

# Introducción

Se han estudiado los números y los símbolos y se han construido estructuras más complejas con estos elementos denominadas listas. En el presente capítulo se utilizarán estructuras aún más complejas, grupos de listas o bien *listas de listas*.

La principal razón para utilizar estas nuevas estructuras es que nos permitirán un mayor nivel de *abstracción*. De esta forma se podrán representar modelos de datos más complejos tales como los conjuntos, vectores, matrices y árboles.

Los conjuntos, vectores y matrices son estructuras muy usadas en matemática. Los árboles son una representación jerárquica muy utilizada en computación.

Se estudiarán las técnicas de representación y se desarrollarán algoritmos que permitan construir nuevas abstracciones. Además se construirán procesos para su creación, modificación y eliminación.

## Representación de los conjuntos

El concepto de conjunto es fundamental en las matemáticas. Un conjunto es una colección de objetos. Estos objetos se suelen llamar los elementos del conjunto.

Las listas se pueden utilizar para representar conjuntos matemáticos. De esta forma se utiliza la siguiente notación.

Para representar el conjunto vacío denominado en matemática por el símbolo {} se utilizará la lista nula denominada por () .

Para representar un conjunto cualquiera de la forma:

$\{e_1, e_2, \dots, e_N\}$

se utilizará una lista de la forma:

$(e_1 \ e_2 \ \dots \ e_N)$

Al igual que con los conjuntos matemáticos, los conjuntos en Scheme deben cumplir las siguientes condiciones.

- a) No deben tener elementos repetidos.
- b) No existe ninguna relación de orden entre sus elementos.
- c) Por lo tanto, dos conjuntos son iguales si tienen los mismos elementos, sin importar el orden en el que se encuentren.

Con la notación definida un conjunto de la forma:

$\{ \ }$  se escribirá como  $( )$ .

$\{a, b, c\}$  se escribirá  $(a \ b \ c)$ .

$\{1, 2, 3, 4, 5\}$  se escribirá como  $(1 \ 2 \ 3 \ 4 \ 5)$ .

Tal como en matemática:  $\{1, 2, 3, 4, 5\}$  es igual a  $\{2, 3, 5, 4, 1\}$

En Scheme se supondrá que:  $(1 \ 2 \ 3 \ 4 \ 5)$  es igual a  $(2 \ 3 \ 5 \ 4 \ 1)$

## Identificación de conjuntos

Se construirá una función denominada (*conjunto?* *lista*). Esta función recibe como argumento un conjunto y verifica que efectivamente sea un conjunto. Básicamente debe revisar que no existan elementos repetidos. Debe producir los siguientes resultados:

```
> (conjunto? '( ))
#t

> (conjunto? '(a b c))
#t

> (conjunto? '(a b c d e))
#t
```

```
> (conjunto? '(a b c c d e))
#f
```

Para construir esta función se pregunta primero si se trata de lista, en caso positivo se recorre el conjunto y se verifica que cada elemento se encuentre presente únicamente una vez en la representación. Este proceso continua hasta terminar con el último elemento. A continuación se presenta el código.

```
; Determina si se recibe o no conjunto
;; Precondiciones:
;; conjunto: lista lineal
;;
;; Identifica si una lista cumple
;; las condiciones para ser un conjunto
;; lista: una lista lineal de símbolos
;;
(define (conjunto lista)
  (cond( (null? lista)
         #t)
       ( (miembro? (car lista) (cdr lista))
         #f)
       ( else
         (conjunto (cdr lista)))))
```

## Igualdad de conjuntos

Se construirá una función denominada (*equal-set?* *conj1 conj2*). Esta función recibe como argumentos dos conjuntos, en caso que sean iguales devuelve en valor de #t, en caso contrario devuelve el valor de #f. A continuación se presentan algunos ejemplos.

```
> (equal-set? '() '())
#t

> (equal-set? '(1 2 3) '(1 2 3 4))
#f

> (equal-set? '(1 2 3 4) '(1 2 3 5))
#f
```

```
> (equal-set? '(1 2 3 4) '(1 2 3 4))
#f
```

```
> (equal-set? '(1 3 2 4) '(4 3 2 1))
#t
```

Para construir esta función se tomará cada elemento del primer conjunto y se verificará que se encuentre en el segundo, si es así se produce el valor de #t, en caso contrario se produce el valor de #f. A continuación se muestra el código.

```
;; Identifica si dos conjuntos son iguales
;; Precondiciones:
;; conj1, conj2: listas lineales
;;
(define (equal-set? conj1 conj2)
  (cond( (and (null? conj1) (null? conj2))
         #t)
        ( (not (equal? (largo conj1) (largo conj2)))
          #f)
        ( else
          (equal-set?-aux conj1 conj2)))))

(define (equal-set?-aux conj1 conj2)
  (cond( (null? conj1)
         #t)
        ( (not (miembro? (car conj1) conj2))
          #f)
        ( else
          (equal-set?-aux (cdr conj1) conj2))))
```

## Unión de conjuntos

Se construirá una función que se denominará (*unión conj1 conj2*). Esta función recibe dos conjuntos como argumentos y su resultado es la unión de esos dos conjuntos. Su comportamiento debe ser.

```
> (unión '() '())
()
```

```
> (unión '(a b c) '())
(a b c)
```

```
> (unión '(a b c) '(d e f))
(a b c d e f)
```

```
> (unión '(a b c) '(a d b e f))
(c a d b e f)
```

Para construir esta función se utilizará el siguiente algoritmo. Unir un conjunto vacío con cualquier otro conjunto debe producir el otro conjunto. Para unir dos conjuntos no vacíos se toma un elemento del primer conjunto, si este elemento está en el segundo conjunto se desecha, en caso contrario, si no está en el segundo conjunto se pone en la solución final. Este proceso continua hasta que no queden elementos en el primer conjunto. Por último se agregan todos los elementos del segundo conjunto.

Esta función se puede programar de otras formas. Puesto que el orden de un conjunto no es importante, cualquier resultado producido se debe valorar utilizando la función de igualdad de conjuntos. Una posible implementación de este algoritmo es la siguiente.

```
;; Unión de conjuntos
;; conj1, conj2: son conjuntos
;;
(define (unión conj1 conj2)
  (cond ((null? conj1)
         conj2)
        ((null? conj2)
         conj1)
        ((miembro? (car conj1) conj2)
         (unión (cdr conj1) conj2))
        (else
         (cons (car conj1)
               (unión (cdr conj1) conj2)))))
```

## Intersección de conjuntos

Se construirá una función que se denominará (*intersección conj1 conj2*). Esta función recibe dos listas como argumentos y su resultado es la intersección de esas dos listas. Su comportamiento debe ser.

```

> (intersección '( ) '(a b c))
( )
> (intersección '(a b c ) '(a c d))
(a c)
> (intersección '(a b c) '(a x b y c))
(a b c)

```

La intersección de un conjunto vacío con cualquier otro conjunto produce un conjunto vacío. Si los dos conjuntos son no vacíos se toma el primer elemento del conjunto1, si este elemento se encuentra en el conjunto2 se pone dentro del conjunto solución, en caso contrario se continua con el siguiente elemento. Este proceso se repite hasta que se acaben los elementos del conjunto1. En ese momento se tiene la solución deseada. El programa puede ser escrito de la siguiente forma.

```

;; Intersección de conjuntos
;; conj1, conj2: son conjuntos
;;
(define (intersección conj1 conj2)
  (cond( (null? conj1)
         '( ))
       ( (null? conj2)
         '( ))
       ( (miembro? (car conj1) conj2)
         (cons (car conj1)
               (intersección (cdr conj1) conj2)))
       ( else
         (intersección (cdr conj1) conj2)))))


```

## Representación de vectores y matrices

Un vector es un arreglo de números de la forma:

$$V = [ v[1] \ v[2] \ \dots \ v[n] ]$$

Que se representará como una lista:

$$V = ( v[1] \ v[2] \ \dots \ v[n] )$$

A los elementos  $v[i]$  se les denomina componentes. Si todos los componentes son iguales a 0 entonces el vector V se denomina vector nulo. Dos vectores V y W son iguales si y solo si poseen los mismos elementos y en el mismo orden.

