplazaViejaPorNueva)
```

EJERCICIOS PROPUESTOS

1. Elabore 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.
2. Elabore un algoritmo que justifique a la derecha un texto dado. El dato de entrada al subprograma es el número de caracteres que debe imprimir por línea.
3. Elabore 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). Su algoritmo debe retornar verdadero si cumple la condición, falso de lo contrario.
4. Elabore algoritmo que determine si una palabra tiene más vocales que consonantes o no. Su algoritmo debe retornar verdadero si cumple la condición, falso de lo contrario.
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".

6. Elabore un algoritmo que determine si una palabra tiene las cinco vocales o no. Su algoritmo debe retornar verdadero si cumple la condición, falso de lo contrario.
7. Elabore 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.
8. Elabore un algoritmo que procese un texto dado y que determine e imprima el número de veces que se halla cada palabra que contenga alguna de las siguientes sílabas: “bra”, “bre”, “bri”, “bro” o “bru”.

MÓDULO 10

HILERAS: REPRESENTACIONES CLASE HILERA, IMPLEMENTACIONES.

Introducción

En el módulo anterior definimos la clase hilera con sus operaciones y se desarrollaron algunos ejemplos de aplicación con objetos de la clase hilera. Note que en los ejercicios elaborados no nos preocupamos por la forma como se representarían las hileras dentro de la máquina. Veremos en este módulo las diferentes formas de representar hileras dentro de un computador.

Objetivos

1. Representar objetos de la clase hilera en arreglos de una dimensión.
2. Representar objetos de la clase hilera como listas ligadas.
3. Desarrollar los algoritmos para las diferentes operaciones de la clase hilera, representándolas en vectores.
4. Desarrollar los algoritmos para las diferentes operaciones de la clase hilera representándolas como listas ligadas.

Preguntas básicas

- 1.Cuál es el tipo base para representar hileras como vectores.
2. Cuáles son las diferentes formas para representar hileras.

10.1 Representación secuencial (en vector). En esta representación los caracteres de una hilera se colocan consecutivamente, siendo cada carácter un dato del tipo primitivo **char**. Si tenemos la hilera S = "peste", cada carácter se almacena en una posición del vector, comenzando en la posición 1. El final de la hilera se controla con un símbolo especial, **además en la posición 0 del vector tendremos el número de caracteres en el vector.**

	0	1	2	3	4	5	6	7	8	9
V	5	p	e	s	t	e	\n			

En otras palabras, definimos la clase hilera derivada de la clase vector. Por consiguiente podremos hacer uso de todos los métodos para manipular vectores. En nuestro caso particular, haremos uso del método longitud y el método datoEnPosicion(i), el cual retorna el dato que se halla en la posición i del vector. Con esta forma de representación algunas funciones serán fácilmente implementables. Funciones como longitud, subHilera, concat tendrán algoritmos fáciles y eficientes. En general, digamos que se definen vectores de tamaño 256 elementos. Para hileras muy cortas se desperdicia memoria, ya que no se utiliza la totalidad del vector definido. Además, operaciones como inserciones y borrados tendrán algoritmos no muy eficientes ya que implica mover datos en el vector.

Una de las operaciones a la cual se le ha dedicado mucho esfuerzo buscando que sea lo más eficiente posible es la función **posición**. Un algoritmo para dicha operación, bajo esta representación secuencial es:

```
1. int posicion(hilera p)
2.     int i, j, m, n
3.     m = p.longitud()
4.     n = longitud()
5.     i = 1
6.     j = 1
7.     while i <= m and j <= n do
8.         if (DatoEnPosicion(j) == p.DatoEnPosicion(i)) then
9.             i = i + 1
10.            j = j + 1
```

```

11.          if i > m then
12.              return j - i + 1
13.          end(if)
14.      else
15.          j = j - i + 2
16.          i = 1
17.      end(if)
18.  end(while)
19.  return 0
20.fin(posicion)

```

Este algoritmo tiene la inconveniente de que se devuelve en la hilera que invocó el método cuando encuentra una desigualdad entre **DatoEnPosicion(j)** y **P.DatoEnPosicion(i)**. Fíjese que cuando el resultado de la comparación de los caracteres de las posiciones i y j (instrucción 8) es falso ejecuta las instrucciones 15 y 16, las cuales lo que hacen es devolverse en la hilera que invocó el método (instrucción 15) y posicionarse nuevamente en el primer carácter de la hilera p (instrucción 16).

Para evitar este retroceso se puede construir un vector, que llamaremos **siguientes** el cual contendrá en la posición **i**, la posición del carácter de la hilera **P** con el cual se debe realizar la siguiente comparación con el carácter de la posición **j** de la hilera que invocó el método, y de esta forma no tener que devolvernos en la hilera que invocó el método.

Es decir, si **siguientes[6] = 4** significa que cuando se encuentre una desigualdad entre **P.DatoEnPosicion(6)** y algún **DatoEnPosicion(J)** el carácter **DatoEnPosicion(J)** habrá que compararlo con **P.DatoEnPosicion(4)**, es decir, el contenido de **siguientes[6]**. Si **siguientes[6]** es igual a cero se continuará comparando **DatoEnPosicion(J+1)** con **P.DatoEnPosicion(1)**.

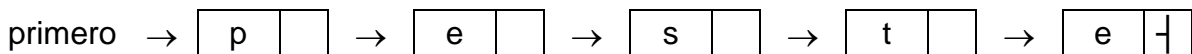
Teniendo construido el vector **siguientes** el algoritmo de **posicion** quedará así:

```

int  posicion(p, t)
    m = p.longitud()
    n = longitud()
    i = 1
    j = 1
    while i <= m and j <= n do
        if (p.DatoEnPosicion(i) == DatoEnPosicion(j)) then
            i = i + 1
            j = j + 1
            if i > m then
                return j - i + 1
            end(if)
        else
            if siguientes[i] > 0 then
                i = siguientes[i]
            else
                i = 1
                j = j + 1
            end(if)
        end(if)
    end(while)
    return 0
fin(posicion)

```

10.2 Como listas ligadas. Para representar hileras como listas ligadas basta definir la clase hilera derivada de la clase LSL. Teniendo esta definición podremos hacer uso de todos los métodos definidos para la clase LSL. Si queremos representar la hilera “peste” tendríamos:



Con esta representación no tendremos desperdicio de memoria y las operaciones de inserción y borrado tendrán algoritmos eficientes.

Veamos algunos algoritmos correspondientes a los métodos definidos. Comencemos con el algoritmo para el método subHilera. Recordemos que nuestro método retorna un objeto de la clase hilera y tiene dos parámetros: uno que indica a partir de cuál posición se debe hacer la copia (parámetro i) y el otro que indica cuántos caracteres se deben copiar (parámetro j). Nuestro algoritmo es:

```

1. hilera subHilera(int i, int j)
2.     int k, n
3.     hilera s
4.     nodoSimple p
5.     n = longitud()
6.     if (i < 1 or i > n) then
7.         write("parámetro i inválido")
8.         return null
9.     end(if)
10.    if (j < 1 or j > n - i + 1) then
11.        write("parámetro j inválido")
12.        return null
13.    end(if)
14.    p = primerNodo()
15.    k = 1
16.    s = new hilera()
17.    while (k < i) do
18.        p = p.retornaLiga()
19.        k = k + 1
20.    end(while)
21.    k = 1
22.    while (k <= j) do
23.        s.insertar(p.retornaDato(), s.ultimoNodo())
24.        p = p.retornaLiga()
25.        k = k + 1
26.    end(while)
27.    return s
28. fin(subHilera)

```

El parámetro i es el que indica a partir de cuál posición se deben copiar j caracteres, el cual es el segundo parámetro.

Instrucciones 2 a 4 definen las variables de trabajo.

En la instrucción 5 se determina la longitud de la hilera que invocó el método, con el fin de controlar que los parámetros i y j sean válidos.

Instrucciones 6 a 9 controlan que el parámetro i sea válido.

Instrucciones 10 a 13 controlan que el parámetro j sea válido.

Instrucciones 14 a 16 asignan los valores iniciales a las variables de trabajo.

Instrucciones 17 a 20 ubican a **p** en el nodo a partir del cual hay que realizar la copia de **j** caracteres.

Instrucciones 22 a 26 copian **j** caracteres insertando nodos al final de la lista que está construyendo. Fíjese que utilizamos los métodos insertar y ultimoNodo que corresponden a la clase LSL.

Consideremos ahora el algoritmo para el método de concatenar. Recuerde que nuestro método lo llamamos concat, es invocado por un objeto de la clase hilera, tiene como parámetro otro objeto de la clase hilera y retorna un nuevo objeto de la clase hilera.

```
1. hilera concat(hilera t)
2.     hilera s
3.     nodoSimple p
4.     s = copiaHilera()
5.     p = t.primerNodo()
6.     while (no t.finDeRecorrido(p)) do
7.         s.insertar(p.retornaDato(), s.ultimoNodo())
8.         p = p.retornaLiga()
9.     end(while)
10.    return s
11.fin(concat)
```

El anterior algoritmo concatena la lista **s** con la lista **t** creando una nueva lista y dejando intactas las listas **s** y **t**.

Instrucciones 2 y 3 definen las variables de trabajo.

El método copiaHilera() es un método que como su nombre lo dice simplemente construye una copia de la hilera que lo invoca. Más adelante presentamos el algoritmo correspondiente a este método.

En instrucción 5 nos ubicamos con **p** en el primer nodo de la lista **t** enviada como parámetro y en las instrucciones 6 a 9 insertamos todos los nodos de la lista **t** al final de la lista **s**.

```
1. hilera copiaHilera()
2.     hilera s
3.     nodoSimple p
4.     p = primerNodo()
5.     s = new hilera()
6.     while (no finDeRecorrido(p)) do
7.         s.insertar(p.retornaDato(), s.ultimoNodo())
8.         p = p.retornaLiga()
9.     end(while)
10.    return s
11.fin(copiaHilera)
```

EJERCICIOS PROPUESTOS

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

2. Elabore algoritmo para la operación concatenación considerando las hileras representadas como vectores.

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

4. Elabore algoritmo para la operación borre considerando las hileras representadas como vectores.
5. Elabore algoritmo para la operación replace considerando las hileras representadas como vectores.
6. Elabore algoritmo para la operación inserte considerando las hileras representadas como listas ligadas.
7. Elabore algoritmo para la operación borre considerando las hileras representadas como listas ligadas.
8. Elabore algoritmo para la operación replace considerando las hileras representadas como listas ligadas.
9. Elabore algoritmo para la operación posición considerando las hileras representadas como listas ligadas.

MÓDULO 11

COLAS: DEFINICIÓN, CLASE COLA, REPRESENTACIONES, IMPLEMENTACIONES.

Introducción

En el módulo 11 tratamos una estructura en la cual la forma de procesamiento es que el último que entra es el primero que sale. Hay otras situaciones en las cuales la forma de procesamiento es que el primero que llega es el primero que sale. Para este tipo de procesamiento se define una nueva estructura la cual se define como una cola. Trataremos en este módulo dicha estructura.

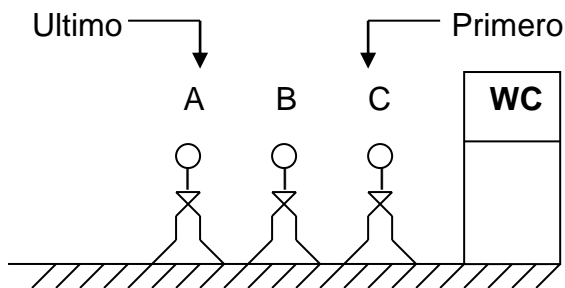
Objetivos

1. Conocer la estructura cola y sus características.
2. Definir una clase cola con sus operaciones.
3. Representar colas dentro de un computador.

Preguntas básicas

1. Qué es una cola?
2. En qué consiste la operación encolar?
3. En que consiste la operación desencolar?
4. En cuáles situaciones se utiliza una cola?
5. Cuáles son las diferentes formas para representar colas en un computador?

11.1 Definición: Una cola es una lista ordenada en la cual las operaciones de inserción se efectúan en un extremo llamado **ultimo** y las operaciones de borrado se efectúan en el otro extremo llamado **primero**. Es una estructura **FIFO** (First Input First Output).



En términos prácticos es lo que supuestamente se debe hacer para tomar un bus, para comprar las boletas para entrar a un cine o para hacer uso de un servicio público. Las operaciones sobre una cola son:

crear: crea una cola vacía.

esVacía(): retorna verdadero si la cola está vacía, falso de lo contrario.

esLlena(): retorna verdadero si la cola está llena, falso de lo contrario.

encolar(d): inserta un dato **d** al final de la cola.

desencolar(): remueve el primer elemento de la cola.

siguiente(): retorna el dato que se halla de primero en la cola.

11.2 Representación de colas en un vector, en forma no circular.

Para representar colas en esta forma se requiere un vector, que llamaremos **V** y dos variables: una que llamaremos **primero**, la cual indique la posición del vector en la cual se halla el primer dato de la cola, y otra, que llamaremos **ultimo**, la cual indica la posición en la cual se halla el último dato de la cola.

En el vector **V** se guardan los datos, **primero** apuntará hacia la posición anterior en la cual se halla realmente el primer dato de la cola, **ultimo** apuntará hacia la posición en la que realmente se halla el último dato de la cola.

Si tenemos un vector con la siguiente configuración:

	1	2	3	4	5	6	7
V			a	b	c		

La variable **primero** valdrá 2 indicando que el primer elemento de la cola se halla en la posición 3 del vector; la variable **ultimo** valdrá 5 indicando que el último elemento de la cola se halla en la posición 5 del vector.

Las operaciones sobre la cola que son encolar y desencolar funcionan de la siguiente forma: si se desea encolar basta con incrementar la variable **ultimo** en uno y llevar a la posición **ultimo** del vector el dato a encolar; si se desea desencolar basta con incrementar en uno la variable **primero** y retornar el dato que se halla en esa posición.

En general, si el vector **V** se ha definido con **n** elementos, cuando la variable **ultimo** sea **n**, se podría pensar que la cola está llena.

	1	2	3	4	5	6	7
				b	c	d	e

Pero, como se observa en la figura, el vector tiene espacio al principio, por consiguiente podemos pensar en mover los datos hacia el extremo izquierdo y actualizar **primero** y **ultimo** para luego encolar. Es decir, el hecho de que **ultimo** se igual a **n** no es suficiente para determinar que la cola está llena. Para que la cola esté llena se deben cumplir dos condiciones: que **primero** sea igual a cero y **ultimo** sea igual a **n**.

Consideremos ahora operaciones sucesivas de desencole para el ejemplo dado: después de ejecutar la primera operación de desencole los valores de las variables **ultimo** y **primero** y del vector **V** será la siguiente:

	1	2	3	4	5	6	7
				b	c		

donde la variable **primero** valdrá 3 y la variable **ultimo** 5.

Al desencolar nuevamente, la configuración del vector será:

	1	2	3	4	5	6	7
					c		

donde la variable **primero** vale 4 y la variable **ultimo** 5.

Si desencolamos de nuevo, la configuración del vector será:

	1	2	3	4	5	6	7

y la variable **primero** vale 5 y la variable **ultimo** 5.

En otras palabras **primero** es igual a **ultimo** y la cola está vacía, o sea, la condición de cola vacía será **primero == ultimo**. Teniendo definidas las condiciones de cola llena y cola vacía procedamos a definir la clase cola.

Clase cola

Privado

entero primero, ultimo, n
objeto V[]

Público

cola(n) // constructor
boolean esVacia()
boolean esLlena()
void encolar(objeto d)
objeto desencolar()
objeto siguiente()

fin(clase cola)

cola(entero m)

n = m
primero = ultimo = 0
V = new objeto[n]

fin(cola)

boolean esVacia()

return primero == ultimo

fin(esVacia)

boolean esLlena()

return primero == 0 and ultimo == n

fin(esLlena)

void encolar(objeto d)

if (esLlena()) then
write("cola llena ")
return

end(if)

if (ultimo == n) then
for (i = primero + 1; i <= n; i++)
V[i - primero] = V[i]
end(for)
ultimo = ultimo - primero
primero = 0

end(if)

ultimo = ultimo + 1
v[ultimo] = d

fin(encolar)

objeto desencolar()

if (esVacia()) then
write("cola Vacía")
return null

end(if)

primero = primero + 1
return V[primero]

fin(desencolar)

si consideramos una situación como la siguiente:

0	1	2	3	4	5	6
	e	f	b	c	d	g

n vale 7, **ultimo** vale 6 y **primero** vale 1.

Si se desea encolar el dato **h** y aplicamos el método **encolar** definido, la acción que efectúa dicho subprograma es mover los elementos desde la posición 1 hasta la 6, una posición hacia la izquierda de tal forma que quede espacio en el vector para incluir la **h** en la cola. El vector quedará así:

0	1	2	3	4	5	6
e	f	b	c	d	g	

con **primero** valiendo 0 y **ultimo** valiendo 5. De esta forma continúa la ejecución del subprograma **encolar** y se incluye el dato **h** en la posición 6 del vector. El vector queda así:

0	1	2	3	4	5	6
e	f	b	c	d	g	h

Con **primero** valiendo 0 y **ultimo** valiendo 6.

Si en este momento se desencola el vector queda así:

0	1	2	3	4	5	6
	f	b	c	d	g	h

con **primero** valiendo 1 y **ultimo** valiendo 6.

Si la situación anterior se repite sucesivas veces, el cual sería el peor de los casos, cada vez que se vaya a encolar se tendrían que mover **n-1** elementos en el vector, lo cual haría ineficiente el manejo de la cola, ya que el proceso de encolar tendría orden de magnitud **O(n)**. Para obviar este problema y poder efectuar los subprogramas de **encolar** y **desencolar** en un tiempo **O(1)** manejaremos el vector circularmente.

11.3 Representación de colas circularmente en un vector.

Para manejar una cola circularmente en un vector se requiere definir un vector de **n** elementos con los subíndices en el rango desde 0 hasta **n-1**, es decir, si el vector tiene 10 elementos los subíndices variarán desde 0 hasta 9, y el incremento de las variables **primero** y **ultimo** se hace utilizando la operación **módulo** (%) de la siguiente manera:

$\text{primero} = (\text{primero} + 1) \% n$
 $\text{ultimo} = (\text{ultimo} + 1) \% n$

Recuerde que la operación módulo (%) retorna el residuo de una división entera. Si se tiene una cola en un vector, con la siguiente configuración:

0	1	2	3	4	5	6
			b	c		

n vale 7, los subíndices varían desde 0 hasta 6, **primero** vale 2 y **ultimo** vale 4.

Si se desea encolar el dato **d** incrementamos la variable **ultimo** utilizando la operación **módulo** así:

$$\text{ultimo} = (\text{ultimo} + 1) \% n$$

o sea, **ultimo** = **(4+1) % 7**, la cual asignará a **ultimo** el residuo de dividir 5 por 7, que es 5. El vector queda así:

0	1	2	3	4	5	6
			b	c	d	

Al encolar el dato **e** el vector queda así:

0	1	2	3	4	5	6
			b	c	d	e

y las variables **primero** y **ultimo** valdrán 2 y 6 respectivamente. Al encolar de nuevo, digamos el dato **f**, aplicamos el operador **%** obteniendo el siguiente resultado:

ultimo = **(6+1) % 7**, el cual es 0, ya que el residuo de dividir 7 por 7 es cero.

Por consiguiente la posición del vector a la cual se llevará el dato **f** es la posición 0. El vector quedará con la siguiente conformación:

0	1	2	3	4	5	6
f			b	c	d	e

con **ultimo** valiendo 0 y **primero** 2. De esta forma hemos podido encolar en el vector sin necesidad de mover elementos en él.

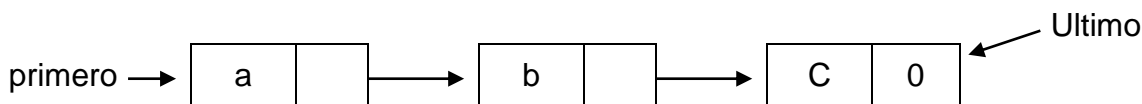
Los subprogramas para encolar y desencolar quedan así:

```
void encolar(objeto d)
    ultimo = (ultimo+1) % n
    if (ultimo == primero) then
        colaLlena()
    end(if)
    V[ultimo] = d
fin(encolar)
```

```
objeto desencolar()
    if (primero == ultimo) then
        colaVacía
    end(if)
    primero = (primero+1) % n
    d = V[primero]
fin(desencolar)
```

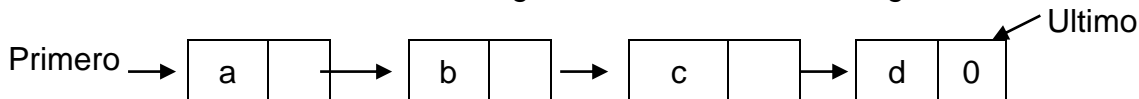
Es importante notar que la condición de cola llena y cola vacía es la misma, con la diferencia de que en el subprograma **encolar** se chequea la condición después de incrementar **ultimo**. Si la condición resulta verdadera invoca el subprograma **colaLlena** cuya función será dejar **ultimo** con el valor que tenía antes de incrementarla y detener el proceso, ya que no se puede encolar. Por consiguiente habrá una posición del vector que no se utilizará, pero que facilita el manejo de las condiciones de cola vacía y cola llena.

11.4 Representación de colas como listas ligadas: La cola la definimos como derivada de la clase LSL.



Las operaciones sobre la cola son *Encolar* y *Desencolar*.

Encolar: Consiste en insertar un registro al final de una lista ligada.