Sea

$$V = ( v[1] \ v[2] \ \dots \ v[n] )$$

$$W = ( w[1] \ w[2] \ \dots \ w[n] )$$

$$V = W \text{ si y solo si}$$

$$v[1]=w[1], \ v[2]=w[2], \ \dots, \ v[n]=w[n]$$

Una matriz es un arreglo rectangular de la forma:

$$\begin{pmatrix} e[1,1] & e[1,2] & \dots & e[1,m] \\ e[1,2] & e[2,2] & \dots & e[2,m] \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ e[n,1] & e[n,2] & \dots & e[n,m] \end{pmatrix}$$

Los valores horizontales se llaman las filas de la matriz, o los vectores fila. Los valores verticales se denominan las columnas o los vectores columna. En este caso la matriz cuenta con n filas y m columnas, por lo que se dice que M es una matriz  $n \times m$ .

Si la matriz se representa por medio de las filas se puede representar mediante una lista de listas de la siguiente manera.

```
> `(( e[1,1]   e[1,2]   ...   e[1,m] )
  ( e[2,1]   e[2,2]   ...   e[2,m] )
  .
  .
  .
  .
  .
  ( e[n,1]   e[n,2]   ...   e[n,m] ))
```

Las tuplas verticales se llaman columnas o vectores columna. Si la matriz se representa por medio de ellos se obtendría el siguiente recorrido.

```
> `(( e[1,1]   e[2,1]   ...   e[n,1] )
```

```
  ( e[1,2]   e[2,2]   ...   e[n,2] )
```

```
      . . .
```

```
      . . .
```

```
      ( e[1,m]   e[2,m]   ...   e[n,m] ))
```

## Igualdad de vectores

Dos vectores son iguales si y solo sí tienen los mismos elementos y éstos se encuentran en el mismo orden.

Se puede utilizar la función `equal?` para determinar la igualdad de dos vectores. Sin embargo se construirá una función llamada (`equal-vec? vec1 vec2`), la cual produce el valor de `#t` si recibe dos vectores iguales y produce el valor de `#f` en cualquier otro caso. A continuación se presentan unos ejemplos.

```
> (equal-vec? '() '())
```

```
#t
```

```
> (equal-vec? '(1 2 3) '(1 2 3))
```

```
#t
```

```
> (equal-vec? '(1 2 3) '(3 2 1))
```

```
#f
```

```
> (equal-vec '(1 2 3) '(1 2 3 4))
```

```
#f
```

Para construir esta función se deben comparar los elementos uno a uno, si en algún momento dos componentes difieren se produce el valor de `#f`, en otro caso, si se llega al final de ambos vectores se produce el valor de `#t`. El código de esta función puede escribirse como:

```
;; Determienaa si dos vectores son iguales
;; vec1, vec2: listas lineales
```

```
;;
(define (equal-vec? vec1 vec2)
  (cond( (and (null? vec1) (null? vec2))
         #t)
       ( (or (null? vec1) (null? vec2))
         #f)
       ( (not (equal? (car vec1) (car vec2)))
         #f)
       ( else
         (equal-vec? (cdr vec1) (cdr vec2)))))
```

## Suma de vectores

La suma de vectores produce como resultado un vector. Se realiza componente con componente. La suma vectorial está definida de acuerdo con la siguiente fórmula:

$$V = ( v[1] \ v[2] \ \dots \ v[n] )$$

$$W = ( w[1] \ w[2] \ \dots \ w[n] )$$

$$V + W = ( v[1] + w[1] \ v[2] + w[2] \ \dots \ v[n] + w[n] )$$

De esta manera una función que sume vectores debe producir los siguientes resultados:

```
> (sumar-vec-vec '(0 0 0) '(1 4 5))
(1 4 5)
```

```
> (sumar-vec-vec '(1 1 1) '(4 6 8))
(5 7 9)
```

```
> (sumar-vec-vec '(2 3 1 1) '(10 20 30 40))
(12 23 31 41)
```

Construir esta función es relativamente sencillo. Se toma el primer elemento del vector1 y se suma con el primer elemento del vector2, el proceso continua hasta que ambos vectores queden vacíos, en ese momento se retorna la lista vacía.

```
;; Suma de dos vectores
;; vec1, vec2 : vectores de igual longitud
;; es decir (equal? (length vec1) (length vec2))
```

```

;;;
(define (sumar-vec-vec vec1 vec2)
  (cond ((and (null? vec1) (null? vec2))
         '())
        (else
         (cons (+ (car vec1) (car vec2))
               (sumar-vec-vec (cdr vec1) (cdr vec2)))))))

```

## Multiplicación de vectores

La multiplicación de vectores produce como resultado un único valor. Multiplica cada punto de un vector con otro y posteriormente suma todos los elementos. La multiplicación de vectores se puede representar mediante la fórmula:

$$V = (v[1] \ v[2] \ \dots \ v[n])$$

$$W = (w[1] \ w[2] \ \dots \ w[n])$$

$$V * W = v[1]*w[1] + v[2]*w[2] + \dots + v[n]*w[n]$$

Una función que multiplique vectores debe producir los siguientes resultados:

> (mul-vec-vec '(0 0 0) '(1 4 5))

0

> (mul-vec-vec '(1 1 1) '(4 6 8))

18

> (mul-vec-vec '(2 3 1 1) '(10 20 30 40))

150

Para programar este algoritmo se utiliza la definición presentada con lo cual se obtiene el siguiente código.

```

;; Multiplicación de dos vectores
;; vec1, vec2 : vectores de igual tamaño
;;
(define (mul-vec-vec vec1 vec2)
  (cond ((and (null? vec1) (null? vec2))
         '())
        (else
         (cons (+ (car vec1) (car vec2))
               (mul-vec-vec (cdr vec1) (cdr vec2))))))

```

```

0)
( else
  (+ (* (car vec1) (car vec2))
    (mul-vec-vec (cdr vec1) (cdr vec2))))))

```

## Transpuesta de una matriz

Obtener la transpuesta de una matriz consiste en convertir sus filas en columnas. Sea  $V$  una matriz, se mostrará el resultado de la (*transpuesta*  $V$ ).

$$V = \begin{pmatrix} (v[1,1] & v[1,2] & \dots & v[1,m]) \\ (v[2,1] & v[2,2] & \dots & v[2,m]) \end{pmatrix}$$

.

.

.

$$(v[n,1] & v[n,2] & \dots & v[n,m]))$$

$$(\text{transpuesta } V) = \begin{pmatrix} (v[1,1] & v[2,1] & \dots & v[n,1]) \\ (v[1,2] & v[2,2] & \dots & v[n,2]) \end{pmatrix}$$

.

.

.

$$(v[1,m] & v[2,m] & \dots & v[n,m]))$$

A continuación se construirá una función llamada (*transpuesta matriz*) que recibe como argumento una matriz representada por filas y produce como salida la transpuesta de dicha matriz. Esta función debe producir los siguientes resultados:

```

> (transpuesta '((1 2)
                  (1 2)))
((1 1)
 (2 2))

```

```
> (transpuesta '((1 2)
                  (3 4)) )
((1 3)
 (2 4))

> (transpuesta '((1 2 3)
                  (1 2 3)
                  (1 2 3)))
((1 1 1)
 (2 2 2)
 (3 3 3))

> (transpuesta '((1 2 3)
                  (4 5 6)
                  (7 8 9)))
((1 4 7)
 (2 5 8)
 (3 6 9))

> (transpuesta '((1 2 3 4)
                  (1 2 3 4)
                  (1 2 3 4)))
((1 1 1)
 (2 2 2)
 (3 3 3)
 (4 4 4))

> (transpuesta '((1 2 3 4)
                  (5 6 7 8)
                  (9 10 11 12)) )
((1 5 9)
 (2 6 10)
 (3 7 11)
 (4 8 12))
```

La respuesta anterior se ha estructurado para que pueda apreciarse tanto la matriz por la cual se pregunta como la matriz que se ha producido como resultado del proceso de evaluación.

Para construir la función transpuesta se utilizan dos funciones adicionales, sacar-1f que construye una lista con el primer elemento de cada fila y borrar-1f que borra el primer elemento de cada fila. A continuación se presenta el código de esta función.

Para el siguiente caso observe el comportamiento de las dos funciones auxiliares propuestas.

```
> (transpuesta '((1 2 3)
                  (4 5 6)
                  (7 8 9)))
```

```
((1 4 7)
 (2 5 8)
 (3 6 9))
```

```
> (sacar-1f '((1 2 3)
                  (4 5 6)
                  (7 8 9)))
```

```
(1 4 7)
```

```
> (borrar-1f '((1 2 3)
                  (4 5 6)
                  (7 8 9)))
```

```
((2 3)
 (5 6)
 (8 9))
```

De esta forma al utilizar esas funciones se puede construir el siguiente proceso:

```
> (transpuesta '((1 2 3)
                  (4 5 6)
                  (7 8 9)))
```

```
(cons '(1 4 7)
```

```
      (transpuesta '((2 3)
                      (5 6)
                      (8 9))))
```

```
(cons '(1 4 7)
```

```
      (cons '(2 5 8)
```

```

(transpuesta '((3)
               (6)
               (9)))))

(cons '(1 4 7)
      (cons '(2 5 8)
            (cons '(3 6 9)
                  (transpuesta '(( )
                                ( )
                                ( )))))))

(cons '(1 4 7)
      (cons '(2 5 8)
            (cons '(3 6 9)
                  '()))))

(cons '(1 4 7)
      (cons '(2 5 8)
            '((3 6 9)))))

(cons '(1 4 7)
      '((2 5 8)
        (3 6 9)))

((1 4 7)
 (2 5 8)
 (3 6 9)))

```

A continuación se presenta el código de estas funciones auxiliares:

```

;; Saca el 1er elemento de cada fila
;; de la matriz mat
;;
(define (sacar-1f mat)
  (cond ( (null? mat)
          '())
        ( else
          (cons (car (car mat))
                (sacar-1f (cdr mat))))))

;; Borra el 1er elemento de cada fila
;; de la matriz mat

```

```
;;
(define (borrar-1f mat)
  (cond ( (null? mat)
            '())
        ( else
          (cons (cdr (car mat))
                (borrar-1f (cdr mat))))))
```

De esta forma, la transpuesta se puede construir como la aplicación sucesiva de estas dos funciones:

```
;; Obtiene la transpuesta de una matriz
;; mat: matriz representada en filas en una lista
;;
(define (transpuesta mat )
  (cond ( (null? (car mat))
            '())
        ( else
          (cons (sacar-1f mat)
                (transpuesta (borrar-1f mat))))))
```

Observe que tanto saca-1f como borra-1f realizan el proceso de pasar por cada uno de los vectores de la matriz. Su única diferencia consiste en que el primero utiliza la función de car sobre cada una de las filas y el segundo la función de cdr. Así que se puede construir una función que reciba este función car o cdr como el argumento a aplicar. Por ejemplo:

```
> (aplicar-f car '((1 2 3)
                     (4 5 6)
                     (7 8 9)))
(1 4 7)
> (aplicar-f cdr '((1 2 3)
                     (4 5 6)
                     (7 8 9)))
((2 3)
 (5 6)
 (8 9))
```

De esta forma el código anterior se puede reescribir de la siguiente manera.

```

;; Aplica una función
;; a cada elemento de una lista
;; fun : una función de scheme
;; lista : una lista lineal
;;
(define (aplicar-f fun lista)
  (cond( (null? lista)
         '())
       ( else
         (cons (fun (car lista))
               (aplicar-f fun (cdr lista))))))

;; Obtiene la transpuesta de una matriz
;; mat: matriz representada en filas en una lista
;;
(define (transpuesta mat)
  (cond( (null? (car mat))
         '())
       ( else
         (cons (aplicar-f car mat)
               (transpuesta (aplicar-f cdr mat))))))

```

Existe en Scheme una función primitiva llamada `map` que realiza un proceso similar al que realiza la función `aplicar` descrita anteriormente. La función `map` realiza otras cosas no contempladas en la función `aplicar`. Sin embargo se puede utilizar para reescribir el programa anterior. De esta forma el código anterior se puede escribir de la siguiente manera:

```

;; Obtiene la transpuesta de una matriz
;; mat: matriz representada en filas en una lista
;;
(define (transpuesta mat)
  (cond( (null? (car mat))
         '())
       ( else
         (cons (map car mat)
               (transpuesta (map cdr mat))))))

```

## Multiplicación de un vector por una matriz

Sea  $V$  un vector de  $n$  elementos y sea  $W$  una matriz de  $n$  filas por  $m$  columnas. Es decir,  $M$  es una matriz  $n \times m$ .

La multiplicación de  $V * W$  se define como:

$$V = ( v[1] \quad v[2] \quad \dots \quad v[n] )$$

$$W = ( w[1,1] \quad w[1,2] \quad \dots \quad w[1,m] ) \\ ( w[2,1] \quad w[2,2] \quad \dots \quad w[2,m] )$$

$$\begin{array}{cccc} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ ( w[n,1] \quad w[n,2] \quad \dots \quad w[n,m] ) \end{array}$$

$$V * W = ( v[1]*w[1,1] + v[2]*w[2,1] + \dots + v[n]*w[n,1] \\ v[1]*w[1,2] + v[2]*w[2,2] + \dots + v[n]*w[n,2] )$$

$$\begin{array}{cccc} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ v[1]*w[1,m] + v[2]*w[2,m] + \dots + v[n]*w[n,m] \end{array}$$

Debido a la anterior definición se puede deducir que una representación por medio de columnas para una matriz simplifica el proceso de programación. Sin embargo, se supondrá que la matriz  $W$  se encuentra representada por medio de filas. Para convertir una representación de filas a una representación de columnas se debe obtener la transpuesta de la matriz.