```
void encolar(objeto d)
    insertar(d, ultimoNodo())
fin(encolar)
```

Desencolar: Consiste en borrar el primer registro de una lista ligada (exacto a Desapilar). Habrá que controlar si la cola está o no vacía.

```
objeto desencolar(primero, ultimo, d)
    if (esVacia()) then
        write("cola vacía, no se puede desencolar")
        return null
    end(if)
    nodoSimple p
    p = primerNodo()
    d = p.retornaDato()
    borrar(p, null)
    return d
fin(desencolar)
```

11.5 Manejo circular de dos colas en un vector de n elementos.

	COLA 1										m	COLA 2									
	0	1	2	3	4	5	6	7	8	9		10	11	12	13	14	15	16	17	18	19
V	a	b	c	d								h	i	J					e	f	g

Veamos ahora cómo trabajar dos colas, cada una circularmente, en un vector de **n** elementos. En nuestro ejemplo **n = 20**.

Inicialmente, a cada cola le corresponderá una mitad del vector. La cola 1 se representará desde la posición cero hasta la posición **m - 1**, y la cola 2 se representará desde la posición **m** hasta la posición **n - 1**.

Llamemos **p1** y **u1** las variables para identificar las posiciones en las cuales se hallan el primero y el último de la cola 1, y **p2** y **u2** las variables para identificar las posiciones en las cuales se hallan el primero y el último de la cola 2.

Las operaciones para encolar y desencolar en la cola 1 son idénticas al manejo circular de una cola en un vector:

$$p1 = (p1 + 1) \% m$$

$$u1 = (u1 + 1) \% m$$

Para encolar y desencolar en la cola 2, y que ésta se comporte circularmente, debemos tener en cuenta que la porción de vector que le corresponde va desde m hasta $n - 1$.

El número de elementos en ella es $n - m$, y para que su comportamiento sea circular debemos, temporalmente, convertir los valores de $p2$ y $u2$ a la forma desde 0 hasta $n - m$.

Para lograr esto, basta con restarle m a los valores de $p2$ y $u2$ antes de aplicar la operación módulo y después incrementar el resultado en m para que el valor obtenido quede en el rango correcto: desde m hasta $n - 1$.

Las instrucciones para incrementar $p2$ y $u2$ son:

$$p2 = (p2 - m + 1) \% (n - m) + m$$
$$u2 = (u2 - m + 1) \% (n - m) + m$$

Teniendo definido la forma de incremento de las variables de manejo de cada cola, nuestros algoritmos para encolar y desencolar en cada una de ellas serán:

```
void encolar(entero cola, objeto d)
  if (cola == 1) then
    u1 = (u1 + 1) % m
    if (u1 == p1) then
      colaLlena(cola)
    end(if)
    V[u1] = d
  else
    u2 = (u2 - m + 1) % (n - m) + m
    if (u2 == p2) then
      colaLlena(cola)
    end(if)
    V[u2] = d
  end(if)
fin(encolar)
```

```
objeto desencolar(entero cola)
  if (cola == 1) then
    if (u1 == p1) then
      write("cola 1 vacía")
      return null
    else
      p1 = (p1 + 1) % m
      d = V[p1]
    end(if)
  else
    if (u2 == p2) then
      write("cola 2 vacía")
      return null
    else
      p2 = (p2 - m + 1) % (n - m) + m
      d = V[p2]
    end(if)
  end(if)
  return d
fin(desencolar)
```

Consideremos el algoritmo colaLlena

	COLA 1									m	COLA 2								
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	a	b	c	d	e	f	g	h	i		p	q	r	s					
2		a	b	c	d	e	f	g	h	i		p	q	r	s				
3	e	f	g	h	i		a	b	c	d						p	q	r	s
4											r	s						p	q

Figura 11.1

El subprograma colaLlena se invoca cuando se desee encolar en una de las dos colas y ésta está llena. En general, el subprograma colaLlena deberá averiguar si hay espacio disponible en la otra cola y efectuar los movimientos apropiados para abrir espacio en la cola que se llenó y poder encolar.

Comencemos considerando la situación en la cual es la cola 1 la que está llena, la cola 2 no está llena y necesitamos abrir espacio para poder encolar en la cola 1.

Para lograr este objetivo debemos hacer algún tratamiento tanto a la cola 1 como a la cola 2, dependiendo de las situaciones de cola llena y de espacio disponible respectivamente

Si la **cola 1** está llena se pueden presentar tres formas diferentes, las cuales llamaremos 1, 2 y 3 (ver figura 11.1). En cualquier situación tenemos $p1 = u1$.

En la situación 1, la cual identificamos porque $p1$ y $u1$ son iguales a $m - 1$, la acción a tomar es simplemente incrementar $p1$ en 1.

En la situación 2, la cual identificamos porque $p1$ y $u1$ son iguales a cero, la acción a tomar es simplemente hacer $u1$ igual a m .

En la situación 3, la cual identificamos porque $p1$ y $u1$ están entre 1 y $m - 2$, hay que mover los datos desde la posición $p1 + 1$ hasta la posición $m - 1$ una posición hacia la derecha y actualizar $p1$.

Las instrucciones correspondientes a estas situaciones son:

casos

$p1 == m - 1$: // situación 1

$p1 = p1 + 1$

$u1 == 0$: // situación 2

$u1 = m$

else: // situación 3

for ($i = m$; $i > u1$; $i --$) do

$V[i] = V[i - 1]$

end(for)

$p1 = p1 + 1$

fin(casos)

Las situaciones de espacio disponible en la cola 2 las llamaremos 1, 2, 3, y 4 (Ver figura 11.1).

En situaciones 1 y 4, las cuales identificamos porque **p2** es mayor que **u2**, hay que mover los datos desde la posición **m** hasta la posición **u2** una posición hacia la derecha y actualizar **u2**.

La situación 2, la cual identificamos porque **p2** es igual a **m** basta con asignar a **p2** el valor de **n - 1**. Pero antes de hacer esto debemos considerar el hecho de que de pronto la cola 2 está vacía y entonces habrá que asignar **n - 1** tanto a **p2** como a **u2**.

En la situación 3 no hay que efectuar operación alguna.

Las instrucciones correspondientes son:

casos

p2 > u2:

for (**i = u2; i >= m; i --**) do

V[i + 1] = V[i]

end(for)

u2 = u2 + 1

p2 == m:

p2 = n - 1

if (**m == u2**) then

u2 = p2

end(if)

fin(casos)

Cualquiera que hubieran sido los tratamientos que se hayan hecho sobre ambas colas el valor de **m** hay que incrementarlo en 1.

Analicemos ahora el caso en que hubiera sido la cola 2 la que está llena y la cola 1 tiene espacio.

Consideremos las diferentes situaciones para este caso.

	COLA 1										m	COLA 2							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	a	b	c	d	e	f					p	q	r	s	t	u	v	w	
2					a	b	c	d	e	f		p	q	r	s	t	u	v	w
3			a	b	c	d	e	f			t	u	v	w		p	q	r	s
4	d	e	f					a	b	c									

Figura 11.2

Las situaciones de la cola 2 llena son 3, las cuales identificaremos como 1, 2 y 3. En cualquier situación **p2 = u2**.

La situación 1, la cual identificamos porque **p2 = n - 1**, basta con hacer **p2 = m - 1**

La situación 2, la cual identificamos porque **u2 = m**, basta con hacer **u2 = m - 1**.

La situación 3 es cuando no se cumple ninguna de las dos condiciones anteriores y será necesario mover los datos en el vector desde la posición **m** hasta la posición **u2** una posición hacia la izquierda.

Las instrucciones correspondientes son:

```

casos
  p2 == n - 1:
    p2 = m - 1
  u2 == m:
    u2 = m - 1
  else:
    for (i = m; i <= u2; i++) do
      V[i - 1] = V[i]
    end(for)
fin(casos)

```

Consideremos ahora el tratamiento a la cola 1. Hay 4 situaciones de cola 1 con espacio (ver figura 11.2):

Situaciones 1 y 4, las cuales identificamos porque $p1 > u1$ se solucionan así: si $p1$ es igual a $m - 1$ basta con restar 1 a $p1$, de lo contrario hay que mover los datos en el vector, desde la posición $p1$ hasta la posición $m - 1$, una posición hacia la izquierda y restarle 1 a $p1$.

Situaciones 2 y 3, las cuales identificamos porque $p1 < u1$ se solucionan así: si $u1$ es menor que $m - 1$ no hay que tomar ninguna acción, de lo contrario hay que mover todos los datos de la cola uno una posición hacia la izquierda y restar uno a $p1$ y a $u1$.

Cualquiera que hubieran sido los tratamientos a las dos colas hay que restar 1 a m .

Las instrucciones correspondientes son;

```

casos
  p1 > u1:
    if (p1 < m - 1) then
      for (i = p1; i < m - 1; i++) do
        V[i] = V[i+1]
      end(for)
    end(if)
    p1 = p1 - 1
  p1 < u1:
    if (u1 == m - 1) then
      for (i = p1; i < m - 1; i++) do
        V[i] = V[i+1]
      end(for)
      p1 = p1 - 1
      u1 = u1 - 1
    end(if)
fin(casos)

```

Agrupando todas las situaciones planteadas nuestro algoritmo de colaLlena queda así:

```

void colaLlena(cola)
  if (cola == 1) then
    if (((u2 - m + 1) % (n - m) + m) <> p2) then // hay espacio en cola 2
      // tratamiento cola 2 cuando cola 1 está llena.
      if (u2 < p2) then
        for (i = u2; i >= m; i --) do
          V[i+1] = V[i]
        end(for)
        u2 = u2 + 1

```

```

else
    if (p2 == m) then
        p2 = n - 1
        if (u2 == m) then
            u2 = p2
        end(if)
    end(if)
end(if)
// Tratamiento cola 1, cuando cola 1 está llena.
if (u1 == m - 1) then
    p1 = p1 + 1
else
    if (u1 == 0) then
        u1 = m
    else
        for (i = m; i > u1+1; i --) do
            vec[i] = vec[i-1]
        end(for)
        p1 = p1 + 1
    end(if)
end(if)
m = m + 1
return
end(if)
else
    if ((u1 + 1) % m <> p1) then // hay espacio en cola 1
        // tratamiento cola 1 cuando cola 2 está llena
        if ((p1 < u1) and (u1 == m - 1)) then
            for (i = p1; i < u1; i++) do
                vec[i] = vec[i+1]
            end(for)
            u1 = u1 - 1
            p1 = p1 - 1
        else
            if ((p1 <> m - 1) and (p1 > u1)) then
                for (i = p1; i < m - 1; i++) do
                    vec[i] = vec[i+1]
                end(for)
                p1 = p1 - 1
            else
                if (p1 == m - 1) then
                    p1 = p1 - 1
                    if (u1 == m - 1) then
                        u1 = p1
                    end(if)
                end(if)
            end(if)
        end(if)
    end(if)
    // tratamiento cola 2 cuando cola 2 está llena
    if (p2 == n - 1) then
        p2 = m - 1
    else
        if (p2 == m)
            u2 = m - 1
        else

```

```

        for (i = m; i < p2; i++) do
            vec[i - 1] = vec[i]
        end(for)
        u2 = u2 - 1
    end(if)
end(if)
m = m - 1
return
end(if)
end(if)
write("ambas colas están llenas")
stop
fin(colaLlena)

```

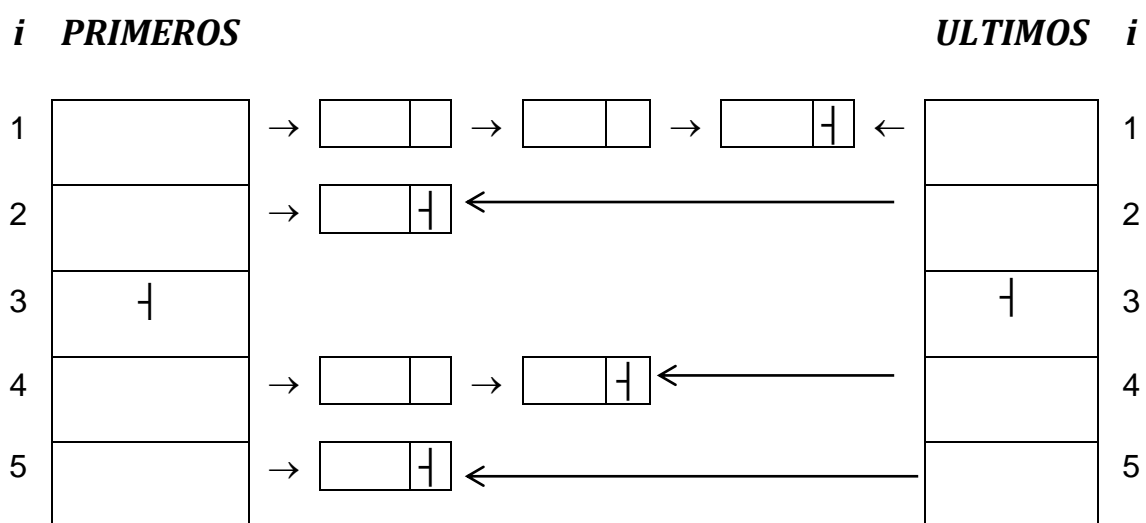
11.6 Manejo de n colas en un vector De la misma forma que puede suceder que haya que manejar n pilas, también puede suceder que haya que manejar n colas. De una forma similar podemos manejar n colas en un vector de m elementos. En el manejo de n colas necesitaremos tres vectores auxiliares: uno que indique en cuál posición del vector empieza cada cola, otro que indique en cuál posición del vector se halla el primer elemento de cada cola y otro que indique en cuál posición del vector se halla el último elemento de cada cola. Cada cola se maneja circularmente

11.7 Manejo de n colas como listas ligadas Equivale a manejar n listas, una lista por cada cola. Para manejar n colas como listas ligadas simplemente manejamos dos vectores, los cuales llamaremos primeros y ultimos.

primeros[i]: Apunta hacia el primer registro de la lista ligada que representa la cola i .

ultimos[i]: Apunta hacia el último registro de la lista ligada que representa la cola i .

Los subprogramas para **encolar** y **desencolar** son los mismos que manejando una sola cola, con la inclusión del parámetro i , el cual indica en cuál cola se va a encolar o a desencolar, y cambiando **primero** por **primeros[i]** y **ultimos** por **últimos[i]**.



Representación gráfica del manejo de N colas, representada cada una como lista ligada.

EJERCICIOS PROPUESTOS

1. Elabore algoritmo colaLlena para el algoritmo encolar desarrollado en el numeral 11.3. 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.
2. Elabore un algoritmo que trabaje una cola circular utilizando todos los elementos del vector.
3. Elabore un algoritmo que maneje una pila y una cola en un vector. La cola se debe manejar circularmente.
4. Diseñe la forma de representar una cola como lista ligada, en la cual sólo utilice un apuntador de entrada a la lista. Su diseño debe ser tal que los algoritmos de encolar y desencolar sean **$O(1)$** .
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.
6. Elabore algoritmo cola llena para el diseño desarrollado en el punto anterior.
7. 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.

MÓDULO 12

PILAS: DEFINICIÓN, CLASE PILA, REPRESENTACIONES, IMPLEMENTACIONES.

Introducción

Hemos tratado hasta ahora, estructuras que permiten almacenar colecciones de datos: arreglos y listas ligadas. En esto de almacenar colecciones de datos y efectuar operaciones de búsqueda, inserción y borrado, hay situaciones en las cuales se requiere guardar datos y recuperarlos en orden inverso al que fueron guardados. Veremos en este módulo la estructura pila, la cual permite procesar los datos de esta manera.

Objetivos

4. Conocer la estructura pila y sus características.
5. Definir una clase pila con sus operaciones.
6. Representar pilas dentro de un computador.
7. Conocer aplicaciones de la clase pila.

Preguntas básicas

6. Qué es una pila?
7. En qué consiste la operación apilar?
8. En que consiste la operación desapilar?
9. Qué diferencia hay entre la operación cima y la operación desapilar?
10. En cuáles situaciones se utiliza una pila?
11. Cuáles son las diferentes formas para representar pilas en un computador?

12.1 Definición

Una pila es una lista ordenada en la cual todas las operaciones (inserción y borrado) se efectúan en un solo extremo llamado **tope**. Es una estructura **LIFO** (Last Input First Output) que son las iniciales de las palabras en inglés último en entrar primero en salir, debido a que los datos almacenados en ella se retiran en orden inverso al que fueron entrados.

Un ejemplo clásico de aplicación de pilas en computadores se presenta en el proceso de llamadas a subprogramas y sus retornos.

Supongamos que tenemos un programa principal y 3 subprogramas así:

Programa principal	void P1	void P2	void P3
---	---	---	---
---	---	---	---
---	---	---	---
---	P2	P3	---
P1	L2→ ---	L3 → ---	---
L1→ ---	---	---	---
---	---	---	---
---	---	---	---
fin(prog. Ppal)	fin(P1)	fin(P2)	fin(P3)

Cuando se ejecuta el programa principal, se hace una llamada al subprograma P1, es decir, ocurre una interrupción a la ejecución del programa principal. Antes de iniciar la ejecución de este subprograma, se guarda la dirección de la instrucción donde debe retornar a continuar la ejecución del programa principal cuando termine de ejecutar el subprograma. Llamemos L1 esta dirección. Cuando ejecuta el subprograma P1 existe una llamada al subprograma P2, hay una nueva interrupción, pero antes de ejecutar el subprograma P2 se guarda la dirección de la instrucción donde debe retornar a continuar la ejecución del subprograma P1, cuando termine de ejecutar el subprograma P2. Llamemos L2 esta dirección.

Hasta el momento hay guardados dos direcciones de retorno:

L1, L2

Cuando ejecuta el subprograma P2 hay llamada a un subprograma P3, lo cual implica una nueva interrupción y por ende guardar una dirección de retorno al subprograma P2, la cual llamamos L3.

Tenemos entonces tres direcciones guardadas así:

L1, L2, L3

Al terminar la ejecución del subprograma P3, retorna a continuar ejecutando en la última dirección que guardó, es decir, extrae la dirección L3 y regresa a continuar ejecutando el subprograma P2 en dicha instrucción. Los datos guardados ya son:

L1, L2

Al terminar el subprograma P2, extrae la última dirección que tiene guardada y en este caso L2, y retorna a esta dirección a continuar la ejecución del subprograma P1.

En este momento los datos guardados son:

L1

Al terminar la ejecución del subprograma P1 retornará a la dirección que tiene guardada, o sea a L1.

Obsérvese que los datos fueron procesados en orden inverso al que fueron almacenados, es decir, último en entrar primero en salir. Es esta forma de procesamiento la que define una estructura PILA.

Veamos cuáles son las operaciones que definiremos para la clase pila.

crear: crea una pila vacía.

apilar: incluye el dato d en la pila.

desapilar: elimina el último elemento de la pila y deja una nueva pila la cual queda con un elemento menos.

cima: retorna el dato que está de último en la pila, sin eliminarlo.

esVacía: retorna verdadero si la pila está vacía, falso de lo contrario.

esLlena: retorna verdadero si la pila está llena, falso de lo contrario.

12.2 Representación de pilas en un vector. La forma más simple es utilizar un arreglo de una dimensión y una variable, que llamaremos **tope**, que indique la posición del arreglo en la cual se halla el último elemento de la pila. Por consiguiente, vamos a definir la clase pila derivada de la clase Vector. La variable **m** de nuestra clase vector funciona como la variable **tope**. Definamos entonces nuestra clase pila.

Clase pila

Privado:

object V[]

int tope, n

Público:

pila(int m) // Constructor

boolean esVacía()

boolean esLlena()

void apilar(objeto d)

```

        object desapilar()
        void desapilar(entero i)
        objeto tope()
fin(clase pila)

```

Como hemos definido la clase pila derivada de la clase Vector veamos cómo son los algoritmos correspondientes a estos métodos.