Se construirá una función llamada `mul-vec-mat` que realizará la operación de multiplicar un vector por una matriz representada por medio de vectores fila. Esta función debe producir los siguientes resultados.

```
> (mul-vec-mat '(1 1 1) '((1 4 3)
                           (2 5 6)
                           (3 6 9)))
   (6 15 18)
```

```

> (mul-vec-mat '(2 2 2) '((1 4 3)
                           (2 5 6)
                           (3 6 9)))
(12 30 36)

> (mul-vec-mat '(1 2 3) '((1 4 3)
                           (2 5 6)
                           (3 6 9)))
(14 32 42)

> (mul-vec-mat '(1 2 3 4) '( (1 1 1)
                           (2 2 2)
                           (3 3 3)
                           (4 4 4)))
(30 30 30)

```

A continuación se implementa la fórmula descrita para la multiplicación de un vector por una matriz.

```

;; Multiplicación un vector por una matriz
;; Precondiciones:
;; vec : un vector de largo n
;; mat : una matriz de n filas y m columnas
;;
(define (mul-vec-mat vec mat)
  (mul-vec-mat-aux vec (transpuesta mat)))

(define (mul-vec-mat-aux vec mat)
  (cond ((null? Mat)
         '())
        (else
         (cons (mul-vec-vec vec (car mat))
               (mul-vec-mat-aux vec (cdr mat)))))))

```

## Multiplicación de matrices

Sea V un matriz de n filas y p columnas y sea W una matriz de p filas por m columnas. El producto de  $V \cdot W$  será una matriz de n filas y m columnas.

La multiplicación de  $V * W$  se define como:

$$V = ((v[1,1] \quad v[1,2] \quad \dots \quad v[1,p]) \\ (v[2,1] \quad v[2,2] \quad \dots \quad v[2,p]) \\ \vdots \quad \vdots \quad \ddots \quad \vdots \\ (v[n,1] \quad v[n,2] \quad \dots \quad v[n,p]))$$

$$W = ((w[1,1] \quad w[1,2] \quad \dots \quad w[1,m]) \\ (w[2,1] \quad w[2,2] \quad \dots \quad w[2,m]) \\ \vdots \quad \vdots \quad \ddots \quad \vdots \\ (w[p,1] \quad w[p,2] \quad \dots \quad w[p,m]))$$

El producto matricial de  $V * W$  se define matemáticamente como:

$$e(i, j) = \sum_{k=1}^p v(i, k) * w(k, j) \text{ con } i = 1, \dots, n \\ j = 1, \dots, m$$

Se construirá una función llamada mul-mat-mat la cual debe producir los siguientes resultados:

```
> (mul-mat-mat '((1 1)
                  (2 2))
      '((1 2)
        (3 4)))
((4 6)
 (8 12))
```

```
> (mul-mat-mat '((1 1 1)
                  (2 2 2)
                  (1 2 3)))
```

```

    '((1 4 3)
      (2 5 6)
      (3 6 9)))
((6 15 18)
 (12 30 36)
 (14 32 42))
> (mul-mat mat '((1 2)
                  (1 2)))
    '((1 2 3 4)
      (5 6 7 8)))
((11 14 17 20)
 (11 14 17 20))

```



El código de esta función se puede escribir de la forma:

```

;; Multiplicación de una matriz por otra
;; Precondiciones:
;; mat1 : matriz de n filas m columnas
;; mat2 : matriz de m filas p columnas
;;
(define (mul-mat-mat mat1 mat2)
  (cond ((null? mat1)
         '())
        (else
          (cons (mul-vec-mat (car mat1) mat2)
                (mul-mat-mat (cdr mat1) mat2)))))
```

## Representación de árboles

Un árbol es una estructura jerárquica sobre un conjunto de objetos llamados nodos. La jerarquía entre los nodos se establece por medio de conectores llamados ramas. Dentro de los nodos, existe un elemento especial llamado *nodo raíz*. Un nodo, al igual que los elementos de las listas puede estar constituido por cualquier tipo de objeto, sin embargo, a lo largo de los ejemplos presentados se utilizarán números o símbolos.

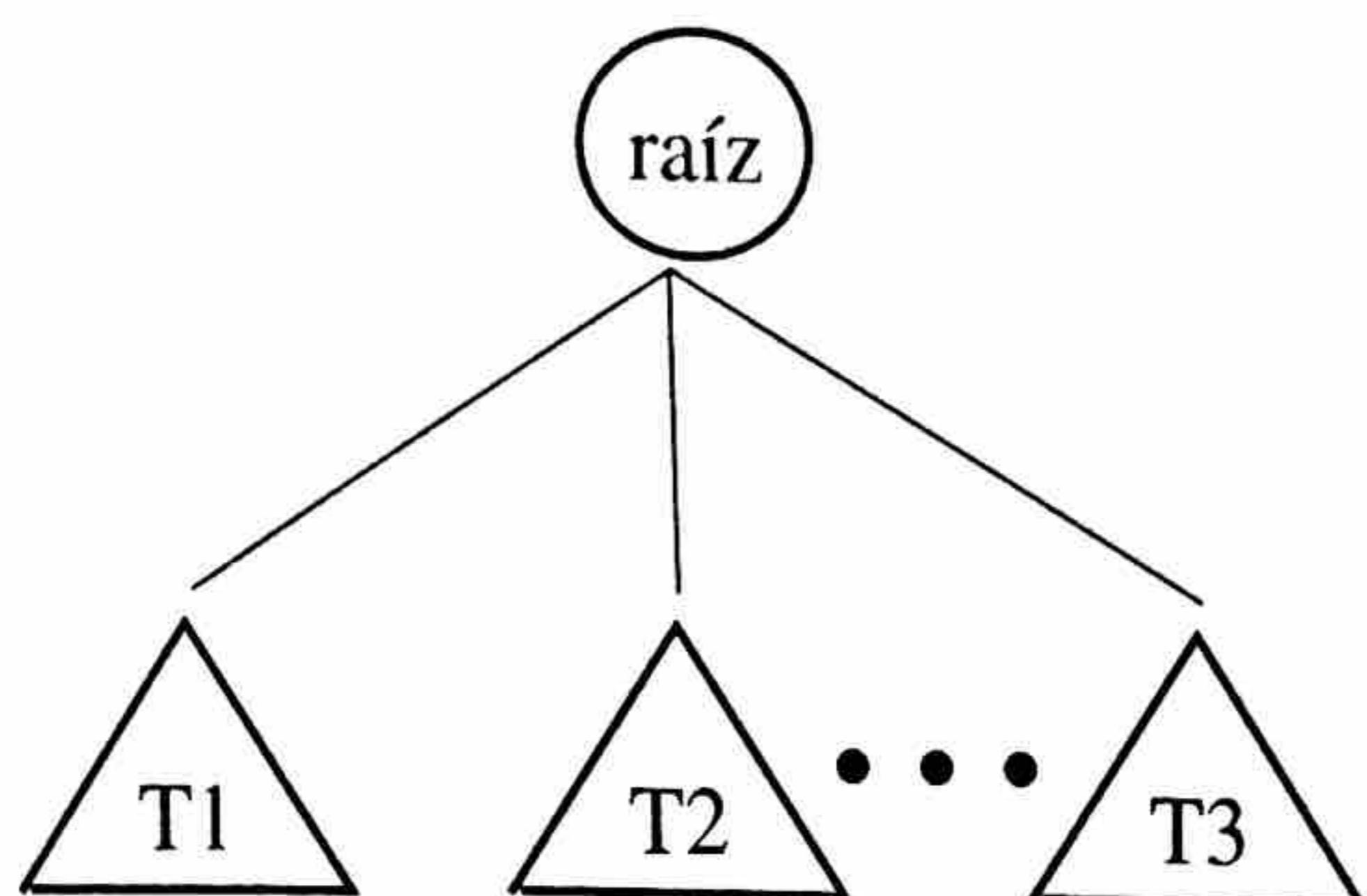
Formalmente un árbol puede definirse recursivamente de la siguiente manera:

El árbol nulo es un árbol.

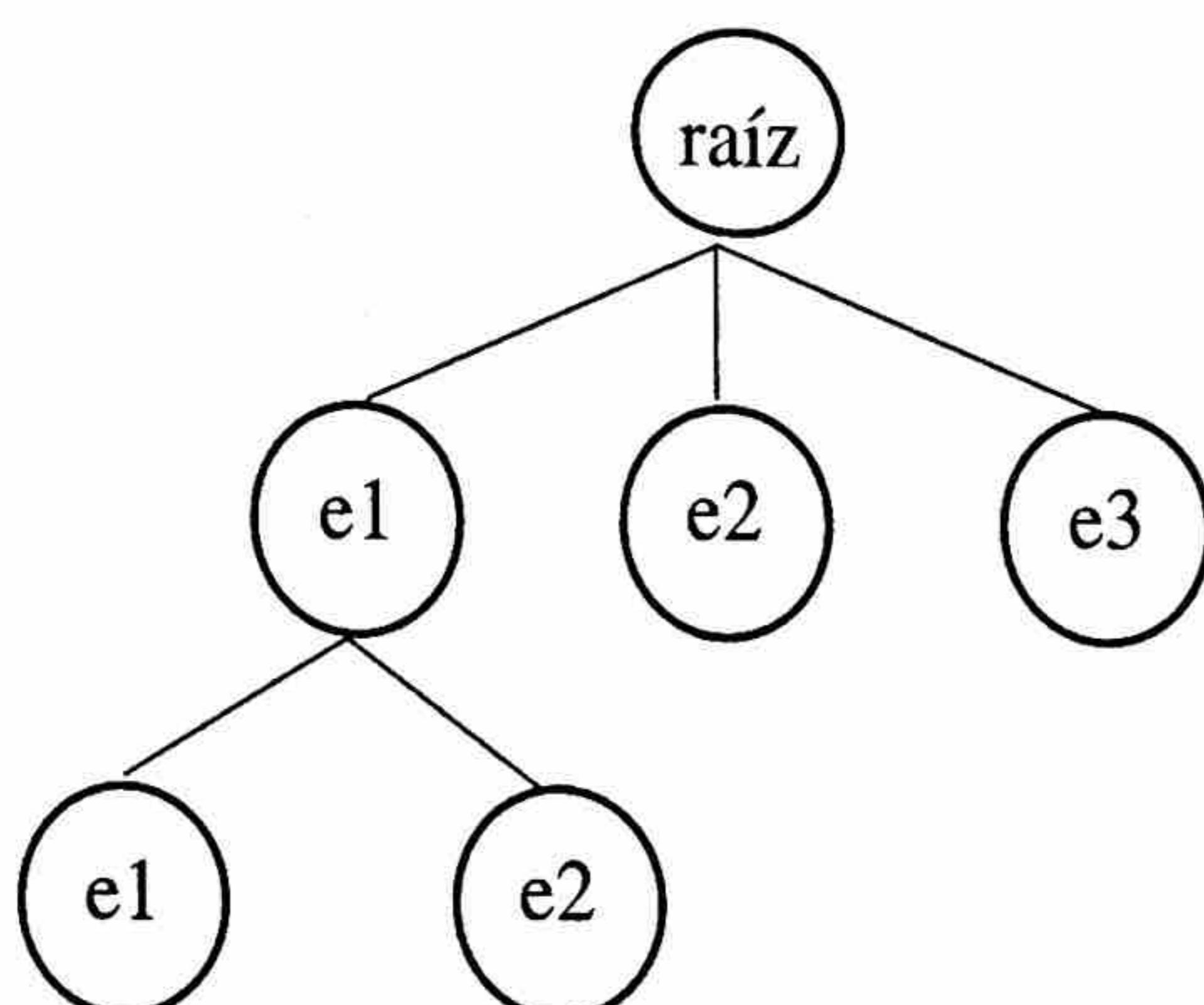
Existe un nodo especial llamado raíz.

Los nodos restantes se parten en subconjuntos  $T_1, T_2, \dots, T_n$ , con  $n > 0$ . Donde cada uno de estos elementos es a su vez un árbol. Cada uno de los elementos  $T_i, i = 1, \dots, n$  recibe el nombre de *subárbol* y posee a su vez una raíz.

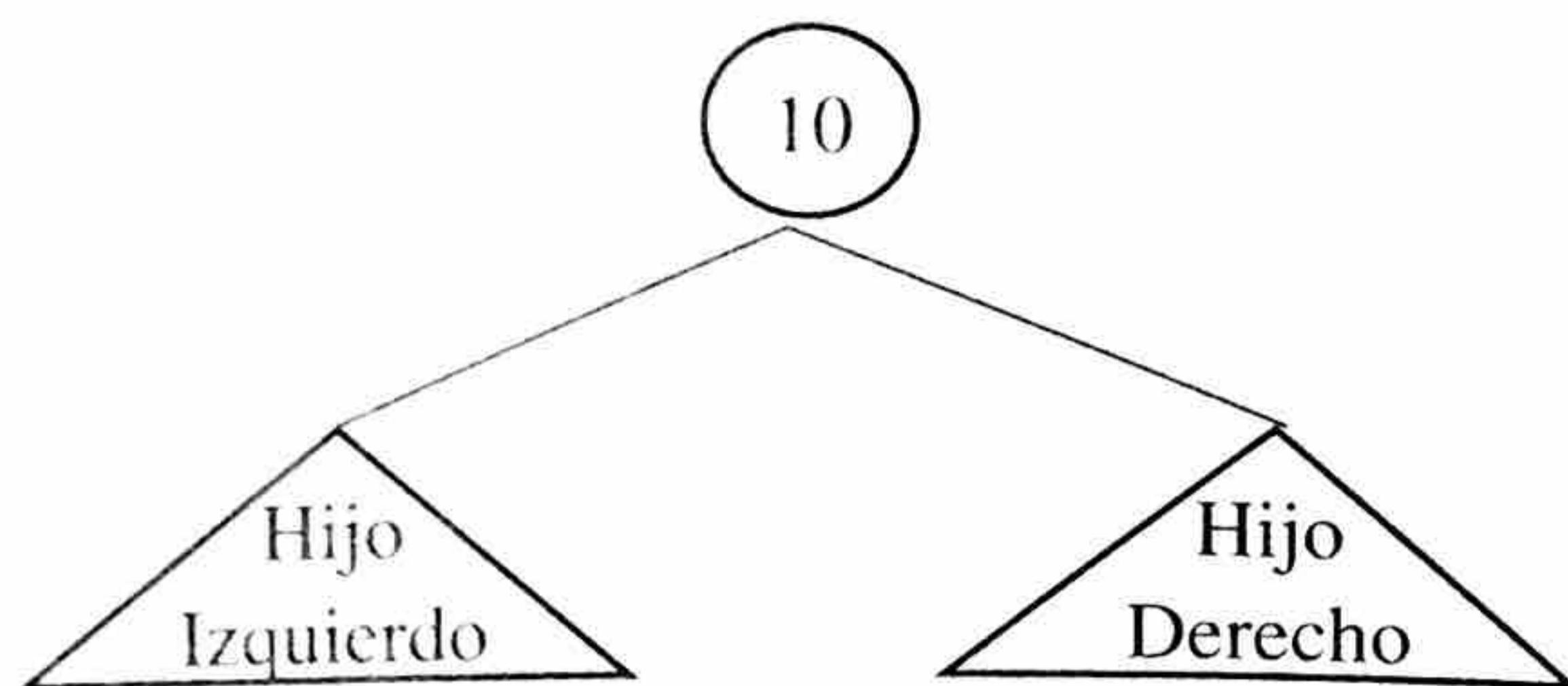
Gráficamente un árbol se puede representar como:



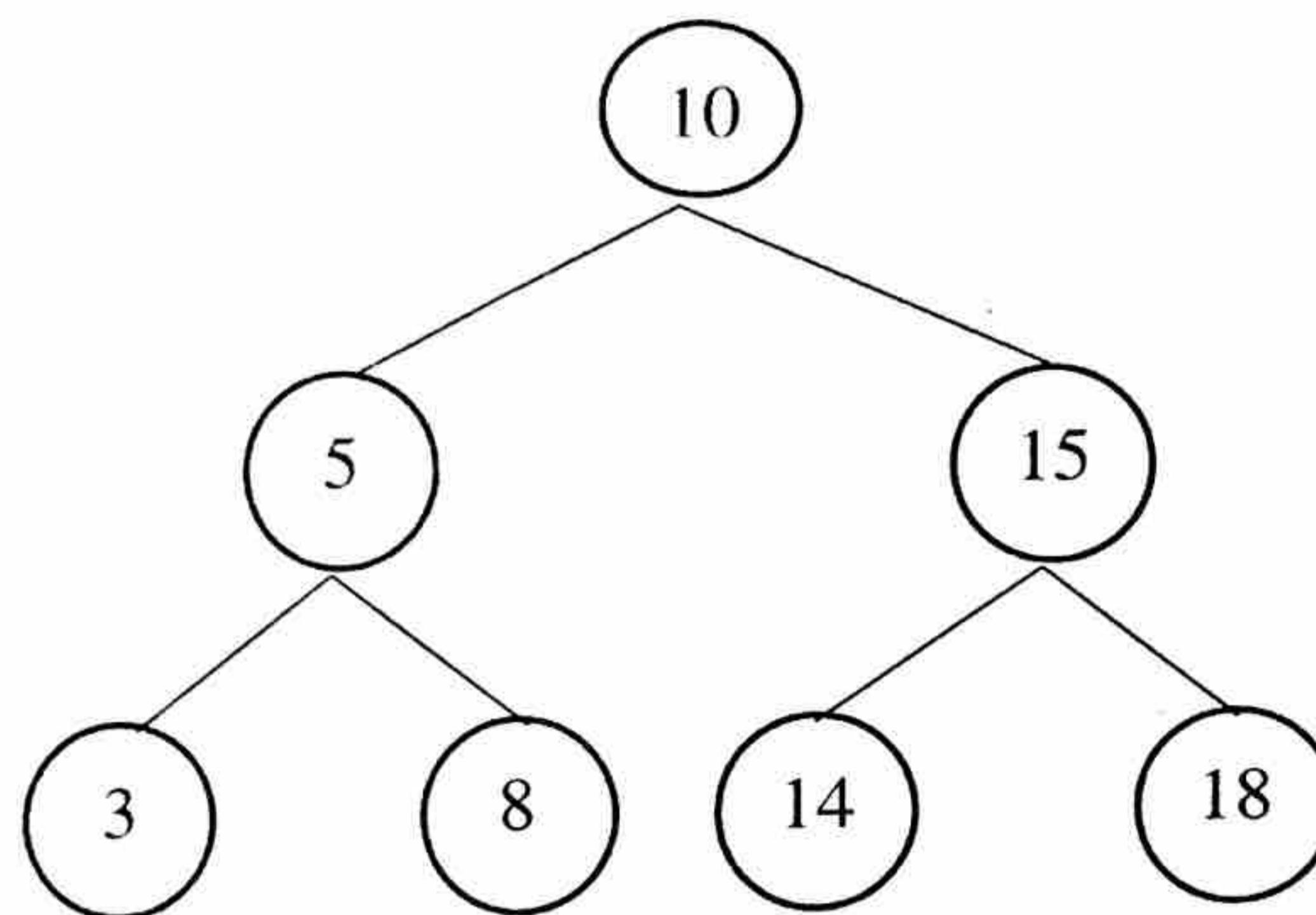
A continuación se presenta un ejemplo de un árbol.



Existe una estructura especial de árbol, denominada *árbol binario*. Un árbol binario es una estructura donde cada nodo se asocia a lo sumo con dos nodos más. En este tipo de estructura es usual nombrar al nodo raíz de un subárbol *nodo padre* y a los nodos asociados *hijo izquierdo e hijo derecho*.



A continuación se presenta un ejemplo de un árbol binario.



Para representar los árboles se usarán listas de listas de tamaño variable.

Cuando un nodo no posee hijos, se suele decir que sus hijos son nulos. Los hijos nulos pueden incluirse en la representación, o bien, pueden omitirse. De esta manera el primer árbol se puede representar mediante la cualquiera de las siguientes estructuras, o alguna combinación de ellas.

- > (define árbol-no-binario  
      '(raíz T1 T2 ... Tn))
- > (define árbol-no-binario  
      '(raíz (e1 e11 e12)  
          e2  
          (e3 e31))))
- > (define árbol-no-binario  
      '(raíz (e1 e11 e12)  
          (e2)  
          (e3 e31))))

La representación del árbol binario es mucho más sencilla, para representar los árboles binarios se utilizará la siguiente estructura: un árbol binario se definirá como una hoja, o bien como una lista con dos árboles binarios asociados. De esta forma un árbol binario puede representarse únicamente por una de estas dos estructuras:

```
> (define árbol1  
  '(10 (5 3 8)  
        (15 14 18)))  
  
> (define árbol2  
  '(10 (5 (3 () ())  
          (8 () ()))  
    (15 (14 () ())  
        (18 () ())))
```

## Operaciones básicas con árboles Binarios

Inicialmente se desarrolla una función para construir un árbol binario. Esta función debe recibir las tres partes básicas de la estructura. La raíz, el hijo izquierdo y el hijo derecho.