```

pila(int m)           // Constructor
    V = new array[m]
    n = m
    tope = 0
fin(pila)

```

```

boolean esVacía()
    return tope == 0
fin(esVacía)

```

```

boolean esLlena()
    return tope == n
fin(esLlena)

```

```

void apilar(objeto d)
    if (esLlena()) then
        write("pila llena")
        return
    end(if)
    tope = tope + 1
    V[tope] = d
fin(apilar)

```

Apilar simplemente consiste en sumarle 1 a tope y llevar el dato a esa posición del vector.

```

objeto desapilar()
    if (esVacía()) then
        write("pila vacía")
        return null
    end(if)
    d = V[tope]
    tope = tope - 1
    return d
fin(desapilar)

```

Estrictamente hablando, desapilar simplemente consiste en eliminar el dato que se halla en el tope de la pila. Sin embargo es usual eliminarlo y retornarlo. Eso es lo que hace nuestro anterior algoritmo para desapilar. El proceso de eliminación simplemente consiste en restarle 1 a tope. Definamos, en forma polimórfica, otro algoritmo para desapilar. Este nuevo algoritmo tendrá un parámetro **i**, el cual indicará cuántos elementos de deben eliminar de la pila.

```

void desapilar(entero i)
    if ((tope - i) >= 0) then
        tope = tope - i
    else
        write("no se pueden eliminar tantos elementos")
    end(if)

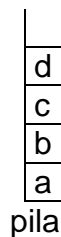
```

```

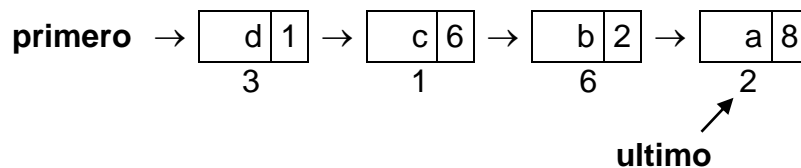
fin(desapilar)
objeto cima()
    if (esVacia()) then
        write("pila vacía")
        return null
    end(if)
    return V[tope]
fin(cima)

```

12.3 Representación de pilas como lista ligada. Para representar una pila como lista ligada basta definir la clase pila derivada de la clase **LSL**. Nuevamente, como estamos definiendo la clase pila derivada de la clase lista simplemente ligada, podremos hacer uso de todos los métodos que hemos definido para la clase **LSL**. A modo de ejemplo, representemos como lista ligada la siguiente pila:



Dibujémosla de la forma como solemos hacerlo con listas ligadas



Como hemos dicho, las operaciones sobre una pila son apilar y desapilar.

apilar: consiste en insertar un registro al principio de una lista ligada.

desapilar: consiste en eliminar el primer nodo de la lista ligada y retornar el dato que se hallaba en ese nodo.

Al definir la clase pila derivada de la clase **LSL** el método para controlar pila llena ya no aplica. Los métodos para la clase pila son:

```

void apilar(objeto d)
    insertar(d, null)
fin(apilar)

```

```

objeto desapilar()
    if (esVacia()) then
        write("pila vacía, no se puede desapilar")
        return null
    end(if)
    nodoSimple p
    p = primerNodo()
    d = p.retornaDato()
    borrar(p, null)
    return d
fin(desapilar)

```

Fíjese que en los métodos apilar y desapilar hemos hecho uso de los métodos insertar y borrar, los cuales fueron definidos para la clase LSL.

```

objeto tope()
    if (esVacia()) then
        write("pila vacía, no hay elemento para mostrar")
        return null
    end(if)
    nodoSimple p
    p = primerNodo()
    return p.retornaDato()
fin(tope)

```

12.4 Manejo de dos pilas en un vector. Como es muy frecuente tener que manejar más de una pila en muchas situaciones veremos cómo manejar dos pilas en un solo vector. Si **n** es el número de elementos del vector, dividimos inicialmente el vector en dos partes iguales. Llamemos **m** la variable que apunta hacia el elemento de la mitad.

La primera mitad del vector será para manejar la pila 1, y la segunda mitad del vector para manejar la pila 2.

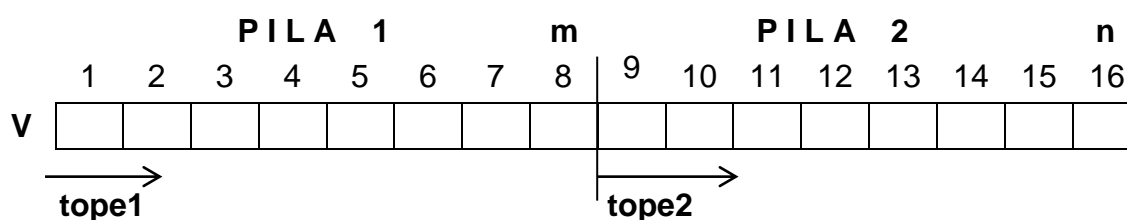
Cada pila requiere una variable **tope** para conocer en qué posición está el último dato de la pila. Llamemos estas variables **tope1** y **tope2** para manejar las pilas 1 y 2 respectivamente.

Consideremos el siguiente ejemplo:

Sea **V** el vector en el cual manejaremos las dos pilas.

El valor de **n** es 16.

Inicialmente el valor de **m** es 8. La mitad de **n**.



La variable **tope1** variará desde 1 hasta **m**.

La variable **tope2** variará desde **m+1** hasta **n**.

La pila 1 estará vacía cuando **tope1** sea igual a cero.

La pila 1 estará llena cuando **tope1** sea igual a **m**.

La pila 2 estará vacía cuando **tope2** sea igual a **m**.

la pila 2 estará llena cuando **tope2** sea igual a **n**.

Si definimos una clase denominada **dosPilas** con datos privados el vector **V**, **m**, **n**, **tope1** y **tope2** veamos cómo serán los algoritmos para manipular dicha clase.

Consideremos primero un algoritmo para apilar un dato en alguna de las dos pilas. Habrá que especificar en cuál pila es que se desea apilar. Para ello utilizaremos una variable llamada **pila**, la cual enviamos como parámetro del subprograma. Si **pila** es igual a 1 hay que apilar en la pila 1, si **pila** es igual a 2, hay que apilar en la pila 2.

Nuestro algoritmo:

```
void apilar(entero pila, objeto d)
  if (pila == 1) then
    if (tope1 == m) then
      pilaLlena(pila)
    end(if)
    tope1 = tope1 + 1
    V[tope] = d
  else
    if (tope2 == n) then
      pilaLlena(pila)
    end(if)
    tope2 = tope2 + 1
    V[tope2] = d
  end(if)
fin(apilar)
```

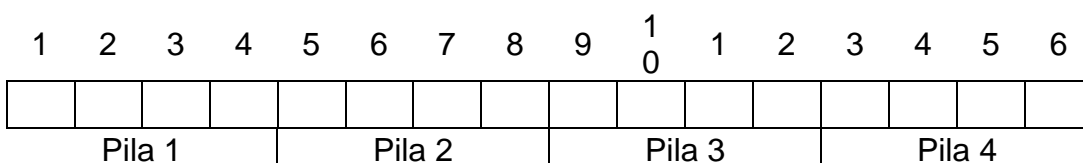
El subprograma **pilaLlena** recibe como parámetro el valor de **pila**.

La tarea de **pilaLlena** es: si el parámetro pila es 1 significa que la pila 1 es la que está llena, por tanto buscará espacio en la pila 2, en caso de que ésta no esté llena, se moverán los datos de la pila 2 una posición hacia la derecha, se actualizará **m** y se regresará al subprograma apilar para apilar en la pila 1; si es la pila 2 la que está llena buscará si hay espacio en la pila 1, en caso de haberlo, moverá los datos de la pila 2 una posición hacia la izquierda, actualizará **m** y regresará a apilar en la pila 2. Un algoritmo que efectúe esta tarea es:

```
1. void pilaLlena(int pila)
2.   int i
3.   if (pila == 1) then
4.     if (tope2 < n) then           // hay espacio en la pila 2
5.       for (i = tope2; i > m; i--) do
6.         V[i+1] = V[i]
7.       end(for)
8.       tope2 = tope2 + 1
9.       m = m+1
10.    end(if)
11.  else
12.    if (tope1 < m) then
13.      for (i = m; i < tope2; i++) do
14.        V[i - 1] = V[i]
15.      end(for)
16.      tope2 = tope2 - 1
17.      m = m - 1
18.    end(if)
19.  end(if)
20.  write("Pilas llenas")
21.  stop
22. fin(pilaLlena)
```

Como se podrá observar la operación de apilar implica mover datos en el vector, debido a que una pila puede crecer más rápido que la otra. En otras palabras, este algoritmo tiene orden de magnitud lineal, el cual se considera ineficiente.

Una mejor alternativa de diseño es la siguiente:



En nuestro ejemplo **m** vale 16 y **n** vale 4, es decir, vamos a representar 4 pilas en un vector de 16 elementos.

Necesitamos asignarle a cada pila una parte del vector, en este caso a cada pila le corresponderán 4 posiciones: la pila 1 desde la posición 1 hasta la 4, la pila 2 desde la posición 5 hasta la 8, la pila 3 desde la posición 9 hasta la 12 y la pila 4 desde la posición 13 hasta la 16.

Necesitamos conocer en cuál posición del vector comienza cada pila, para ello utilizaremos un vector que llamamos **bases**, el cual en la posición **i** contendrá la posición anterior a la que en realidad comienza la pila **i**, es decir, si **bases[i]** vale 8 significa que la porción de vector que corresponde a la pila **i** comienza en la posición 9.

Adicionalmente, necesitamos conocer la posición del vector en la cual se halla el último elemento de cada pila, para ello utilizaremos un vector que llamaremos **topes**. **topes[i]** dará la posición, en el vector **V**, en la cual se halla el último elemento de la pila **i**. Si **topes[i]** vale 7 significa que el último dato de la pila **i** está en la posición 7 del vector **V**.

Si vamos a manejar **n** pilas, entonces el vector **topes** tendrá **n** elementos y el vector **bases** tendrá **n+1** elementos.

Inicialmente los vectores de **bases** y **topes** tendrán los mismos datos indicando que cada pila se halla vacía. Es decir, la condición de que la pila **i** se halla vacía es: **topes[i] == bases[i]**.

Si tenemos el vector **V** con la siguiente configuración:

1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
a	b							c	d	e	f	g	h	i	
Pila 1				Pila 2				Pila 3				Pila 4			

Los vectores de **bases** y **topes** tendrán la siguiente información:

	1	2	3	4	5
Bases	0	4	8	12	16

	1	2	3	4
Topes	2	4	12	15

La pila 3 se encuentra llena. Observe que **topes[3]** es igual a **bases[4]**. La condición de que la pila **i** se encuentra llena es: **topes[i] == bases[i+1]**. Es por esta razón por la que el vector de **bases** tiene un elemento más que el vector de **topes**: se facilita el control de pila llena de la pila **n**. Teniendo definida la representación y las condiciones de pila vacía y pila llena desarrollaremos los algoritmos para dar las condiciones iniciales y para apilar y desapilar de cualquier pila.

Las condiciones iniciales se definen conocidos **m** y **n**, mediante las siguientes instrucciones:

```
p = m / n
for (i = 1; i <= n; i++) do
    topes[i] = (i - 1)*p
    bases[i] = (i - 1)*p
```

```
end(for)
bases[n+1] = m
El método para apilar en la pila i es:
```

```
void apilar(entero i, objeto d)
    if (topes[i] == bases[i+1]) then
        pilaLlena(i)
    end(if)
    topes[i] = topes[i] + 1
    V[topes[i]] = d
fin(apilar)
```

Donde **V** es el vector, **i** la pila en la cual se desea apilar y **d** el dato a apilar.

El subprograma para desapilar de la pila **i** es:

```
objeto desapilar(entero i)
    if (topes[i] == bases[i]) then
        pilaVacía(i)
    end(if)
    d = V[topes[i]]
    topes[i] = topes[i] - 1
    return d
fin(desapilar)
```

Donde **V** es el vector, **i** la pila de la cual se va a desapilar y **d** el dato desapilado.

En el subprograma para apilar se chequea la condición de pila llena, la cual si resulta verdadera, ocasiona la ejecución del subprograma **pilaLlena**, enviando como parámetro la variable **i**, indicando que fue esta la pila que se llenó.

El subprograma **pilaLlena** buscará en las otras pilas para ver si hay espacio disponible, en cuyo caso hará los movimientos apropiados en el vector **V**, y las actualizaciones apropiadas en los vectores de **bases** y **topes** para poder apilar el dato requerido en la pila **i**.

La táctica que sigue dicho subprograma es: primero busca en las pilas a la derecha de la pila **i** (desde la pila **i+1** hasta la pila **n**), de no hallar espacio en ninguna de estas pilas buscará en las pilas a la izquierda de la pila **i** (desde la pila **i-1** hasta la pila 1), si aún no encuentra espacio, significa que todas las pilas están llenas y la operación de apilar no se puede efectuar, por tanto detendrá el proceso emitiendo un mensaje apropiado.

Nuestro algoritmo **pilaLlena** es:

```
void pilaLlena(int i)
    j = i + 1 // busca a la derecha de la pila i
    while ((j <= n and topes[j]) == bases[j+1]) do
        j = j + 1
    end(while)
    if (j <= n) then // encontró espacio en la pila j
        k = topes[j]
        while (k > topes[i]) do // mueve los elementos
            V[k+1] = V[k] // en el vector V
            k = k - 1
        end(while)
        for (k = i+1; k <= j; k++) do // actualiza los
```



```

        topes[k] = topes[k] + 1           // vectores de topes
        bases[k] = bases[k] + 1         // y bases
    end(for)
    return                               // retorna al subprograma de apilar
end(if)
j = i - 1                               // busca a la izquierda de la pila i
while ((j > 0) and (topes[j] == bases[j+1])) do
    j = j - 1
end(while)
if (j > 0) then                          // encontró espacio en la pila j
    for (k = bases[j+1]; k < topes[i]; k++) do
        V[k] = V[k+1]                  // mueve los elementos del vector V
    end(for)
    for (k = j+1; k <= i; k++) do       // actualiza los vectores
        topes[k] = topes[k] - 1        // de topes y bases
        bases[k] = bases[k] - 1
    end(for)
    return                             // retorna al subprograma de apilar
end(if)
write("todas las pilas están llenas")
stop
fin(pilaLlena)

```

Note que el subprograma **pilaLlena** no efectúa la operación de apilar, él solamente abre espacio y retorna al subprograma **apilar** para que éste haga dicha operación.

12.6 Manejo de n Pilas como Listas Ligadas: Con **n** pilas deberán tenerse **n** listas (una por cada pila). Para no tener que manejar **n** variables **tope**, utilizaremos un vector de apuntadores que llamaremos **topes**. **topes[i]** apunta hacia el primer registro de la lista que representa la pila **i**. Ver figura 12.1. Los subprogramas para apilar y desapilar son los correspondientes a insertar un nodo al principio, en una lista simplemente ligada y a borrar el primer nodo de una lista simplemente ligada. En el subprograma para apilar el parámetro **d** es el dato a apilar y el parámetro **i** indica la pila en la cual se debe guardar el dato **d**.

```

void apilar(objeto d, int i)
    x = new nodoSimple(d)
    x.asignaLiga(topes[i])
    topes[i] = x
fin(apilar)

```

En desapilar el parámetro **i** indica la pila en la cual hay que efectuar la operación de desapilar.