Se utilizará la siguiente representación, si un nodo particular no tiene hijos se presenta el nodo, en caso de tener uno o dos hijos se debe presentar una lista de la forma (raíz hijo-izquierdo hijo-derecho).

Como es natural cualquiera de los hijos podría ser un árbol nulo. Por ejemplo:

```
> (árbol 10 '() '())  
10  
  
> (árbol 10 5 '())  
(10 5 ())  
  
> (árbol 10 5 15)  
(10 5 15)  
  
> (árbol 10 '(5 3 8) '(15 14 18))  
(10 (5 3 8) (15 14 18))
```

```
> (árbol 10 (árbol 5 3 8) (árbol 15 14 18))
(10 (5 3 8) (15 14 18))
```

A continuación se presenta el código del constructor de árboles:

```
;; Construye un árbol binario de tres elementos
;; centro : un nodo
;; izq : un árbol binario
;; der : un árbol binario
;;
(define (árbol centro izq der)
  (cond( (and (null? izq)
              (null? der))
          centro)
       ( else
         (list centro izq der))))
```

Se implementarán funciones para obtener información de un árbol binario. En particular se escribe una función de nombre (raíz arb) que devuelve la raíz de un árbol, (hijo-izq arb) que devuelve el subárbol izquierdo e (hijo-der arb) que devuelve el subárbol derecho.

```
> (raíz '())
;; produce un error,
;; el árbol nulo no tiene raíz...
> (raíz '(10 (5 3 8) (15 14 18)))
10
> (hijo-izq '(10 5 15))
(5)
> (hijo-izq '(10 (5 3 8) (15 14 18)))
(5 3 8)
> (hijo-der '(10 5 15))
15
> (hijo-der '(10 (5 3 8) (15 14 18)))
(15 14 18)
> (hijo-izq (hijo-der '(10 (5 3 8) (15 14 18))))
14
```

```
> (hijo-izq (hijo-izq (hijo-der '(10 (5 3 8) (15 14 18)))))  
( )
```

El código de estas tres funciones se presenta a continuación. Observe como la función raíz devuelve como resultado un nodo, en cambio las funciones *hijo-izq* e *hijo-der* devuelven siempre un árbol.

```
;; Función auxiliar para detectar  
;; un árbol que no es representado por una lista  
;; x : un símbolo cualquiera  
;  
(define (atom? x)  
  (not (list? x)))
```

```
;; Devuelve la raíz de un árbol  
;; si es un átomo, el átomo es la raíz  
;; arb : un árbol binario  
;  
(define (raíz arb)  
  (cond ( (atom? arb)  
          arb)  
        ( else  
          (car arb))))
```

```
;; Devuelve el hijo izquierdo de un árbol  
;; si el árbol es un átomo, sus hijos son nulos  
;; arb : un árbol binario  
;  
(define (hijo-izq arb)  
  (cond ( (atom? arb)  
          '())  
        ( else  
          (cadr arb))))
```

```
;; Devuelve el hijo derecho de un árbol  
;; si el árbol es un átomo, sus hijos son nulos  
;; arb : un árbol binario  
;;
```

```
(define (hijo-der arb)
  (cond ( (atom? arb)
            '())
        ( else
            (caddr arb))))
```

Se implementará ahora una función para saber si una estructura es un árbol o una hoja del árbol. Si se trata de un árbol debe ser una lista, si se trata de una lista con un único elemento o de un átomo es una hoja. Esta función se denominará (hoja? ele) y debe producir los siguientes resultados:

```
> (hoja? '())
#f

> (hoja? '1)
#t

> (hoja? '(1 2 3))
#f

> (hoja '(1 (2 ( ) ( ))
              (3 ( ) ( )))))
#f

> (hoja? (hijo-izq '(1 2 3)))
#t

> (hoja? (hijo-izq (hijo-izq '(1 2 3))))
#f
```

Esta función se implementa mediante el siguiente código:

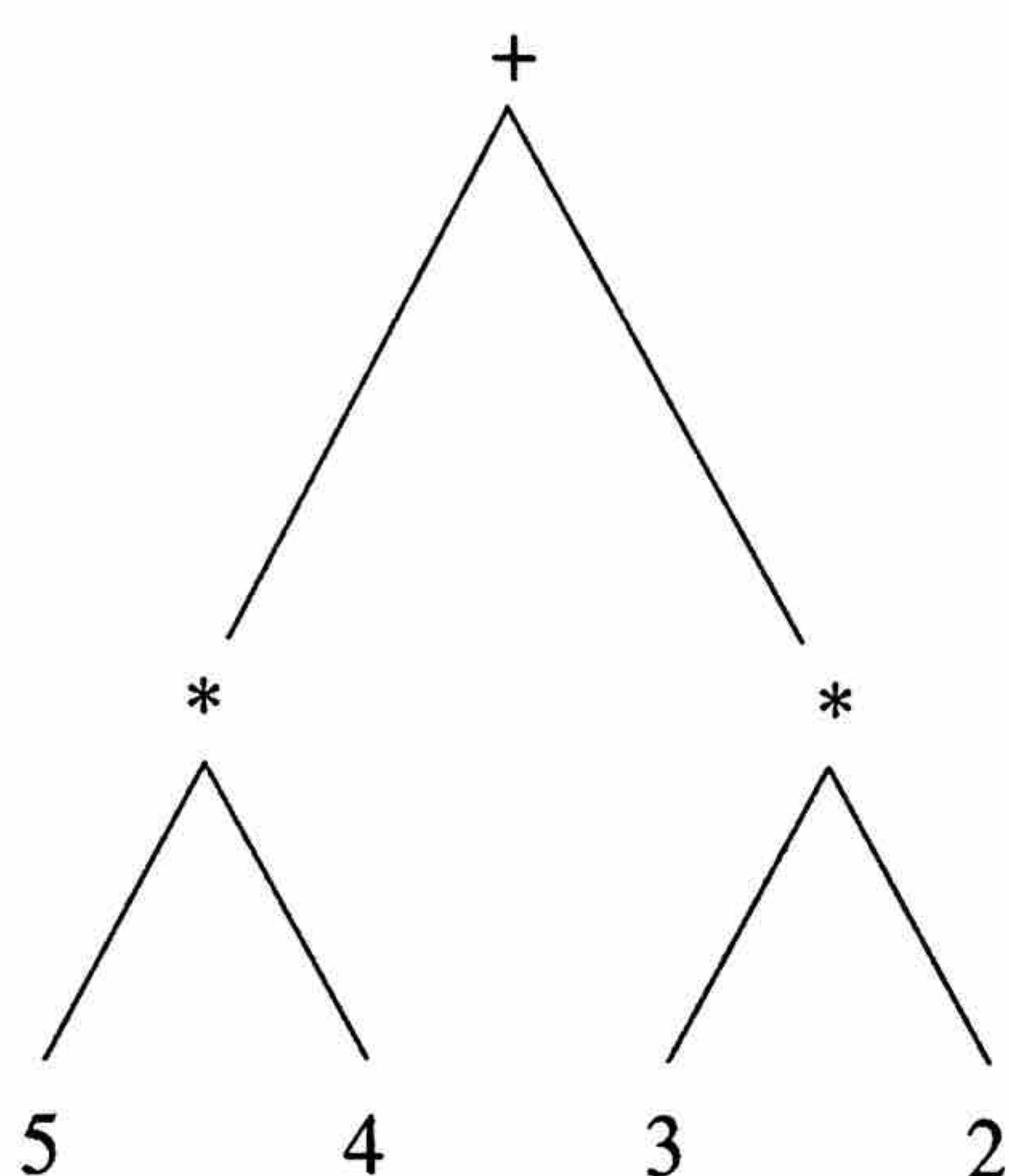
```
;; Determina si un nodo corresponde o no a una hoja
;; nodo: un nodo de un árbol
;;
(define (hoja? nodo)
  (cond ( (null? nodo)
            #f)
        ( (atom? nodo)
            #t)
        ( (and (null? (hijo-izq nodo))
               (null? (hijo-der nodo))))
```

```
#t)
( else
#f)))
```