```

objeto desapilar(int i)
    if (topes[i] == null) then
        pilaVacía(i)
    end(if)
    d = dato(topes[i])
    topes[i].asignaLiga(topes[i].retornaLiga())
fin(desapilar)

```

Estos subprogramas son mucho más eficientes que manejando la pila en un vector. La principal diferencia estriba en que en el subprograma para apilar se evita el proceso de **pilaLlena**.

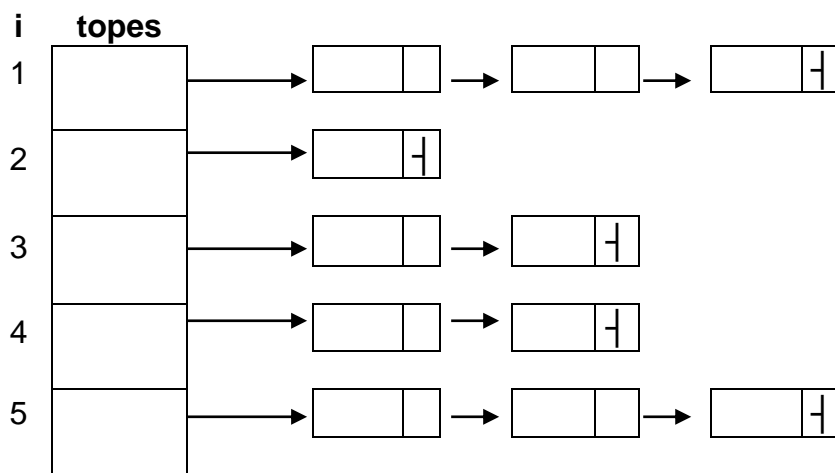


Figura 10.1

EJERCICIOS PROPUESTOS

1. Elabore algoritmo pilaLlena para la representación descrita en el módulo 12.5 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.

2. Supongamos que TEST es una función Boolean que recibe un entero y retorna un valor igual o distinto a cero. 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

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.

4. Se tiene una matriz de m filas y n columnas. En cada posición hay un cero o un uno: cero significa que se puede avanzar hacia esa casilla, uno que esa casilla está bloqueada. Teniendo

definida una casilla de entrada y una casilla de salida elabore algoritmo que determine e imprima el camino a seguir para ir desde la entrada hacia la salida.

MÓDULO 13

PILAS: EXPRESIONES.

Introducción

Vimos en el módulo anterior la definición y representación de pilas. Esta estructura dentro de lo sencilla que es, es de gran uso en el ambiente de programación de computadores. En sistemas operativos, en compiladores y en recursión, para no mencionar sino unos cuantos casos las pilas aparecen como una herramienta indispensable. Veremos en este módulo una aplicación de pilas, la cual se presenta en la manipulación de expresiones en los lenguajes de programación de computadores.

Objetivos

1. Aplicar la estructura pila en manipulación de expresiones.
2. Aprender a evaluar expresiones en prefijo, posfijo e infijo.
3. Convertir expresiones entre las diferentes formas de representación.

Preguntas básicas

1. Qué es una expresión?
2. De cuántas formas se puede escribir una expresión?
- 3.Cuál es la forma general de expresiones en infijo?
- 4.Cuál es la forma general de expresiones en prefijo?
- 5.Cuál es la forma general de expresiones en posfijo?

13.1 Notaciones para expresiones. Cuando los pioneros de la ciencia de los computadores concibieron la idea de lenguajes de programación de alto nivel, se enfrentaron a muchas dificultades técnicas. Una de las mayores fue cómo generar instrucciones en lenguaje de máquina para evaluar correctamente una expresión aritmética.

En general, hay tres formas diferentes para escribir expresiones. Si queremos indicar que hay que sumar **a** con **b**, lo podremos hacer así:

- **a+b**, notación **INFIJO**, cuya forma general es: operando operador operando.
- **ab+**, notación **POSFIFO**, cuya forma general es: operando operando operador.
- **+ab**, notación **PREFIJO**, cuya forma general es: operador operando operando.

Cuando se tiene una expresión **infijo**, el operador actúa sobre el operando que le precede y el que le sucede.

Cuando se tiene una expresión **posfijo**, el operador actúa sobre los dos operandos que le preceden.

Cuando se tiene una expresión **prefijo**, el operador actúa sobre los dos operandos que le suceden.

En realidad nosotros estamos familiarizados con escribir expresiones en **infijo**. Esa es la forma como nos educaron. Comencemos haciendo un análisis a esta notación.

Una expresión como:

$$a / b ^ c + d * e - a * c$$

puede tener varios significados; más aún, si estuviera definida únicamente, es decir, utilizando paréntesis para definir el orden de ejecución de las operaciones, aún se ve dificultoso generar una secuencia correcta y razonable de instrucciones para evaluar dicha expresión.

Afortunadamente la solución de que se dispone en la actualidad es simple y elegante. Más aún, es tan simple que este aspecto, en la elaboración de compiladores, es de los que requiere menor esfuerzo.

Una expresión es una sucesión de operandos y operadores. La expresión anterior tiene 5 operandos: a, b, c, d, y e.

Aunque en el ejemplo, las variables son de una letra, los operandos pueden ser cualquier nombre de variable o constantes en algún lenguaje de programación.

En cualquier lenguaje de programación los valores de las variables deben ser consistentes con las operaciones que se ejecutan sobre ellas. Estas operaciones son descritas por los operadores.

En la mayoría de los lenguajes de programación hay diferentes operadores, los cuales se aplican a los diferentes tipos de datos que las variables pueden almacenar.

Primero están los operadores aritméticos básicos: suma, resta, multiplicación, división, módulo y potenciación (+, -, *, /, %, ^). Otros operadores aritméticos son el más y el menos unario.

Luego, están los operadores relacionales: <, <=, =, >, >=, <>, los cuales generalmente se aplican sobre operadores aritméticos, aunque también se aplican a hileras de caracteres. ("gato" es menor que "perro" debido a su precedencia en orden alfabético). El resultado de evaluar una expresión que contiene operadores relacionales es verdadero o falso.

Existen también los operadores lógicos: & (and), | (or), ! (not), los cuales se aplican sobre expresiones o variables lógicas.

El primer problema para evaluar una expresión es decidir el orden en que se ejecutan las operaciones. Esto implica definir ese orden. Por ejemplo, si **a** = 4, **b** = 2, **c** = 2, **d** = 3 y **e** = 3 el valor de la expresión dada anteriormente podría ser:

$$\begin{aligned} &4/(2^2) + (3*3) - (4*2) \\ &(4/4) + 9 - 8 \\ &2 \end{aligned}$$

Sin embargo, la verdadera intención del programador podría haber sido:

$$\begin{aligned} &(4/2)^{(2+3)} * (3-4) * 2 \\ &(4/2)^{(5)} * (-1) * 2 \\ &(2^5) * (-2) \\ &32 * (-2) \\ &- 64 \end{aligned}$$

Obviamente, él pudo haber especificado este último orden de evaluación utilizando paréntesis:

$$(((a / b)^{(c + d)} * (e - a)) * c)$$

Para definir el orden de evaluación se ha asignado a cada operador una prioridad. Entonces, los operadores con mayor prioridad se ejecutan primero, los operadores con igual prioridad se ejecutan en el orden en que aparezcan de izquierda a derecha (asociatividad). Una muestra de prioridades de operadores es:

Operador	Prioridad
^	6
*, /	5

+, -	4
<, <=, =, >, >=, <>	3
!	2
&	1
	0

Cuando se tiene una expresión con operaciones consecutivas de la misma prioridad es necesario reglamentar cuál se ejecuta primero (esto es lo que se denomina asociatividad). En álgebra se considera a^b^c como a^b^c y esta es la norma que rige para la potenciación, la potenciación se ejecuta de derecha a izquierda, es decir, es asociativa por la derecha. Sin embargo, expresiones como $a*b/c$ ó $a+b-c$ se evalúan de izquierda a derecha, es decir, son asociativas por la izquierda.

Este orden se puede alterar utilizando paréntesis, en cuyo caso la evaluación se efectúa desde los paréntesis más internos hacia los más externos. Teniendo definido lo que es la prioridad y la asociatividad sabemos cómo se evalúa correctamente una expresión escrita en **infijo**.

13.2 Evaluación de una expresión en posfijo. Consideremos la expresión que hemos estado trabajando:

$$a / b ^ c + d * e - a * c$$

Su forma **posfijo** es: abc^*/de^*+ac^*-

Hagamos un seguimiento a la forma de evaluación.

Cada vez que se efectúe una operación almacenamos su resultado en una posición temporal.

Procesando de izquierda a derecha, el primer operador que se encuentra es el de potenciación, el cual se aplica sobre los dos operandos que le proceden que son **b** y **c**. A continuación damos una tabla del orden de ejecución de las operaciones y la forma como va quedando la expresión posfijo.

Operación		Posfijo
$r_1 = b^c$	→	ar_1/de^*+ac^*-
$r_2 = a/r_1$	→	$r_2de^*+ac^*-$
$r_3 = d * e$	→	$r_2r_3+ac^*-$
$r_4 = r_2 + r_3$	→	r_4ac^*-
$r_5 = a * c$	→	r_4r_5-
$r_6 = r_4 - r_5$	→	r_6

Y r_6 contiene el resultado.

Fíjese que la expresión se evalúa estrictamente de izquierda a derecha. Los paréntesis no son necesarios y la prioridad de los operadores no es relevante. La expresión se evalúa haciendo una búsqueda de izquierda a derecha, apilando operandos y aplicando los operadores sobre los dos últimos operandos que se hallen en la pila y colocando el resultado en la pila.

Este proceso de evaluación es mucho más simple que intentar una evaluación directa sobre la notación infijo.

Veamos ahora un algoritmo para evaluar una expresión en notación POSFIJO.

```

real evapos(string e)
    pila pp(100)
    x = siguienteToken(e)

```

```

while (x <> "|") do
  if (x in operadores) then
    opn2 = pp.desapilar()
    opn1 = pp.desapilar()
    casos de x
      *: res = opn1 * opn2
      ./: res = opn1 / opn2
      +: res = opn1 + opn2
      -: res = opn1 - opn2
    fin(casos)
    pp.apilar(res)
  else
    pp.apilar(x)
  end(if)
  x = siguienteToken(e)
end(while)
return pp.tope()
fin(evapos)

```

El anterior algoritmo utiliza una función llamada **siguienteToken()**. Cuando se tiene una expresión los operandos y operadores no necesariamente son de un carácter. En general, cualquier elemento de una expresión, operando u operador, recibe el nombre de **token**, la función **siguienteToken()**, se encarga de extraer de la expresión, el elemento a procesar, sin importar el número de caracteres que tenga.

13.3 Conversión desde infijo hacia posfijo manualmente. Veamos ahora cómo convertir una expresión en notación **infijo** a notación **posfijo**. Los pasos a seguir son:

- Parentizar completamente la expresión infijo.
- Mover los operadores a reemplazar su correspondiente paréntesis derecho.
- Borrar todos los paréntesis izquierdos.

Como ejemplo consideremos la expresión:

$$a / b^c + d * e - a * c$$

Primer paso: parentizarla completamente de acuerdo al orden de ejecución de los operadores. A cada operador le corresponderá un paréntesis izquierdo y un paréntesis derecho.

$$(((a / (b^c)) + (d * e)) - (a * c))$$

Segundo paso: mover los operadores a reemplazar su correspondiente paréntesis derecho. Al hacer los movimientos la expresión queda:

$$(((a (bc^ / (de * + (ac * -$$

Tercer paso: reescribir la expresión suprimiendo los paréntesis izquierdos. La expresión quedará:

$$abc^ / de * + ac * -$$

El problema con este método es que requiere varios pasos: parentizar la expresión, mover operadores y suprimir paréntesis izquierdos.

13.4 Conversión infijo hacia posfijo usando una pila. Si observamos bien ambas expresiones (en infijo y en posfijo) vemos que los operandos conservan el mismo orden relativo. Por tanto, si recorremos la expresión infijo de izquierda a derecha, podemos comenzar a armar la expresión posfijo, escribiendo los operandos a medida que se encuentren y el problema se reduce a un manejo de movimiento de operadores, es decir, en qué momento se debe colocar un operador en la expresión posfijo. La solución es almacenarlos en una pila y determinar el momento preciso en el cual se deben pasar a la expresión posfijo.

Por ejemplo, si tenemos la expresión $a + b * c$ y deseamos llegar a su forma posfijo $abc*+$ la secuencia de apilar y desapilar sería:

SIGUIENTE-TOKEN	PILA	EXPRESION POSFIJO
a	Vacía	a
+	+	a
b	+	ab

En este punto el SIGUIENTE_TOKEN es el operador $*$ y debemos determinar si se coloca en la pila o si desapilamos el operador $+$. Debido a que el $*$ es de mayor prioridad que el $+$ lo apilamos:

*	+ *	ab
c	+ *	abc

Hemos terminado con la expresión infijo y lo que hay que hacer es sacar los operadores que tenemos en la pila y llevarlos a la expresión obteniendo $abc*+$.

Consideremos otro ejemplo: $a*(b+c)*d$, su forma posfijo es $abc+*d*$, veamos la secuencia de operaciones para obtener esta expresión:

SIGUIENTE-TOKEN	PILA	EXPRESION POSFIJO
a	Vacía	a
*	*	a
(* (a
b	* (ab
+	* (+	ab
c	* (+	abc

En este momento el siguiente token es $)$, la acción a tomar es desapilar todos los operadores hasta encontrar el paréntesis izquierdo correspondiente y luego sacarlo de la pila sin incluirlo en la expresión posfijo:

)	*	abc+
---	---	------

El siguiente token es un asterisco, y el operador que está en la pila es también un asterisco; como son operadores de igual prioridad entonces sacamos el operador de la pila y lo llevamos a la expresión posfijo y apilamos el operador que se acabó de leer:

*	*	abc+*
d	*	abc+*d

Terminamos la expresión infijo, entonces sacamos los operadores que hay en la pila y obtenemos $abc+*d*$ que es la expresión posfija buscada.

En general, cuando se lea un operador siempre debe ser llevado a la pila, sin embargo, antes de hacer esto, debemos sacar de la pila todos los operadores cuya prioridad sea mayor o igual que la prioridad del operador que va a entrar.

Cuando se encuentre un paréntesis izquierdo siempre debe ser llevado a la pila sin sacar operadores de ella.

Cuando se encuentre un paréntesis derecho, se deben sacar todos los operadores que haya en la pila hacia la expresión posfijo, hasta hallar su correspondiente paréntesis izquierdo, el cual se borra de la pila sin llevarlo a la expresión posfijo.

Consideremos el caso de la potenciación, la cual se ejecuta de derecha a izquierda. Si tenemos la expresión a^b^c su forma posfijo es $abc^{\wedge\wedge}$

Al aplicar el proceso establecido tenemos:

SIGUIENTE-TOKEN	PILA	EXPRESION POSFIJO
a	Vacía	a
\wedge	\wedge	a
b	\wedge	ab

El siguiente token es otro operador de potenciación y como la prioridad del operador que está en la pila es igual a la del operador que va a entrar, la acción es extraer el operador de la pila e incluir el que viene, por tanto obtenemos:

\wedge	\wedge	ab^{\wedge}
c	\wedge	ab^c

Se terminó la expresión infijo, luego la expresión posfijo resultante será:

$ab^{c^{\wedge}}$

la cual, es diferente a la expresión posfija correcta que es $abc^{\wedge\wedge}$

Lo anterior nos lleva a adoptar una condición diferente para los operadores de operaciones que se ejecutan de derecha a izquierda. La solución a este problema es asignar prioridad distinta de ejecución a estos operadores para cuando están dentro de la pila y para cuando están fuera de la pila. Como el operador dentro de la pila no puede salir, le asignamos a estos operadores una prioridad mayor fuera de la pila que dentro de la pila.

Todas estas condiciones anteriores nos llevan a construir una tabla de prioridad de los operadores dentro y fuera de la pila, la cual es la siguiente:

Operador	Prioridad dentro de la pila	Prioridad fuera de la pila
\wedge	3	4
*	2	2
/	2	2
+	1	1
-	1	1
(0	4

El subprograma para convertir una expresión **infijo** a **posfijo** es:

1. **void** intopos(String e)
2. pila pp(100)
3. String x
4. x = siguienteToken(e)

```

5.      while (x <> "}") do
6.          casos de x
7.              x in operadores:
8.                  while ((no pp.esVacia() and pdp(pp.cima()) >= pfp(x)) do
9.                      y = pp.desapilar()
10.                     write(y)
11.                 end(while)
12.                 pp.apilar(x)
13.             x = ")":
14.                 while (pp.cima() <> "(") do
15.                     y = pp.desapilar()
16.                     write(y)
17.                 end(while)
18.                 pp.desapilar(1)
19.             else:
20.                 write(x)
21.             fin(casos)
22.             x = siguienteToken(e)
23.         end(while)
24.     while (no pp.esVacia()) do
25.         y = pp.desapilar()
26.         write(y)
27.     end(while)
28. end(intopos)

```

En las instrucciones 2 a 4 se definen las variables de trabajo y se accede el primer token.

Instrucciones 5 a 23 son el ciclo principal del algoritmo.

Instrucciones 8 a 12 tratan el caso en que el token sea un operador.

Instrucciones 13 a 18 tratan el caso en que el token es un cierre paréntesis,

Instrucción 20 se ejecuta cuando el token es un operando.

Instrucciones 24 a 27 llevan a la expresión en posfijo los operadores que quedaron en la pila.

EJERCICIOS PROPUESTOS

1. Elabore un algoritmo para convertir una expresión de infijo a prefijo.
2. Elabore un algoritmo para convertir una expresión de posfijo a prefijo.
3. Elabore un algoritmo para convertir una expresión de posfijo a infijo.
4. Elabore un algoritmo para convertir una expresión de prefijo a infijo.
5. Elabore un algoritmo para convertir una expresión de prefijo a posfijo.
6. Elabore un algoritmo para evaluar una expresión en infijo.
7. Elabore un algoritmo para evaluar una expresión en prefijo.

MODULO 14

RECURSION (DEFINICIÓN Y EJEMPLOS)

Introducción

Todos los algoritmos que hemos venido trabajando hasta ahora tienen la característica de que se ejecutan secuencialmente, desarrollando ciclos, procesando decisiones y ejecutando subprogramas. Algunos de esos algoritmos son demasiado complejos utilizando únicamente esas herramientas. Presentamos aquí el concepto de recursión, una técnica que facilitará la definición y el desarrollo de algoritmos para ciertas estructuras y ciertos problemas que se presentan con frecuencia en el ejercicio profesional de un desarrollador de programas por computador.

Objetivos

1. Adquirir el concepto de recursión.
2. Aprender a elaborar definiciones recursivas.
3. Construir algoritmo recursivo con base en definición recursiva.
4. Construir algoritmo recursivo con base en un ciclo.
5. Aprender a hacer seguimiento recursivo a un algoritmo recursivo.

Preguntas básicas

1. Qué es recursión?
2. Cómo se construye algoritmo recursivo con base en definición recursiva?
3. Todo ciclo se podrá convertir a algoritmo recursivo?
4. Para qué situaciones es aconsejable la recursión?

14.1 Definición. Recursión es la técnica mediante la cual se define una función o un proceso en términos de sí mismo. Es una poderosa herramienta que no ha tenido la difusión ni la acogida que debiera. Hay básicamente dos razones para ello: una, es que hay muchas personas que aprendieron el arte de programar computadores con lenguajes que no admiten esta técnica, por lo tanto no han podido disfrutar de sus beneficios; dos, que, por lo general, los algoritmos recursivos tienen una sobrecarga de tiempo en la ejecución. Sin embargo, esto no debe ser obstáculo para utilizar la recursión debido a que hay procedimientos para convertir algoritmos recursivos a versiones no recursivas e incrementar la eficiencia de éstos. Además la recursión ha tenido el mito de la complejidad, idea poco aceptable, debido a que quizá es más sencillo y más legible, en muchos casos, plantear algoritmos recursivos que su versión iterativa alterna. El objetivo principal de este módulo es aprender a pensar recursivamente

14.2 Conceptos. Digamos también, que independientemente de los computadores, esta es una técnica que los matemáticos utilizan muy frecuentemente en la definición de sus funciones. Un ejemplo clásico es la función factorial. Según los matemáticos:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n*(n-1)! & \text{si } n > 0 \end{cases}$$

Esta es una definición recursiva ya que define el factorial de **n** en términos del factorial de **n-1**. Si queremos calcular 4! Con base en esta definición seguimos los siguientes pasos:

1. $4! = 4*3!$
2. $3! = 3*2!$
3. $2! = 2*1!$
4. $1! = 1*0!$
5. $0! = 1$

Los pasos 1 a 4 son la aplicación de la segunda parte de la definición de $n!$. Fíjese que en todos estos pasos queda pendiente el producto de dos datos: en el paso 1 queda pendiente el producto

de 4 por 3!, en el paso 2 queda pendiente el producto de 3 por 2!, y así sucesivamente. Estos productos se podrán realizar cuando se haya determinado el valor del factorial.

Cuando se llega al paso 5 se aplica la primera parte de la definición y se obtiene que el factorial de 0 es uno. Teniendo este resultado nos devolvemos al paso 4 para efectuar el producto de 1 por 0! y obtenemos el resultado de 1!. Con este resultado nos devolvemos al paso tres para efectuar el producto de 2 por 1!, el cual da 2. Con este nuevo resultado nos devolvemos al paso 2 y obtenemos el resultado de 3*2!, el cual es 6, y con este nuevo resultado nos devolvemos al paso 1 para efectuar el producto entre 4 y 3!, el cual da 24. Hemos calculado 4! Haciendo uso de la definición recursiva para la función factorial. Este es un desarrollo típico de cómo trabaja una función recursiva.

Construyamos ahora un algoritmo recursivo para el factorial de un número con base en la definición que tenemos.

```
1. int factorial(int n)
2.     if (n == 0) then
3.         x = 1
4.     else
5.         x = n*factorial(n - 1)
6.     end(if)
7.     return x
8. fin(factorial)
```

Este algoritmo lo hemos construido con base en la definición dada. Basta con escribir en forma algorítmica lo planteado en la definición. Fíjese que en la instrucción 5 se invoca la función factorial. Esto es recursión: dentro del subprograma denominado factorial hay instrucciones que invocan el mismo subprograma factorial. Un subprograma es recursivo cuando dentro de él hay una o más instrucciones en las cuales se invoca a si mismo. Para familiarizarnos con elaborar definiciones recursivas y construir algoritmo recursivo con base en la definición vamos a considerar las operaciones aritméticas en el ambiente de los números naturales.

14.2.1 Operación de potenciación. Si deseamos efectuar 3 elevado a la potencia 4, recordemos que la potenciación es una sucesión de multiplicaciones, por tanto podremos escribir:

$$3^4 = 3*3*3*3$$

lo cual también podremos escribir así:

$$3^4 = 3*(3*3*3)$$

que es lo mismo que

$$3^4 = 3*3^3$$

y ahí tenemos tres a la cuatro expresado en términos de tres a la tres. Esto es recursión, definir algo en términos de sí mismo. De una manera análoga podemos seguir definiendo tres a la tres:

$$3^3 = 3*3^2$$

$$3^2 = 3*3^1$$

$$3^1 = 3*3^0$$

$$3^0 = 1$$

De acuerdo al anterior análisis podemos sintetizar la operación de exponenciación de la siguiente manera:

$$n^r = \begin{cases} 1 & \text{si } r = 0 \\ n*n^{r-1} & \text{si } r > 0 \end{cases}$$

y el algoritmo recursivo correspondiente a esta definición es:

```

1. int potencia(int n, int r)
2.     if (r == 0) then
3.         x = 1
4.     else
5.         x = n*potencia(n, r - 1)
6.     end(if)
7.     return x
8. fin(potencia)

```

Fíjese que teniendo elaborada la definición recursiva, la construcción del algoritmo recursivo es directa. Y la definición recursiva se elabora con base en el análisis que se haga sobre el problema a resolver.

14.2.2 Operación de multiplicación. Consideremos que queremos multiplicar $3*4$

$$3*4 = 3+3+3+3$$

La cual podemos escribir:

$$3*4 = 3+(3+3+3)$$

Que es lo mismo que

$$3*4 = 3+3*3$$

Desde que Ud. sea capaz de llegar a este punto Ud. ya está siendo recursivo: está definiendo la operación de multiplicación en términos de multiplicación.

Siguiendo con este mismo pensamiento tenemos:

$$3*3 = 3+3*2$$

$$3*2 = 3+3*1$$

$$3*1 = 3+3*0$$

$$3*0 = 0$$

Y con base en este análisis sintetizamos la operación de multiplicación de forma recursiva así:

$$n*r = \begin{cases} 0 & \text{si } r = 0 \\ n + n*(r - 1) & \text{si } r > 0 \end{cases}$$

y el algoritmo recursivo correspondiente a esta definición es:

```

1. int multiplica(int n, int r)
2.     if (r == 0) then
3.         x = 0
4.     else
5.         x = n + multiplica(n, r - 1)
6.     end(if)
7.     return x
8. fin(multiplica)

```

14.2.3 Operación de division. Consideremos que se quiere dividir 12 entre 4

$$12 / 4 = 1 + (12 - 4) / 4$$

es decir,

$$12 / 4 = 1 + 8 / 4$$

ahora,

$$8 / 4 = 1 + 4 / 4$$

y

$$4 / 4 = 1 + 0 / 4$$

y

$$0 / 4 = 0$$

Ahora, si quisiéramos dividir 12 entre 5 tendríamos:

$$\begin{aligned}12 / 5 &= 1 + 7 / 5 \\7 / 5 &= 1 + 2 / 5 \\2 / 5 &= 0\end{aligned}$$

Con base en este análisis una síntesis recursiva es:

$$n/r = \begin{cases} 0 & \text{si } n < r \\ 1 + (n - r)/r & \text{en caso contrario} \end{cases}$$

y el algoritmo recursivo correspondiente a esta definición es:

```
1. int divide(int n, int r)
2.     if (n < r) then
3.         x = 0
4.     else
5.         x = 1 + divide(n - r, r)
6.     end(if)
7.     return x
8. fin(divide)
```

14.2.4 Suma de los datos de un vector. Consideremos el siguiente vector el cual llamaremos V.

1	2	3	4	5
3	1	6	2	8

Figura 14.1

Nos interesa sumar los datos de dicho vector. Planteémoslo así: sumaDatos(V, 1, 5) lo cual significa: sumar los datos que hay en el vector V desde la posición 1 hasta la posición 5, por tanto:

$$\text{sumaDatos}(V, 1, 5) = V[1] + V[2] + V[3] + V[4] + V[5]$$

Dicha suma la podemos escribir también así:

$$\text{sumaDatos}(V, 1, 5) = V[1] + (V[2] + V[3] + V[4] + V[5])$$

Fíjese que hemos aplicado la ley asociativa de la suma: hemos agrupado los sumandos desde V[2] hasta V[5], por consiguiente podremos escribir:

$$\text{sumaDatos}(V, 1, 5) = V[1] + \text{sumaDatos}(V, 2, 5)$$

y esto ya es un planteamiento recursivo: estamos definiendo sumaDatos en términos de sumaDatos. Continuando con esta idea tenemos:

$$\begin{aligned}\text{sumaDatos}(V, 2, 5) &= V[2] + \text{sumaDatos}(V, 3, 5) \\ \text{sumaDatos}(V, 3, 5) &= V[3] + \text{sumaDatos}(V, 4, 5) \\ \text{sumaDatos}(V, 4, 5) &= V[4] + \text{sumaDatos}(V, 5, 5) \\ \text{sumaDatos}(V, 5, 5) &= V[5]\end{aligned}$$

lo cual podemos sintetizar con la siguiente definición recursiva:

$$\text{sumaDatos}(V, i, n) = \begin{cases} V[n] & \text{si } n = i \\ V[i] + \text{sumaDatos}(V, i+1, n) & \text{si } i < n \end{cases}$$

y con base en esta definición escribimos un algoritmo recursivo con el cual se podrán sumar los datos de un vector cualquiera que sea el rango de posiciones a sumar.

```
1. int sumaDatos(int i, int n)
2.     if (i == n) then
```

```

3.         s = V[n]
4.     else
5.         s = V[i] + sumaDatos(i+1, n)
6.     end(if)
7.     return s
8. fin(sumaDatos)

```

Observe que no hemos incluido el nombre del vector en los parámetros del subprograma puesto que este método pertenecerá a una clase vector en la cual V es un dato privado de dicha clase.

Resumiendo, para elaborar una definición recursiva se debe hacer un buen análisis del problema a resolver. Realmente no hay una fórmula que nos permita elaborar definiciones recursivas, pero es bueno tener en cuenta dos cositas: una, se debe definir siempre un punto de parada, es decir plantear una situación en la cual no haya definición recursiva, y dos toda definición recursiva se debe acercar siempre al punto de parada, esto con el fin de garantizar que nuestro algoritmo en algún momento termine.

14.3 Conversión de ciclo en subprograma recursivo. Otra de las formas de inducir al estudiante a pensar recursivamente es ver cómo se convierte un ciclo a algoritmo recursivo. Presentaremos cómo convertir un ciclo **for** en algoritmo recursivo. Los pasos a seguir son:

1. Reemplazar el ciclo **for** por sus instrucciones primitivas. Este reemplazo consiste en:
 - 1.1 Asignar valor inicial a la variable controladora del ciclo.
 - 1.2 Escribir una instrucción **if** que compare el valor de la variable controladora del ciclo con el valor final de dicha variable.
 - 1.3 Rotular el **if** escrito en el paso 1.2.
 - 1.4 Escribir las instrucciones propias del ciclo.
 - 1.5 Incrementar la variable controladora del ciclo.
 - 1.6 Transferir el control (**goto**) al label definido en el paso 1.3.
 - 1.7 Cerrar el **if** definido en el paso 1.2.
2. El subprograma recursivo lo conformarán todas las instrucciones del ciclo y su parámetro es la variable controladora del ciclo.
3. En el programa llamante se suprimen todas las instrucciones del ciclo y se reemplaza la instrucción de asignación del valor inicial a la variable controladora del ciclo, por una llamada al subprograma recursivo. El valor del parámetro en esta llamada es el valor inicial de la variable controladora del ciclo.
4. En el subprograma recursivo se reemplazan las instrucciones de incremento de la variable controladora del ciclo y de transferencia de control (**goto**) por una llamada recursiva, con parámetro la variable controladora del ciclo modificada.

Consideremos el siguiente ejemplo. Sea un programa principal:

```

1. read(n)
2. x = 0
3. for (i = 1; i <= n; i++) do
4.     x = x + 1
5. end(for)
6. write(x)

```

n y **x** son variables globales.

Paso 1: Reemplazar el ciclo **for** por sus instrucciones elementales:

```

1.     read(n)
2.     x = 0
3.     i = 1

```

```

4. L1:   if i <= n then
5.         x = x + 1
6.         i = i + 1
7.         goto L1
8.     end(if)
9.     write(x)

```

Las instrucciones del ciclo **for** (el for y el end(for)) las hemos reemplazado por las instrucciones 3, 4, 6 y 7 conforme a los pasos descritos en el numeral 1.

Paso 2: El subprograma recursivo lo conforman:

```

L1: (if i <= n) then
      x = x + 1
      i = i + 1
      goto L1
end(if)

```

Dicho subprograma lo llamaremos **cicloRecursivo**

Paso 3: El programa principal queda así:

```

read(n)
x = 0
cicloRecursivo(1)
write(x)

```

Paso 4: El subprograma recursivo queda:

```

void cicloRecursivo(i)
  if i <= n then
    x = x + 1
    cicloRecursivo(i+1)
  end(if)
fin(cicloRecursivo)

```

Hemos construido así un algoritmo recursivo con base en un ciclo for. Cualquiera que sea la instrucción de ciclo que se tenga, siempre será posible convertirlo en algoritmo recursivo siguiendo los pasos definidos.

14.4 Seguimiento a algoritmos recursivos. Veremos ahora cómo hacer prueba de escritorio a un algoritmo recursivo. Para ello repasemos algunos conceptos importantes.

Variable global: es una variable definida externamente al subprograma recursivo y que es trabajada y modificada dentro del subprograma. Toda modificación que sufra dentro del subprograma afecta su valor externo.

Variable local: es una variable definida dentro del subprograma. Nace y muere dentro del subprograma. Es independiente del programa llamante o principal.

Parámetros del subprograma: Son los datos que recibe el subprograma. Hay básicamente dos formas de recibir dichos datos: por **valor**, o por **variable o referencia**.

Parámetros por valor: son datos o variables que recibe el subprograma. Cualquier modificación que sufran estas variables dentro del subprograma es vigente sólo dentro del subprograma, es decir, el contenido externo de la variable no sufre ninguna modificación.

Parámetros por variable o referencia: son datos que recibe el subprograma en nombres de variables. Toda modificación que se haga dentro del subprograma al contenido de estas variables afecta la variable en el programa llamante. Se comportan como si fueran variables globales.

Dirección de retorno: se refiere a la instrucción donde debe continuar la ejecución de un programa o subprograma cuando termina la ejecución de otro subprograma que fue invocado desde el primero. En esto de las direcciones de retorno hay que distinguir entre los dos tipos de subprogramas que existen: Funciones y Procedimientos. Los procedimientos son subprogramas tipo void, mientras que una función es cualquier subprograma que no es tipo void. Recordemos que las funciones retornan un valor, mientras que los procedimientos ejecutan alguna tarea específica y el nombre de ellos es simplemente un nombre para poderlos referenciar.

Teniendo en cuenta esta diferencia, la dirección de retorno para las funciones es la misma instrucción que contiene la llamada a la función, mientras que para los procedimientos la dirección de retorno es la instrucción siguiente a la instrucción que hizo la llamada al procedimiento.

La recursión se presenta cuando dentro de las instrucciones de un subprograma existe una o más llamadas a la ejecución del mismo subprograma.

Debido entonces, a que antes de terminarse la ejecución del subprograma se reinicia de nuevo su ejecución, y así sucesivamente, se podría pensar que la ejecución del subprograma se repite indefinidamente; es por esto por lo que cuando se presenta recursión, la llamada recursiva debe estar sujeta a alguna condición. Cuando esta condición no se cumple no habrá llamada recursiva y la ejecución del subprograma de alguna forma terminará.

Considerando el caso general, un subprograma (**function** o **procedure**) tiene uno o más parámetros de llamada. Dependiendo del valor de uno o varios de esos parámetros se hace llamada recursiva o no. Esta llamada recursiva enviará nuevos valores de los parámetros con los cuales se determinará si hay o no nuevas llamadas recursivas. Cuando ocurra que no se hizo llamada recursiva el subprograma terminará la ejecución de esa llamada y, como en la ejecución de cualquier subprograma, el control retornará al programa llamante (en este caso el mismo subprograma), al sitio apropiado, a continuar con una ejecución interrumpida.

Cuando se inicia la ejecución de un subprograma recursivo, es decir, cuando es invocado por primera vez, se asignan unos valores iniciales a los parámetros del subprograma. A medida que avanza la ejecución se van asignando y/o alterando los valores correspondientes a los parámetros y a algunas variables propias del subprograma (variables locales), y dependiendo de alguna condición se hará o no llamada recursiva. Cuando se hace la llamada recursiva, los parámetros y variables locales tienen algún valor almacenado, los cuales hay que conservar para que cuando se regrese a continuar con la primera llamada, o sea, cuando se termine con la llamada recursiva, se continúe con los valores correctos.

Debido a que la llamada recursiva se puede ejecutar un sinnúmero de veces, por cada una de ellas hay que conservar los valores de variables locales y parámetros correspondientes a esa llamada, para que cuando regrese a continuarla, lo haga con los valores correctos.

La forma de recuperar estos datos es en forma inversa a la que se guardan, por lo tanto, la estructura apropiada para guardarlos es una pila. Por consiguiente, para hacer seguimiento a un algoritmo recursivo utilizaremos una pila.

Resumiendo todo lo anterior, los pasos para hacer seguimiento a un algoritmo recursivo son los siguientes:

1. Definir una pila vacía.
2. Identificar instrucciones de retorno y asignarles label (cada label es una dirección de retorno).
3. Cada vez que se encuentre una llamada recursiva guardar en la pila:
 - parámetros del subprograma,
 - variables locales,
 - dirección de retorno y

- nombre de la función, si se trata de una función.
4. Asignar los nuevos valores de llamada a los parámetros.
 5. Comenzar de nuevo la ejecución con los nuevos datos de los parámetros.
 6. Cada vez que se termine una ejecución sacar de la pila los datos correspondientes a una llamada recursiva y regresar a la dirección de retorno desapilada a continuar la ejecución con los datos que se sacaron de la pila.
 7. Cuando no haya más datos en la pila es señal de que se terminó el proceso y se regresa al programa llamante a continuar su ejecución.

14.4.1 Seguimiento (prueba de escritorio) a un subprograma tipo void. Consideremos el siguiente programa principal con su correspondiente subprograma:

```

1. programaPrincipal
2.   read(n)
3.   x = 0
4.   pp(1)
5.   write(x)
6. fin(programaPrincipal)

7. void pp(int i)
8.   if i <= n then
9.       x = x + 1
10.      pp(i+1)
11.   end(if)
12. fin(pp)

```

En los cuales las variables **n** y **x** son globales.

En virtud del primer paso definiremos una pila vacía en la cual se almacenarán los datos correspondientes a la dirección de retorno y el parámetro por valor **i**. El segundo paso es identificar instrucciones de retorno y asignarles label: en el programa principal la instrucción de retorno es la instrucción `write(x)` y en el subprograma es la instrucción `end(if)` ya que son las instrucciones siguientes a la llamada de `pp` el cual es un subprograma tipo void (recuerde que cuando el subprograma es tipo void la instrucción de retorno es la instrucción siguiente a la invocación del subprograma). Teniendo rotuladas las instrucciones de retorno el escenario es:

```

1. programaPrincipal
2.   read(n)
3.   x = 0
4.   pp(1)
5. L0: write(x)
6. fin(programaPrincipal)

1. void pp(int i)
2.   if i <= n then
3.       x = x + 1
4.       pp(i+1)
5. L1: end(if)
6. fin(pp)

```

Ejecutemos el programa principal: el valor leído para **n** es 4, a **x** se le asigna 0 y encontramos la llamada al subprograma `pp`, instrucción 4, el cual se le envía como parámetro un 1. La pila queda así:

L0	1
Dr	i

Esta primera línea significa: se va a ejecutar el subprograma **pp** con la **i** valiendo 1, cuando termine esta ejecución retornará a la instrucción rotulada L0 para continuar ejecutando el programa principal.

Ejecutemos el subprograma pp con la i valiendo 1: en la instrucción 8 se compara la i con la n (1 con 4), como el resultado de esta comparación es verdadero entra a ejecutar las instrucciones dentro del if: a x le suma 1 e invoca el subprograma pp enviando como parámetro i+1, es decir, 2. Nuestra pila queda así:

L1	2
L0	1

Dr i

Esa segunda línea en la pila significa: se va a ejecutar el subprograma pp con la i valiendo 2, cuando termine esta ejecución retornará a la instrucción rotulada L1 para continuar con la ejecución de pp con la i valiendo 1. Comenzamos a ejecutar nuevamente el subprograma pp con la i valiendo 2: nuevamente, en la instrucción 8 se compara la i con la n (2 con 4), como el resultado de esta comparación es verdadero entra a ejecutar las instrucciones dentro del if: a x le suma 1 e invoca el subprograma pp enviando como parámetro i+1, es decir, 3. Nuestra pila queda así:

L1	3
L1	2
L0	1

Dr i

Esa segunda línea en la pila significa: se va a ejecutar el subprograma pp con la i valiendo 3, cuando termine esta ejecución retornará a la instrucción rotulada L1 para continuar con la ejecución de pp con la i valiendo 2. Al ejecutar este nuevo llamado la pila queda:

L1	4
L1	3
L1	2
L0	1

Dr i

Se ejecuta nuevamente el subprograma pp con la i valiendo 4 y la pila queda:

L1	5
L1	4
L1	3
L1	2
L0	1

Dr i

En este punto, al ejecutar el subprograma pp con la i valiendo 5 el resultado de la comparación en la instrucción 8 es falso, por consiguiente, se salta todas las instrucciones pertenecientes al if (instrucciones 9 a 11) y continúa en la instrucción siguiente al end(if). Dicha instrucción es el fin del subprograma, lo cual significa que terminó de ejecutar pp con la i valiendo 5, por tanto, desafila los datos que hay en el tope de la pila y regresa a la instrucción cuyo label desapiló (L1) para continuar la ejecución de pp con la i valiendo 4. En este momento la pila está así:

L1	4
L1	3

L1	2
L0	1

Dr i

Ha regresado a la instrucción 11 (end(if)) a continuar la ejecución del subprograma pp con la i valiendo 3: ejecuta la instrucción 11 y continúa con la 12, la cual es fin del subprograma, por tanto desapila los datos en el tope y retorna la instrucción cuyo label desapiló (L1). La pila está así:

L1	3
L1	2
L0	1

Dr i

Ha regresado nuevamente a la instrucción 11 (end(if)) a continuar la ejecución del subprograma pp con la i valiendo 2: ejecuta la instrucción 11 y continúa con la 12, la cual es fin del subprograma, por tanto desapila los datos en el tope y retorna la instrucción cuyo label desapiló (L1). La pila queda así:

L1	2
L0	1

Dr i

Regresa nuevamente a la instrucción 11 (end(if)) a continuar la ejecución del subprograma pp con la i valiendo 1: ejecuta la instrucción 11 y continúa con la 12, la cual es fin del subprograma, por tanto desapila los datos en el tope y retorna la instrucción cuyo label desapiló (L1). La pila queda así:

L0	1

Dr i

Al regresar nuevamente a la instrucción 11 (end(if)) a continuar la ejecución del subprograma pp con la i valiendo 1: ejecuta la instrucción 11 y continúa con la 12, la cual es fin del subprograma, por tanto desapila los datos en el tope y retorna la instrucción cuyo label desapiló (L0). Al regresar a L0 (instrucción 5) simplemente escribe el valor de x y termina la ejecución del programa principal.

Es supremamente importante que entienda este seguimiento. Es la base para entender cómo trabaja la recursión.

14.4.2 Seguimiento (prueba de escritorio) a un subprograma que no es tipo void.

Consideremos el siguiente programa principal con su correspondiente subprograma el cual no es tipo void:

1. programaPrincipal
2. read(n)
3. m = factorial(n)
4. write(m)
5. fin(programaPrincipal)

6. int factorial(n)
7. if (n == 0) then
8. x = 1
9. else
10. x = n*factorial(n – 1)
11. end(if)
12. return x
13. fin(factorial)

El primer paso es definir una pila vacía, la cual contendrá los datos correspondientes a: dirección de retorno, n (parámetro de la función), x (variable local) y un cuarto dato que tendrá el mismo nombre de la función y que nos ayudará a explicar el seguimiento.

Las instrucciones de retorno son la 3 y la 10. En estas instrucciones se encuentra la llamada al subprograma factorial, el cual no es de tipo void, por tanto la dirección de retorno es la misma instrucción que contiene el llamado al subprograma. Llamemos L0 la dirección de retorno correspondiente a la instrucción 3 y L1 la dirección de retorno correspondiente a la instrucción 10. Nuestro escenario es:

```

1.  programaPrincipal
2.      read(n)
3.  L0:    m = factorial(n)
4.      write(m)
5.  fin(programaPrincipal)

6.  int factorial(n)
7.      if (n == 0) then
8.          x = 1
9.      else
10. L1:     x = n*factorial(n - 1)
11.     end(if)
12.     return x
13. fin(factorial)

```

Al ejecutar el programa principal lo primero es leer el valor de n (digamos que leyó 4), luego ejecuta la instrucción 3, en la cual se halla una llamada al subprograma factorial, por tanto, la pila quedará:

L0	4		
Dr	n	x	factorial

Significando que se va a ejecutar el subprograma factorial con la n valiendo 4, y que cuando termine regresará a la instrucción rotulada L0 para asignarle a m lo que retorna el subprograma factorial y continuar la ejecución del programa principal.

Al ejecutar el subprograma factorial con n=4, la primera instrucción (instrucción 7) es comparar n con 0: en este caso el resultado de la comparación es falso, por tanto continuará ejecutando la instrucción 10. Al ejecutar la instrucción 10 encuentra el llamado al subprograma factorial por tanto, debe almacenar en la pila los datos correspondientes a esta interrupción. La pila queda así:

L1	3		
L0	4		
Dr	n	x	factorial

Esa segunda línea significa: se va a ejecutar el subprograma factorial con la n valiendo 3, cuando termine, regresará a la instrucción rotulada L1 para continuar ejecutando dicha instrucción, es decir multiplicar el valor de n (n=4) por el dato que retorne la llamada ejecutada y asignarle dicho resultado a x.

Al ejecutar de nuevo el subprograma factorial con n=3, el resultado de la comparación de la instrucción 7 es nuevamente falso, por consiguiente volverá a ejecutar la instrucción 10, que es la que contiene la llamada recursiva. Al ejecutar esta instrucción la pila queda así:

--	--	--	--

L1	2		
L1	3		
L0	4		
Dr n x factorial			

Esa tercera línea significa: se va a ejecutar el subprograma factorial con la n valiendo 2, cuando termine, regresará a la instrucción rotulada L1 para continuar ejecutando dicha instrucción, es decir multiplicar el valor de n ($n=3$) por el dato que retorne la llamada ejecutada y asignarle dicho resultado a x.

La ejecución del subprograma factorial con n valiendo 2 y 1 respectivamente nos producirá la pila así:

L1	0		
L1	1		
L1	2		
L1	3		
L0	4		
Dr n x factorial			

En este punto hay que ejecutar el subprograma factorial con n valiendo 0: el resultado de ejecutar la comparación de la instrucción 7 es verdadero, por tanto, ejecuta la instrucción 8, la cual le asigna 1 a x, y continúa la ejecución con la instrucción 12. En este punto la pila está así:

L1	0	1	
L1	1		
L1	2		
L1	3		
L0	4		
Dr n x factorial			

Ejecutar la instrucción 12 es retornar x. La instrucción return significa que terminó de ejecutar la llamada correspondiente a $n = 0$, por tanto hay que desapilar los datos que hay en el tope de la pila y regresar a la instrucción cuyo label se desapiló (instrucción rotulada L1) para continuar la ejecución del subprograma factorial con la n valiendo 1. Al ejecutar el return la pila queda:

L1	1		1
L1	2		
L1	3		
L0	4		
Dr n x factorial			

Estando la pila en esta situación se ejecuta totalmente la instrucción 10: lo que retornó la llamada que se terminó lo guardamos en la pila en la columna llamada factorial. Entonces, la ejecución de la instrucción 10 es multiplicar lo que hay en factorial por n y asignarle el resultado a x. La pila queda:

L1	1	1	1
L1	2		
L1	3		
L0	4		
Dr n x factorial			

Al terminar de ejecutar totalmente la instrucción 10 continúa con la instrucción 12, la cual es return x, por tanto hay que desapilar los datos que hay en el tope de la pila y regresar a la instrucción cuyo label se desapiló (instrucción rotulada L1) para continuar la ejecución del subprograma factorial con la n valiendo 2. Al ejecutar el return la pila queda:

L1	2		1
L1	3		
L0	4		
Dr	n	x	factorial

Se ejecuta nuevamente la instrucción 10 en su totalidad y la pila queda:

L1	2	2	1
L1	3		
L0	4		
Dr	n	x	factorial

Continúa nuevamente con la instrucción 12, la cual es el return y la pila queda:

L1	3		2
L0	4		
Dr	n	x	factorial

Ejecuta nuevamente la instrucción 10 y la pila queda:

L1	3	6	2
L0	4		
Dr	n	x	factorial

Continúa nuevamente con la instrucción 12 y la pila queda:

L0	4		6
Dr	n	x	factorial

Termina de ejecutar nuevamente la instrucción 10 y la pila queda:

L0	4	24	6
Dr	n	x	factorial

Ejecuta nuevamente la instrucción 12 y retorna a la instrucción rotulada L0 (instrucción 3) y le asigna 24 a la variable m. Continúa con la instrucción 4, escribe m y termina la ejecución del programa principal.

EJERCICIOS PROPUESTOS

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

2. Escriba una función recursiva que calcule el producto de cualesquier cantidad de elementos de un vector dado. Con base en dicha función escriba el algoritmo recursivo correspondiente.

3. La función de Ackerman se define así:

$\text{ack}(m,n) = n+1$ si $m=0$
 $\text{ack}(m,n) = \text{ack}(m-1,1)$ si $n=0$
 $\text{ack}(m,n) = \text{ack}(m-1, \text{ack}(m, n-1))$ en otro caso.

Escriba un algoritmo recursivo para evaluar dicha función.

4. Escriba algoritmo recursivo para generar e imprimir el enésimo término de la serie de fibonacci. Recuerde que el enésimo término de la serie de fibonacci es el resultado de la suma de los dos términos anteriores.

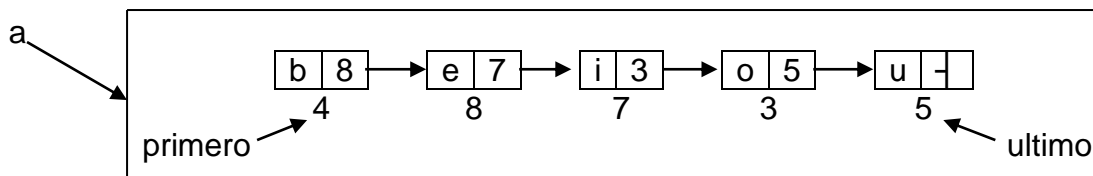
5. Haga seguimiento a los siguientes algoritmos recursivos y determine la diferencia entre ellos.

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

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

Al ser invocados por el siguiente programa principal:

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



MODULO 15

RECURSION (TORRES DE HANOI Y PERMUTACIONES)

Introducción

Ya vimos en el módulo anterior los principios y conceptos de lo que es la recursión. Vamos a tratar en este módulo un par de ejercicios que son bastante ilustrativos para el uso de esta técnica: el clásico de las torres de Hanoi y una solución al problema de las permutaciones

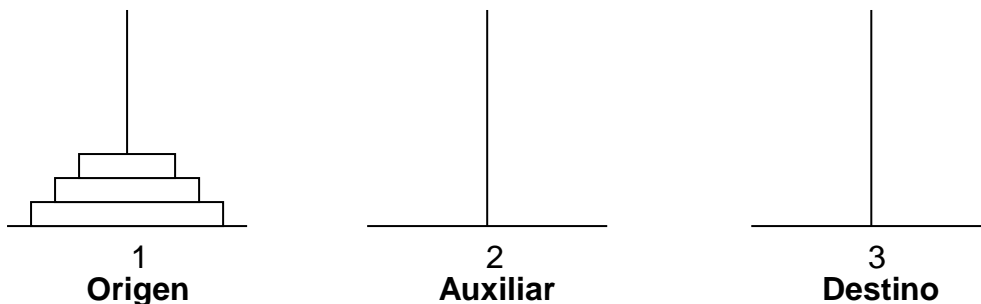
Objetivos

1. Plantear solución recursiva a problemas más complejos.
2. Construir algoritmo recursivo con base en el planteamiento recursivo.
3. Hacer seguimiento a algoritmos que tienen más de una llamada recursiva.
4. Hacer seguimiento a algoritmos que tienen llamada recursiva dentro de un ciclo.

Preguntas básicas

1. En qué consiste el entretenimiento de las torres de Hanoi?
2. En qué consiste el problema de las permutaciones?
3. Con base en qué se hace el análisis para construir un algoritmo que resuelva el problema de las torres de Hanoi?
- 4.Cuál es el punto de parada en la solución a las torres de Hanoi?
- 5.Cuál es el punto de parada en la solución al problema de las permutaciones?

15.1 TORRES DE HANOI. Consideremos otro problema clásico en materia de recursión: el algoritmo de las Torres de Hanoi. Analizaremos el problema y plantearemos su solución recursiva y escribiremos el correspondiente algoritmo. El problema en realidad es un juego que consiste en lo siguiente: se tienen 3 columnas numeradas 1, 2, y 3 y en alguna de ellas hay n aros de diferente tamaño, colocados de tal forma que el aro más grande se halla en el fondo y el aro más pequeño en la parte superior. La columna en la que se hallan los aros se denomina columna **origen**. El objetivo es trasladar los aros desde la columna origen hacia otra de las columnas, denominada **destino**, utilizando la tercera columna como **auxiliar**, cumpliendo siempre dos condiciones: sólo se puede mover una aro a la vez y no podrá nunca haber un aro de algún tamaño encima de un aro de menor tamaño. Esquemáticamente la posición inicial podría ser la siguiente:



En la columna 1 tenemos 3 aros, los cuales debemos pasar hacia la columna 3, utilizando la columna 2 como auxiliar. Nuestro programa de computador debe imprimir los movimientos apropiados para efectuar el traslado, cumpliendo las condiciones dadas. Una secuencia de movimientos para nuestro ejemplo es:

1. $1 \rightarrow 3$
2. $1 \rightarrow 2$
3. $3 \rightarrow 2$
4. $1 \rightarrow 3$
5. $2 \rightarrow 1$
6. $2 \rightarrow 3$
7. $1 \rightarrow 3$

La cual, si analizamos detenidamente, la podemos descomponer de esta forma:

Tenemos una torre de 3 aros ($n=3$),

Los primeros 3 movimientos trasladan una torre de 2 aros ($n - 1$ aros) desde la columna **origen** hacia la columna **auxiliar**,

El movimiento 4 traslada el aro de la base de la columna **origen** hacia la columna **destino**,

Los movimientos 5 a 7 trasladan torre de 2 aros ($n - 1$ aros) desde la columna **auxiliar** hacia la columna **destino**.

Diciéndolo de otra forma, para poder llevar el aro que está en la base de una torre de n aros, desde una columna origen hacia una columna destino tengo que trasladar primero $n - 1$ aros desde la columna origen hacia la columna auxiliar, luego muevo el aro (imprimo el movimiento) desde origen hacia destino y por último traslado la torre de $n - 1$ aros que tengo en la columna auxiliar hacia la columna destino. Escribiéndolo en una forma más esquemática tendremos:

Mover torre de n aros desde Origen hacia Destino usando Auxiliar es:

1. Mover torre de $n - 1$ aros desde Origen hacia Auxiliar.
2. Mover aro desde Origen hacia Destino.
3. Mover torre de $n - 1$ aros desde Auxiliar hacia Destino.

Lo cual es un planteamiento recursivo del problema, ya que estoy definiendo el proceso de mover torre en términos de mover torre y eso es recursión: definir algo en términos de sí mismo. Dicho proceso se repite hasta que no haya aros para mover, es decir, hasta que n sea igual a cero. El algoritmo recursivo simplemente consistirá en escribir los tres pasos definidos en forma de algoritmo. Estos tres pasos se ejecutan siempre y cuando haya aros para mover, es decir si n es mayor que cero. El algoritmo para mover torre de n aros es entonces:

1. void moverTorre(entero n , entero o , entero a , entero d)
2. if ($n > 0$) then
3. moverTorre($n - 1$, o , d , a)
4. L1: write(o , "--->", d)
5. moverTorre($n - 1$, a , o , d)
6. L2: end(if)
7. fin(moverTorre)

Los parámetros del subprograma, los cuales son todos parámetros por valor, son:

n = número de aros
 o = columna origen
 a = columna auxiliar
 d = columna destino

Veamos ahora cómo hacer el seguimiento a este algoritmo, con una llamada para mover 2 aros desde la columna 1 hacia la columna 3 utilizando como auxiliar la columna 2. Sea este el programa llamante:

```
----  
----  
moverTorre(2, 1, 2, 3)  
L0 : ----  
----
```

Nuestro primer paso será definir las direcciones de retorno, las cuales, como es un subprograma tipo void serán las instrucciones siguientes a las instrucciones donde haya llamada al subprograma moverTorre. En el programa llamante llamamos esta instrucción L0 y en el

subprograma moverTorre las llamamos L1 y L2 respectivamente, que corresponden a las instrucciones 4 y 6 de nuestro algoritmo moverTorre.

Utilizaremos una pila en la cual guardaremos la información correspondiente a los parámetros n, o, a, d y a las direcciones de retorno. Variables locales no hay. Al ejecutarse la primera llamada nuestra pila quedará así:

L0	2	1	2	3
DR	n	o	a	d

indicando que se va a ejecutar el subprograma moverTorre con n=2, o=1, a=2 y d=3 y que cuando termine, regresará a la instrucción llamada L0 a continuar ejecutando el programa llamante. Al comenzar la ejecución del subprograma moverTorre la primera instrucción pregunta si n es mayor que cero, como el resultado es verdadero ejecutará la instrucción 3, la cual es una llamada recursiva, por tanto guardará en la pila los datos correspondientes a la nueva llamada y la pila quedará así:

L1	1	1	3	2
L0	2	1	2	3
DR	n	o	a	d

indicando que va a ejecutar el subprograma moverTorre con n=1, o=1, a=3 y d=2 y que cuando termine regresará a la instrucción rotulada L1 a continuar ejecutando moverTorre con los datos de n, o, a y d que están inmediatamente debajo en la pila.

Al comenzar a ejecutar de nuevo moverTorre, la condición n mayor que cero, es nuevamente verdadera y ejecutará de nuevo moverTorre con n=0, o=1, a=2 y d=3 y la dirección de retorno será L1. El estado de la pila será entonces el siguiente:

L1	0	1	2	3
L1	1	1	3	2
L0	2	1	2	3
DR	n	o	a	d

indicando que se va ejecutar moverTorre con n=0, o=1, a=2 y d=3 y que cuando termine regresará a la instrucción L1 a continuar ejecutando una llamada interrumpida.

Como n es igual a cero la condición de la instrucción 2 es falsa y la ejecución continúa en la instrucción siguiente al end(if), la cual es el fin del subprograma, lo que significa que terminó de ejecutar dicha llamada y por consiguiente desapilará los datos correspondientes a esta llamada y regresará a la instrucción rotulada L1 a continuar la ejecución del subprograma moverTorre con los datos que están inmediatamente debajo. El estado de la pila es entonces el siguiente:

L1	1	1	3	2
L0	2	1	2	3
DR	n	o	a	d

La instrucción L1 dice que imprima **o** y **d**, por tanto imprime **1 ---> 2** y continúa con la siguiente instrucción que es un nuevo llamado a moverTorre, con n=0, o=3, a=1 y d=2 y dirección de retorno L2. La pila queda así:

L2	0	3	1	2
L1	1	1	3	2
L0	2	1	2	3
DR	n	o	a	d

indicando que ejecutará moverTorre con los datos que hay en el tope y que cuando termine regresará a la instrucción rotulada L2 a continuar la ejecución con los datos que están inmediatamente debajo.

La ejecución de moverTorre con estos datos ocasiona que la condición de la instrucción 2 sea falsa y la ejecución continúa en la instrucción siguiente al end(if), la cual es el fin del subprograma, por tanto desapilará los datos correspondientes a esta llamada y regresará a la instrucción rotulada L2 a continuar la llamada anterior que se había interrumpido y cuyos datos están en el nivel inmediatamente debajo en la pila. El estado de la pila es el siguiente:

L1	1	1	3	2
L0	2	1	2	3
DR	n	o	a	d

Al regresar a la instrucción rotulada L2, ésta es el end(if), y la instrucción siguiente es el fin del subprograma, lo que significa que terminó de ejecutar esta llamada, por tanto desapila los datos correspondientes a esta llamada y regresa a la instrucción rotulada L1 a continuar la ejecución de moverTorre con los datos que están inmediatamente debajo. El estado de la pila es el siguiente:

L0	2	1	2	3
DR	n	o	a	d

La instrucción L1 es write(o, "→", d), luego escribe **1 ---> 3**, que es el siguiente movimiento y continúa con la instrucción siguiente al write, la cual es una llamada recursiva al subprograma moverTorre con n=1, o=2, a=1 y d=3 y dirección de retorno L2.

El estado de la pila es el siguiente:

L2	1	2	1	3
L0	2	1	2	3
DR	n	o	a	d

Al comenzar a ejecutar de nuevo el subprograma moverTorre con estos datos, la condición de la instrucción 2 es verdadera y ejecuta la instrucción 3, la cual es una llamada a moverTorre con n=0, o=2, a=3 y d=1 y dirección de retorno L1. La pila queda así:

L1	0	2	3	1
L2	1	2	1	3
L0	2	1	2	3
DR	n	o	a	d

La ejecución de moverTorre con estos datos hacen que la instrucción 2 sea falsa y ocasionan la terminación de esta llamada, por tanto desapila los datos del tope y retorna a la instrucción L1 a continuar la llamada interrumpida cuyos datos se hallan inmediatamente debajo. El estado de la pila es el siguiente:

L2	1	2	1	3
L0	2	1	2	3
DR	n	o	a	d

La instrucción L1 dice que imprima el contenido de o y d, o sea, que imprime '2 → 3' y continúa con la instrucción siguiente que es otra llamada a moverTorre con n=0, o=1, a=2 y d=3 y dirección de retorno L2. El estado de la pila será entonces:

L2	0	1	2	3
L2	1	2	1	3
L0	2	1	2	3
DR	n	o	a	d

La ejecución con estos últimos datos no ocasionan una nueva llamada ya que n=0 da por terminada la ejecución del subprograma, luego desapila estos datos y regresa a la instrucción llamada L2 a continuar ejecutando la llamada anterior. El estado de la pila es:

L2	1	2	1	3
L0	2	1	2	3
DR	n	o	a	d

La instrucción L2 es el end(if) y luego sigue el fin del subprograma, o sea que termina esta llamada y por tanto desapila los datos correspondientes y regresa a la instrucción L2 a continuar con la llamada cuyos datos están inmediatamente debajo en la pila. El estado de la pila es:

L0	2	1	2	3
DR	n	o	a	d

La instrucción L2, según hemos visto antes, da por terminado el proceso, entonces desapila y retorna a la instrucción rotulada L0 a continuar ejecutando el programa llamante.

Es muy importante que el lector haya entendido perfectamente el anterior seguimiento para que pueda avanzar en el tema.

15.2 PERMUTACIONES El problema de las permutaciones es uno de los clásicos en análisis combinatorio. Veamos cómo analizar y escribir un algoritmo que efectúe dicha labor. Consideraremos que los datos a permutar se hallan en un vector de n elementos llamado V . En nuestro ejemplo $n = 11$.

	1	2	3	4	5	6	7	8	9	10	11
V	a	b	c	d	e	f	g	h	i	j	k

Si deseo producir las permutaciones de n elementos, dejo fijo el primer dato (el de la posición 1) y produzco las permutaciones de los $n - 1$ datos restantes (de la posición 2 hasta la n , ($n=11$)); cuando haya terminado con estas permutaciones intercambio el dato de la posición 1 (que estaba fija) con el dato de la posición 2 y vuelvo a producir las permutaciones de los datos desde la posición 2 hasta la posición n ($n=11$); terminado esto, intercambio el dato de la posición 1 con el dato de la posición 3 y nuevamente repito permutaciones desde la posición 2 hasta la 11. En otras palabras, tengo que intercambiar el dato de la posición 1 con cada uno de los demás datos y por cada intercambio debo hacer las permutaciones de los $n - 1$ datos restantes, de la posición 2 a la 11.

Para hacer las permutaciones de los datos desde la posición 2 hasta la posición n ($n=11$) el proceso es análogo: dejo fijo el dato de la posición 2 y produzco las permutaciones de los datos desde la posición 3 hasta la posición n ($n=11$); terminadas estas permutaciones intercambio el dato de la posición 2 con el dato de la posición 3 y nuevamente produzco las permutaciones de los datos desde la posición 3 hasta la posición n . O sea, el dato de la posición 2 lo debo intercambiar con todos los datos desde la posición 3 hasta la posición n y por cada intercambio producir las permutaciones desde la posición 3 hasta la posición n .

Generalizando el anterior proceso, debemos manejar una variable que indique a partir de qué posición es que se desean hacer permutaciones, llamemos i dicha variable, y otra variable que indique hasta qué posición se desean hacer las permutaciones, llamemos n esta otra variable.

El dato de la posición i debemos intercambiarlo con todos los datos desde la posición $i+1$ hasta la posición n y por cada intercambio debemos hacer permutaciones (llamada recursiva) desde la posición $i+1$ hasta la posición n .

El algoritmo que ejecuta dicho proceso es el siguiente:

```

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)

```

Sólo haremos una parte del seguimiento, la que consideramos suficiente para entender cómo funciona dicho algoritmo. Consideremos el siguiente vector V , con $n=3$:

1	2	3
a	b	c

El programa llamante del subprograma permuta es:

```

----
----
permuta(V, 3, 1)
L0 → ----
----

```

Las direcciones de retorno las rotularemos L0 y L1 respectivamente, las cuales son las instrucciones siguientes a las instrucciones que llaman a permuta, ya que se trata de un subprograma. La pila para manejar los datos correspondientes a llamadas interrumpidas contendrá la información perteneciente a los parámetros (**V**, **n**, **i**), la dirección de retorno y la variable local **k**.

Cuando se ejecuta la llamada a permuta en el programa llamante el estado de la pila es el siguiente:

L0	a	b	c	3	1
DR	V			n	i
					k

que significa que se va a ejecutar el subprograma permuta con los datos correspondientes de **V**, **n** e **i** y que cuando termine regresará a la instrucción llamada L0 a continuar ejecutando el programa llamante.

La condición de la instrucción 2 es falsa, por consiguiente continúa ejecutando a partir de la instrucción 5. Esta instrucción es el comienzo de un ciclo for, el cual hará que la variable **k** tenga 3 valores: 1, 2 y 3, y comenzará ejecutando con el valor de **k=1**. Esta situación la representaremos en la pila de la siguiente forma:

L0	a	b	c	3	1
DR	V			n	i
					k

representando que vamos a ejecutar el ciclo for con la variable **k = 1**.

Las instrucciones del ciclo for son: intercambiar el contenido de **V[i]** con **V[k]**, lo cual deja el vector sin modificación debido a que **i** y **k** tienen el mismo valor; la instrucción siguiente es llamada recursiva al subprograma permuta. El estado de la pila será el siguiente:

L1	a	b	c	3	2
L0	a	b	c	3	1
DR	V			n	i
					k

indicando que vamos a ejecutar dicho subprograma con los valores de **V**, **n** e **i** que hay en el tope y con dirección de retorno a la instrucción L1 para continuar ejecutando el ciclo for de la llamada interrumpida. La nueva ejecución ocasiona que la condición de la instrucción 2 sea falsa, por tanto, la ejecución continúa a partir de la instrucción 5, la cual inicia el ciclo for con **k** variando desde 2 hasta 3 y su valor inicial es 2. El estado de la pila es:

L1	a	b	c	3	2
L0	a	b	c	3	1
DR	V			n	i
					k

indicando que comenzaremos la ejecución del ciclo for con **k=2**.

La primera instrucción del ciclo intercambia **V[i]** con **V[k]**, dejando el vector inalterado (**k** e **i** son iguales) y encuentra una nueva llamada recursiva al subprograma permuta, que en la pila representamos así:

L1	a	b	c	3	3
L1	a	b	c	3	2 2 3
L0	a	b	c	3	1 1 2 3
DR	V			n	i k

indicando que ejecutaremos permuta con los valores de **V**, **n** e **i** que hay en el tope, con dirección de retorno a la instrucción rotulada L1 para continuar la ejecución del ciclo que se interrumpió con la variable **k** valiendo 2.

La ejecución de permuta con estos nuevos valores de los parámetros ocasiona que la condición de la instrucción 2 sea verdadera, ejecutando por consiguiente la instrucción 3, la cual imprime el vector **V**, produciendo la siguiente impresión: **abc**.

Continúa con la instrucción siguiente al end(if), la cual es el fin del subprograma, indicando que ha terminado la ejecución del subprograma permuta, por tanto desapila los datos que hay en el tope de la pila y retorna a la instrucción rotulada L1 a continuar el ciclo for de la llamada que había sido interrumpida. El estado de la pila es el siguiente:

L1	a	b	c	3	2 2 3
L0	a	b	c	3	1 1 2 3
DR	V			n	i k

La instrucción L1 es el fin del ciclo for, lo que significa que ha terminado de ejecutar el ciclo para **k=2** y que debe repetir de nuevo el ciclo con **k=3**. Representaremos esta situación en la pila así:

L1	a	b	c	3	2 2 3
L0	a	b	c	3	1 1 2 3
DR	V			n	i k

indicando que hemos ejecutado el ciclo for con **k=2** y que continuamos la ejecución del ciclo con **k=3**. Las instrucciones del ciclo son: intercambiar **V[i]** con **V[k]**, por tanto intercambia **V(2)** con **V(3)** y el estado de la pila es el siguiente:

L1	a	c	b	3	2 2 3
L0	a	b	c	3	1 1 2 3
DR	V			n	i k

Luego llama nuevamente a permuta y la pila queda así:

L1	a	c	b	3	3
L1	a	c	b	3	2 2 3
L0	a	b	c	3	1 1 2 3
DR	V			n	i K

indicando que ejecutará el subprograma con los datos que hay en el tope y que retornará a L1 para continuar la ejecución del ciclo for con **k=3**.

Esta nueva ejecución ocasiona que la condición de la instrucción 2 sea verdadera, por tanto imprime el contenido del vector **V**, produciendo la siguiente impresión: **acb**.

Continúa con la instrucción siguiente al end(if), la cual es el fin del subprograma, por tanto desapila los datos que hay en el tope y regresa a la instrucción llamada L1 para continuar la ejecución del ciclo for con **k=3**. El estado de la pila es el siguiente:

L1	a	c	b	3	2 2 3
L0	a	b	c	3	1 1 2 3
DR	V			n	i k

La instrucción L1 es el fin del for, lo que significa que ha terminado de ejecutar el ciclo for para **k=3**, pero como éste es el último valor que había de tomar la **k**, el ciclo for ha terminado para esta llamada y continúa con el end(if) y luego el fin del subprograma y ha terminado de ejecutar esta llamada. Entonces desapila y regresa a L1 a continuar ejecutando el ciclo de la llamada que había interrumpido. El estado de la pila ahora es:

L0	a	b	c	3	1 1 2 3
DR	V			n	i K

La instrucción L1 es el fin del for, o sea que ha terminado para **k=1** y debe repetir las instrucciones del ciclo con **k=2**. Representamos esto en la pila así:

L0	a	b	c	3	1 4 2 3
DR	V			n	i k

indicando que ejecutaremos de nuevo las instrucciones del ciclo for con **k=2**.

Las instrucciones del ciclo for son: intercambiar el contenido de **V[i]** con **V[k]** (intercambia V(1) con V(2)) y llamada recursiva a permuta con el nuevo estado del vector **V** y **n=3** e **i=2**. el estado de la pila es:

L1	b	a	c	3	2
L0	A	b	c	3	1 4 2 3
DR	V			n	i k

A partir de este punto se presenta una situación análoga a la que hemos visto anteriormente, en la cual se inicia de nuevo el ciclo for con valores para k de 2 y 3, produciendo las impresiones **bac** y **bca** respectivamente.

Al retornar a L1, habrá terminado el ciclo para **k=2** y lo repetirá de nuevo para **k=3**, lo cual producirá las impresiones **cab** y **cba**. Dejamos al estudiante que compruebe dichos resultados continuando con el seguimiento.

EJERCICIOS PROPUESTOS

1. Escriba una función recursiva que evalúe el determinante asociado a una matriz cuadrada de orden n utilizando el método de los menores.
2. Escriba un algoritmo recursivo que imprima las combinaciones de n elementos tomados de r .
3. Escriba un algoritmo recursivo que imprima las 2^n posibles combinaciones de n variables lógicas.
4. El conjunto potencia, de un conjunto dado, se define como todos los subconjuntos que se pueden conformar a partir de los elementos de un conjunto dado. Escriba un algoritmo recursivo que imprima el conjunto potencia de un conjunto dado.
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.
6. Elabore algoritmo recursivo que tome una pila representada como lista ligada y construya la pila representada en vector.

MÓDULO 16

RECURSION (QUICKSORT, SORTMERGE, CONVERSION A NO RECURSIVO)

Introducción

Ya hemos introducido todo lo que es la programación recursiva en los dos módulos anteriores, y en el módulo 2 tratamos algunos algoritmos de ordenamiento, los cuales tienen orden de magnitud cuadrático. Veremos en este módulo dos nuevos métodos de ordenamiento usando la técnica de recursión. Adicionalmente veremos cómo convertir algoritmos recursivos a no recursivos.

Objetivos

1. Ordenar los datos de un vector usando técnicas recursivas.
2. Convertir algoritmos recursivos tipo void a versiones no recursivas.
3. Convertir algoritmos recursivos que retornan un dato a versión no recursiva.

Preguntas básicas

1. En qué consiste la técnica de ordenamiento denominada quickSort?
- 2.Cuál es el orden de magnitud promedio de la técnica quickSort?
3. Cuándo se presenta el peor caso para la técnica de ordenamiento quickSort?
4. En qué consiste la técnica de ordenamiento sortMerge?
- 5.Cuál es el orden de magnitud promedio de la técnica sortMerge?
6. Qué diferencia hay entre convertir un algoritmo recursivo tipo void a no recursivo frente a convertir un algoritmo recursivo que no es tipo void a no recursivo?

16.1 Método de ordenamiento quickSort: Este es el método de ordenamiento que estadísticamente es el más eficiente, en el caso promedio, para ordenar los datos de un vector. El método, básicamente consiste en elegir un dato del vector y ubicarlo en el sitio que le corresponde, es decir, los anteriores serán todos menores que ese dato y los siguientes serán todos mayores que ese dato. Luego de efectuar esta operación se procede a ordenar los datos que quedaron antes y los que quedaron después usando la misma técnica. El algoritmo es el siguiente:

```
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)
```

Nuestro algoritmo es tipo void, ya que no retorna ningún dato sino que simplemente ejecuta una tarea sobre el vector que invoca el método. Los parámetros son dos datos enteros que indican desde dónde hasta dónde se debe efectuar el proceso de ordenamiento: **m** es la posición inicial y **n** es la posición final.

En el ciclo de las instrucciones 6 a 16, conjuntamente con la instrucción 17 se coloca el dato elegido en la posición que le corresponde. En nuestro algoritmo, por comodidad, el dato elegido para colocarlo en la posición correcta es el primero del vector a ordenar, es decir, el dato de la posición **m**.

Para ver cómo funciona el proceso de colocar un dato en la posición correcta consideremos el siguiente vector:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
V	16	13	12	4	8	65	29	1	43	55	72	28	9	10	64	19	35

Al invocar el método por primer vez el valor de la **m** es 1 y el valor de la **n** es 16. En nuestro algoritmo si la **m** es menor que la **n** (instrucción 3) se le asigna a la **i** el valor de la **m** y a la **j** el valor de **n+1**. Por consiguiente la **i** queda valiendo 1 y la **j** 17.

El ciclo que se inicia en la instrucción 6 se ejecuta mientras la **i** sea menor que la **j**. Inicial mente esta condición es verdadera, por tanto se procede a ejecutar el ciclo de las instrucciones 7 a 9. En dicho se comienza incrementando la **i** en 1 y se controla (instrucción 9) que la **i** sea menor o igual que la **n** y que el dato de la posición **i** sea menor o igual que el dato de la posición **m**. Para los valores de **i** 2, 3 y 4 ambas condiciones son verdaderas. Cuando la **i** vale 5 el dato de la posición 5 es mayor que el dato de la posición **m** ($m = 1$), por tanto termina la ejecución de esta ciclo y la **i** queda valiendo 5. Terminado este ciclo continúa con la ejecución del ciclo de las instrucciones 10 a 12. En este ciclo se decrementa la **j** mientras se cumpla que la **j** sea diferente de la **m** y el dato de la posición **j** sea mayor o igual que el dato de la posición **m**. Para los valores de **j** 16, 15, 14 la condición es verdadera, cuando la **j** toma el valor de 13 el dato de la posición 13 es menor que el dato de la posición **m** ($m = 1$), por tanto, se sale del ciclo y la **j** queda valiendo 13. En este momento la situación es la siguiente:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
V	16	13	12	4	8	65	29	1	43	55	72	28	9	10	64	19	35

Diagram showing pointers: **m** points to index 1, **i** points to index 5, **j** points to index 13, and **n** points to index 16.

Terminado el ciclo de las instrucciones 10 a 12 se controla que la **i** sea menor que la **j** con la instrucción 13. De ser cierta esta condición se intercambia el dato de la posición **i** con el dato de la posición **j** con la instrucción 14 y el vector queda así:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
V	16	13	12	4	8	10	29	1	43	55	72	28	9	65	64	19	35

Diagram showing pointers: **m** points to index 1, **i** points to index 5, **j** points to index 13, and **n** points to index 16.

Se llega al fin del ciclo de las instrucciones 6 a 16, por consiguiente regresa a evaluar si continúa con el ciclo o no. En nuestro caso la **i** es menor que la **j**, por tanto, ejecuta nuevamente el ciclo de las instrucciones 7 a 9 y el ciclo de las instrucciones 10 a 12. Al terminar de ejecutar estos dos ciclos las variables **i** y **j** quedan valiendo 6 y 12 respectivamente. La situación es la siguiente:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
V	16	13	12	4	8	10	29	1	43	55	72	28	9	65	64	19	35

Diagram showing pointers: **m** points to index 1, **i** points to index 6, **j** points to index 12, and **n** points to index 16.

La condición de la instrucción 13 es nuevamente verdadera por consiguiente intercambia el dato de la posición 6 con el dato de la posición 12. El vector queda:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
V	16	13	12	4	8	10	9	1	43	55	72	28	29	65	64	19	35

Diagram showing pointers: **m** points to index 1, **i** points to index 6, **j** points to index 12, and **n** points to index 16.

Nuevamente la condición del ciclo de la instrucción 6 verdadera por tanto, ejecutará los ciclos internos (instrucciones 7 a 9 y 10 a 12) y los valores de *i* y *j* son 8 y 7 respectivamente. La situación es la siguiente:

	m						j		i								n
V	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	16	13	12	4	8	10	9	1	43	55	72	28	29	65	64	19	35

La condición de la instrucción 13 es falsa, por tanto, no efectúa ningún intercambio. La condición del ciclo de la instrucción 6 también es falsa, por consiguiente se sale del ciclo y ejecuta el intercambio de la instrucción 17, es decir, intercambia el dato de la posición *j* con el dato de la posición *m*. El vector queda:

	m						j		i								n
V	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	1	13	12	4	8	10	9	16	43	55	72	28	29	65	64	19	35

Como podrá observar el dato de la posición *j* está en el sitio que le corresponde: los datos anteriores a él son todos menores y los datos a continuación son todos mayores.

Lo que sigue será ordenar los datos desde la posición *m* hasta la posición *j* - 1 y los datos desde la posición *j* + 1 hasta la posición *n*. El ordenamiento de estos dos subvectores se efectúa de la misma forma que este proceso que hemos realizado, por consiguiente se invoca nuevamente el método quickSort enviándole los parámetros apropiados (instrucciones 18 y 19).

Veamos ahora cómo es la eficiencia de este algoritmo. El proceso de colocar un dato en la posición que le corresponde tiene orden de magnitud $O(n)$, siendo *n* la diferencia entre la posición inicial y la posición final a ordenar. Si consideramos que el dato elegido para ubicarlo en la posición correcta queda ubicado aproximadamente en la mitad del vector significa que cada llamada recursiva será para ordenar la mita de los datos del vector que se está ordenando, por consiguiente la ecuación de recurrencia para evaluar el orden de magnitud de este algoritmo es de la forma $T(n) = 2T(n/2) + n$, la cual resolviéndola según el teorema maestro se obtiene que el orden de magnitud es $O(n \log_2 n)$. La situación más desfavorable se presenta cuando los datos del vector están ordenados y se aplica este método de ordenamiento, ya que la división del vector en subvectores queda desbalanceada: un subvector de un elemento y el otro con *n* - 1 elementos, ocasionando que la eficiencia del algoritmo sea cuadrática, es decir $O(n^2)$. Igual cosa sucedería si todos los datos del vector fueran iguales. Al respecto se han hecho una gran cantidad de estudios estadísticos y se ha encontrado que estas dos situaciones planteadas casi nunca se presentan, por tanto, por lo general la ejecución de este método de ordenamiento tiene orden de magnitud semilogarítmico ($O(n \log_2 n)$).

16.2 Método de ordenamiento sortMerge: Este método de ordenamiento no busca ubicar inicialmente un dato en la posición que le corresponde. Simplemente divide el vector en dos subvectores de igual tamaño, los ordena por separado y luego los intercala. Este método tiene la “inconveniencia” de que requiere memoria adicional para ejecutarse.

1. **void** sortMerge(entero primero, entero ultimo)
2. entero mitad
3. **if** (primero < ultimo) **then**
4. mitad = (primero + ultimo) / 2
5. sortMerge(primero, mitad)
6. sortMerge(mitad+1, ultimo)
7. intercalar(primero, mitad, ultimo)
8. **end(if)**
9. **end(sortMerge)**

Nuestro método lo denominamos `sortMerge`, es de tipo `void` ya que simplemente ejecuta una tarea sobre el objeto que invoca el método y tiene dos parámetros de tipo entero que representan la posición inicial y la posición final del vector a ordenar.

Definimos un variable local que llamamos `mitad`. Si `primero` es menor que `ultimo` hallamos la posición de la `mitad` y procedemos a ordenar cada una de esas mitades con las llamadas recursivas de las instrucciones 5 y 6. Al terminar de ordenar cada una de esas mitades procedemos a intercalarlas obteniendo de esta forma los datos del vector ordenados en su totalidad.

```
1. void intercalar(entero primero, entero mitad, entero ultimo)
2.     entero i, j, k, b[]
3.     b = new entero[V[0]]
4.     if (primero > mitad) or (mitad + 1 > ultimo) then return
5.     for (k = primero; k <= ultimo; k++) do
6.         b.asigneDato(V[k], k)
7.     end(for)
8.     i = primero
9.     j = mitad + 1
10.    k = primero - 1
11.    while ((i <= mitad) and (j <= ultimo)) do
12.        k = k + 1
13.        if (b.datoEnPosicion(i) <= b.datoEnPosicion(j)) then
14.            V[k] = b.datoEnPosicion(i)
15.            i = i + 1
16.        else
17.            V[k] = b.datoEnPosicion(j)
18.            j = j + 1
19.        end(if)
20.    end(while)
21.    while (i <= mitad) do
22.        k = k + 1
23.        V[k] = b.datoEnPosicion(i)
24.        i = i + 1
25.    end(while)
26.    while (j <= ultimo) do
27.        k = k + 1
28.        V[k] = b.datoEnPosicion(j)
29.        j = j + 1
30.    end(while)
31. fin(intercala)
```

Nuestro método para intercalar lo primero que hace es una copia del vector que invocó el método de los datos que hay desde la posición `primero` hasta la posición `ultimo` (instrucciones 5 a 7). Luego utilizando las variables `i`, `j` y `k` procedemos a intercalar los datos que se hallan en las posiciones desde **primero** hasta **mitad** del vector en el objeto **b** con los datos que se hallan en las posiciones desde **mitad + 1** hasta **ultimo** en el mismo objeto **b**, llevando dicha intercalación al vector **V** del objeto que invocó el método. Este algoritmo de intercalación es un algoritmo clásico de intercalación de dos vectores cuyo análisis corresponde al curso de lógica I.

Veamos ahora cómo es la eficiencia de este algoritmo. El proceso de intercalar dos vectores tiene orden de magnitud $O(n)$, siendo n el total de elementos de los dos vectores. Por consiguiente la ecuación de recurrencia para evaluar el orden de magnitud de este algoritmo es de la forma $T(n) = 2T(n/2) + n$, la cual resolviéndola según el teorema maestro, se obtiene que el orden de magnitud es $O(n \log_2 n)$. Cualquiera que sea la forma como lleguen los datos a este método de ordenamiento el orden de magnitud es semilogarítmico ($O(n \log_2 n)$).

16.3 Conversión de algoritmo recursivo tipo void en algoritmo no recursivo

Realmente, si se tiene clara la forma de hacer seguimiento a un algoritmo recursivo, la conversión a algoritmo no recursivo es simplemente escribir las instrucciones apropiadas para que el computador ejecute lo mismo que hacemos al hacer el seguimiento. Los pasos a seguir son:

1. Definir una pila vacía.
2. Asignar label (rótulo) a la primera instrucción ejecutable del subprograma.
3. Definir dirección de retorno, dependiendo si es una función o un procedimiento.
4. Asignar label (rótulo) a las instrucciones definidas como direcciones de retorno en el paso 3
5. Remover llamadas recursivas. Por cada llamada recursiva:
 - 5.1 Guardar en la pila:
 - Dirección de retorno.
 - Parámetros por valor del subprograma.
 - Variables locales del subprograma.
 - 5.2 Asignar nuevos valores a los parámetros del subprograma.
 - 5.3 Transferir el control a la instrucción cuyo label se definió en el paso 2.
6. Antes de cualquier return o end del subprograma codificar instrucciones que hagan lo siguiente:
 - Controlar que la pila no esté vacía (if tope > 0)
 - Si la pila no está vacía, desapilar los datos correspondientes a una llamada recursiva: parámetros, variables locales y dirección de retorno, y transferir el control a la instrucción rotulada con la dirección de retorno extraída de la pila.

El procedimiento descrito anteriormente es completamente válido para subprogramas tipo void. Si se trata de un subprograma que no es tipo void hay que hacer unas pequeñas modificaciones, las cuales trataremos cuando hagamos un ejemplo con una de ellas.

Ilustraremos los pasos definidos, con el algoritmo moverTorre, el cual es tipo void.

Su versión recursiva es:

```
1 void moverTorre(entero n, entero o, entero a, entero d)
2   if (n > 0) then
3     moverTorre(n - 1, o, d, a)
4     write(o, "→", d)
5     moverTorre(n - 1, a, o, d)
6   end(if)
7 fin(moverTorre)
```

la versión no recursiva es:

```
1 void moverTorre(entero n, entero o, entero a, entero d)
2   entero pila = new pila[100]
3   tope = 0
4 L0: if (n > 0) then
5     pila[tope + 1] = n
6     pila[tope + 2] = o
7     pila[tope + 3] = a
8     pila[tope + 4] = d
9     pila[tope + 5] = 'L1'
10    tope = tope + 5
11    n = n - 1
12    o = o
13    a = d
14    d = pila[tope - 2]
15    goto L0
16 L1: write(o, "→", d)
17    pila[tope + 1] = n
18    pila[tope + 2] = o
19    pila[tope + 3] = a
```

```

20     pila[tope + 4] = d
21     pila[tope + 5] = 'L2'
22     tope = tope + 5
23     n = n - 1
24     o = a
25     a = pila[tope - 3]
26     d = d
27     goto L0
28 L2:end(if)
29     if (tope > 0) then
30         tope = tope - 5
31         n = pila[tope + 1]
32         o = pila[tope + 2]
33         a = pila[tope + 3]
34         d = pila[tope + 4]
35         dr = pila[tope + 5]
36         goto dr
37     end(if)
38 fin(moverTorre)

```

Las líneas 2 y 3 son la definición de la pila vacía.

En la línea 4 se le asigna rótulo (label) L0 a la primera instrucción ejecutable del subprograma.

Las direcciones de retorno serán L1 y L2, con las cuales rotularemos las instrucciones siguientes a las llamadas recursivas, las cuales son las instrucciones write (16) y end (28) respectivamente.

Las líneas 5 a 15 son las que reemplazan la primera llamada recursiva.

En líneas 5 a 8 se apilan los parámetros del procedimiento.

En la línea 9 se apila la dirección de retorno L1, que es con la cual se rotula la instrucción write, la cual es la instrucción siguiente a la primera llamada recursiva.

Líneas 11 a 14 asignan nuevos valores a los parámetros, y línea 15 transfiere el control al principio del procedimiento.

Línea 16 es simplemente la instrucción donde continúa el proceso cuando termina de resolver la primera llamada recursiva. Es la instrucción a la cual se le asigna rótulo (dirección de retorno).

Líneas 17 a 27 reemplazan la segunda llamada recursiva: 17 a 22 apilan los datos correspondientes a la llamada que se está interrumpiendo; 23 a 26 asignan los nuevos valores a los parámetros; línea 27 retorna el control al principio del procedimiento.

Línea 28 es la instrucción que es dirección de retorno correspondiente a la segunda llamada recursiva.

Líneas 29 a 37 son las instrucciones para desapilar los datos correspondientes a una llamada recursiva que fue interrumpida. El control se transfiere a L1 o a L2 dependiendo del dato que se haya sacado de la pila.

Hasta aquí es el proceso, digamos mecánico, que se sigue para convertir un procedimiento recursivo en uno no recursivo. En esta primera versión, el algoritmo obtenido tiene instrucciones redundantes y una o varias instrucciones goto. El paso siguiente es hacer un análisis lógico del algoritmo para eliminar las instrucciones redundantes y los goto.

En nuestro algoritmo, las instrucciones 12 y 26 son, a todas luces, instrucciones que sobran. Veamos ahora, otras instrucciones que no es tan obvio que sobren, pero que sobran. Cuando pregunta si tope es mayor que cero y resulta verdadera la condición, desapila los datos

correspondientes a una llamada recursiva con su correspondiente dirección de retorno y regresa a L1 o a L2 dependiendo de lo que hubiera desapilado. Cuando retorna a L2, comienza a ejecutar a partir del end(if) de la instrucción 28, la instrucción siguiente es preguntar de nuevo si hay datos correspondientes a una llamada interrumpida para volver a desapilar, por consiguiente los datos correspondientes a una llamada que se interrumpe, con dirección de retorno L2 no tienen utilización alguna y guardarlos no tiene sentido. De lo anterior, deducimos que las instrucciones para apilar estos datos sobran, es decir, las instrucciones 17 a 22. Ahora bien, si no vamos a guardar estos datos, la única dirección de retorno que quedará en la pila será L1, o sea que no habrá necesidad de almacenarla tampoco en la pila y la instrucción 9 también sobraría y la instrucción 36 se reemplaza por una instrucción que sea goto L1. Con estas modificaciones el algoritmo quedará así:

```

1 void moverTorre(entero n, entero o, entero a, entero d)
2     objeto pila = new pila[100]
3     tope = 0
4 L0: if (n > 0) then
5         pila[tope + 1] = n
6         pila[tope + 2] = o
7         pila[tope + 3] = a
8         pila[tope + 4] = d
9         tope = tope + 4
10        n = n - 1
11        a = d
12        d = pila[tope - 1]
13        goto L0
14 L1: write(o, "→", d)
15        n = n - 1
16        o = a
17        a = pila[tope + 2]
18        goto L0
19    end(if)
20    if (tope > 0) then
21        tope = tope - 4
22        n = pila[tope + 1]
23        o = pila[tope + 2]
24        a = pila[tope + 3]
25        d = pila[tope + 4]
26        goto L1
27    end(if)
28 fin(moverTorre)

```

El paso siguiente es eliminar los goto:

Cuando la condición de la instrucción 4 es verdadera ejecuta una serie de instrucciones y regresa siempre a chequear de nuevo la condición mediante la instrucción 13. Este grupo de instrucciones es simplemente un ciclo while: Mientras **n** sea mayor que cero.

La única forma que ejecute a partir de la instrucción 14 es que tope hubiera sido mayor que cero, ya que es allí donde se transfiere el control a la instrucción 14, por consiguiente, desde la instrucción 14 hasta la instrucción 18 pertenecen a la condición de la instrucción 20 y pueden ser trasladadas allí a reemplazar la instrucción 26. Nuestro algoritmo queda así:

```

1 void moverTorre(entero n, entero o, entero a, entero d)
2     objeto pila = new pila[100]
3     tope = 0
4 L0: while (n > 0) do
5         pila[tope + 1] = n
6         pila[tope + 2] = o
7         pila[tope + 3] = a
8         pila[tope + 4] = d

```

```

9      tope = tope + 4
10     n = n - 1
11     a = d
12     d = pila[tope - 1]
13 end(while)
14 if (tope > 0) then
15     tope = tope - 4
16     n = pila[tope + 1]
17     o = pila[tope + 2]
18     a = pila[tope + 3]
19     d = pila[tope + 4]
20     write(o, "→", d)
21     n = n - 1
22     o = a
23     a = pila[tope + 2]
24     goto L0
25 end(if)
26 fin(moverTorre)

```

Falta por eliminar el goto de la instrucción 24. La única forma de que esta instrucción se ejecute es que tope sea mayor que cero. La manera de eliminarlo es utilizando un ciclo, controlado por la condición de que tope sea mayor que cero. Usamos la instrucción **do** debido a que con la instrucción while no entraría al ciclo ya que el valor inicial de tope es cero. Nuevamente, de la instrucción 16 a la 23 hay instrucciones que sobran: las instrucciones 16 y 21 son una doble asignación a la variable **n**, las instrucciones 17 y 22 son doble asignación a la variable **o** y las 18 y 23 a la variable **a**. Reagrupando estas instrucciones y eliminando el goto nuestro algoritmo queda así:

```

1 void moverTorre(entero n, entero o, entero a, entero d)
2   objeto pila = new pila[100]
3   tope = 0
4   do
5     while (n > 0) do
6       pila[tope + 1] = n
7       pila[tope + 2] = o
8       pila[tope + 3] = a
9       pila[tope + 4] = d
10      tope = tope + 4
11      n = n - 1
12      a = d
13      d = pila[tope - 1]
14    end(while)
15    if (tope > 0) then
16      tope = tope - 4
17      n = pila[tope + 1] - 1
18      o = pila[tope + 3]
19      a = pila[tope + 2]
20      d = pila[tope + 4]
21      write(a, "→", d)
22    end(if)
23    while (tope > 0 or n > 0)
24 end(moverTorre)

```

16.4 Conversión de un subprograma recursivo que no es tipo void en NO recursivo

La conversión de una función recursiva a no recursiva difiere de la conversión de un procedimiento en algunos pequeños detalles: el primero es que el valor de la función se llevará siempre en el tope de la pila; el segundo es que la dirección de retorno es la misma instrucción que contiene la llamada recursiva, y el tercero es que la llamada recursiva se reemplaza siempre por pila(tope) y la instrucción siguiente a la que contiene la llamada recursiva será siempre

decrementar tope en 1. Ilustraremos el mecanismo convirtiendo la función factorial vista anteriormente, a su versión no recursiva. El algoritmo recursivo es:

```
1.  entero factorial(entero n)
2.      entero x
3.      if (n == 0) then
4.          x = 1
5.      else
6.          x = n * factorial(n - 1)
7.      end(if)
8.      return x
9.  fin(factorial)
```

la versión no recursiva será:

```
1.  entero factorial(entero n)
2.      entero x
3.      objeto pila = new pila[100]
4.      tope = 0
5. L0:  if (n == 0) then
6.          x = 1
7.      else
8.          pila[tope+1] = n
9.          pila[tope+2] = 'L1'
10.         tope = tope + 2
11.         n = n - 1
12.         goto L0
13. L1:  x = n * pila[tope]
14.         tope = tope - 1
15.     end(if)
16.     if (tope > 0) then
17.         tope = tope - 1
18.         n = pila[tope]
19.         dr = pila[tope+1]
20.         pila[tope] = x
21.         goto dr
22.     end(if)
23.     return x
24.  fin(factorial)
```

La eliminación de instrucciones redundantes es sencilla. Como sólo hay una dirección de retorno no es necesario guardar en la pila dicha dirección, por tanto las instrucciones 9 y 19 sobran y la instrucción 21 se reemplaza por la instrucción goto L1.

Las instrucciones 13 y 14 sólo se ejecutan cuando tope = 0, por tanto se pueden trasladar antes de la instrucción 21 y se traslada el rótulo L1 hacia la instrucción if tope > 0. Nuestro algoritmo quedará así:

```
1  entero factorial(entero n)
2  entero x
3  objeto pila = new pila[100]
4  tope = 0
5 L0:  if (n == 0) then
6      x = 1
7  else
8      tope = tope + 1
9      pila[tope] = n
10     n = n - 1
11     goto L0
12  end(if)
13 L1:  if (tope > 0) then
14     n = pila[tope]
```

```

15         pila(tope) = x
16         x = n * pila[tope]
17         tope = tope - 1
18         goto L1
19     end(if)
20     return x
21 fin(factorial)

```

Para eliminar el goto L0 vemos que las instrucciones 8 a 11 sólo se ejecutan cuando **n** es mayor que cero y se regresa a chequear de nuevo el valor de **n**, lo que configura un ciclo mientras. Las instrucciones 13 a 18 también configuran un ciclo mientras. Nuestro algoritmo queda así:

```

1  entero factorial(entero n)
2      objeto pila = new pila[100]
3      tope = 0
4      while (n > 0) do
5          tope = tope + 1
6          pila[tope] = n
7          n = n - 1
8      end(while)
9      x = 1
10     while (tope > 0) do
11         n = pila[tope]
12         pila[tope] = x
13         x = n * pila[tope]
14         tope = tope - 1
15     end(while)
16     return x
17 fin(factorial)

```

Y este algoritmo muestra el típico funcionamiento de la recursión: guardar en una pila los datos correspondientes a llamadas interrumpidas y luego resolver para dichos datos.

EJERCICIOS PROPUESTOS

1. 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

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

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

3. Convierta a no recursivo el algoritmo de permutaciones visto en el módulo 15.

4. Convierta a no recursivo el algoritmo para la función de Ackerman escrito en las soluciones del módulo 14 de este texto.