## Representaciones

Los árboles binarios son herramientas útiles para almacenar y procesar expresiones numéricas. Cuando se almacena una expresión numérica se debe establecer el orden de aparición del operador y los operandos.

Suponga que se tiene el siguiente árbol binario:



En Scheme se ha utilizado una notación prefija para representar las expresiones numéricas. Esta notación tiene la forma:

> (operador operando-izq operando-der)

Por ejemplo el árbol descrito se expresaría de la forma:

> '(+ (\* 5 4) (\* 3 2))

La notación infija es la notación usual a la cual estamos todos acostumbrados:

> (operando-izq operador operando-der)

Por ejemplo:

> '((5 \* 4) + (3 \* 2))

La notación postfija, conocida también como notación polaca, es la notación inversa a la utilizada en el lenguaje Scheme:

(operando-izq operando-der operador)

Por ejemplo:

```
> `((5 4 *) (3 2 *)) +)
```

Si se toma una expresión prefija, se puede recorrer de manera tal que su recorrido presente la expresión final en notación infija. Se construirá la función (en-orden arb) la cual recibe un árbol binario en notación de prefija y lo reconstruye para presentarlo en infijo.

```
> (en-orden `(+ 5 4))
(5 + 4)
> (en-orden `(+ (* 5 4) 2))
((5 * 4) + 2)
> (en-orden `(+ (* 5 4) (* 3 2)))
((5 * 4) + (3 * 2))
```

El algoritmo de recorrido en-orden se presenta a continuación:

```
;; Recibe un árbol binario en pre-orden
;; y produce un árbol en-orden
;; arb: árbol binario
;;
(define (en-orden arb)
  (cond ( (null? arb)
           `())
        ( (atom? arb)
           arb)
        ( else
           (cons (en-orden (hijo-izq arb))
                 (cons (raíz arb)
                       (cons (en-orden (hijo-der arb))
                             `()))))))
```

Si se toma una expresión prefija, se puede recorrer de manera tal que su recorrido presente la expresión final en forma postfija. Se construirá la función (post-orden arb) la cual recibe un árbol binario en notación de prefija y lo

reconstruye para presentarlo postfijo. A continuación algunos ejemplos de su funcionamiento.

```
> (post-orden '(+ 5 4))
(4 5 +)
> (post-orden '(+ (* 5 4) 2))
(2 (4 5 *) +)
> (post-orden '(+ (* 5 4) (* 3 2)))
((2 3 *) (4 5 *) +)
```

El algoritmo que realiza el recorrido en postfijo de un árbol se presenta a continuación.

```
;; Recibe un árbol binario en pre-orden
;; y produce un árbol post-orden
;; arb: árbol binario
;;
(define (post-orden arb)
  (cond ( (null? arb)
           '())
        ( (atom? arb)
           arb)
        ( else
           (cons (post-orden (hijo-der arb))
                 (cons (post-orden (hijo-izq arb))
                       (cons (raíz arb)
                             ()))))))
```

## Aplastar árboles Binarios

Ocasionalmente puede ser útil convertir una estructura en otra. O de manera equivalente convertir un modelo en otro modelo conocido. La función aplastar recibe como argumento un árbol binario y produce como salida todos los elementos de ese árbol en una lista. Por ejemplo:

```
> (aplastar '())
()
```

```

> (aplastar '(1 2 3))
(1 2 3)

> (aplastar '(10 ( 5 3 8)
                  (15 14 18)))
(10 5 3 8 15 14 18)

> (aplastar '(10 ( 5 (3 (2 1 ( )) 4)
                  ( ))
                  (15 11
                     (18 17 20)))
(10 5 3 2 1 4 15 11 18 17 20)

```

El algoritmo para aplastar un árbol binario es el siguiente: aplastar un árbol nulo devuelve el árbol nulo, si se llega a un elemento sin hijos, una hoja, se devuelve este elemento, en caso contrario si se está en un subárbol se debe construir la lista que posea a la raíz en la cabeza, con el subárbol izquierdo aplastado y el subárbol derecho aplastado. A continuación el código.

```

;; Recibe un árbol binario
;; y lo convierte en una lista lineal
;; arb: árbol binario
;;
(define (aplastar arb)
  (cond ( (null? arb)
          '())
        ( else
          (append (list (raíz arb))
                  (aplastar (hijo-izq arb))
                  (aplastar (hijo-der arb)))))))

```

Veamos como trabaja con algunos ejemplos:

```

> (aplastar 8)
(append '(8)
         (aplastar ( )))
         (aplastar ( ))))

```

```
> (aplastar '(5 2 7))
  (append '(5)
    (aplastar 2)
    (aplastar 7))

  (append '(5)
    '(2)
    '(7))

  (5 2 7)

> (aplastar '(10 (5 3 8) (15 12 17)))
  (append '(10)
    (aplastar '(5 3 8))
    (aplastar '(15 12 17)))

  (append '(10)
    '(5 3 8)
    '(15 12 17))

  (10 5 3 8 15 12 17)
```

## Encontrar un elemento en un árbol

Al igual que con las listas existía la instrucción (miembro? ele lista), se construirá una función de nombre (nodo? ele arb). Como su nombre lo indica esta función devolverá el valor de #t si el elemento está en el árbol y #f en cualquier otro caso.

```
> (nodo? 2 '(10 (5 3 8) (15 13 18)))
#f

> (nodo? 5 '(10 (5 3 8) (15 13 18)))
#t

> (nodo? 18 '(10 (5 3 8) (15 13 18)))
#t

> (nodo? 18 '(10 (5 3 18) (15 13 18)))
#t
```

Un árbol vacío no puede poseer elementos, por lo tanto si se trata de este caso, se devuelve el valor de #f. Si el árbol no se encuentra vacío debe tener una raíz. Si el elemento que se busca es igual a la raíz se devuelve el valor de #t, en caso contrario se pasa a buscar el elemento en el subárbol izquierdo, si no está ahí se pasa a buscar al subárbol derecho. Una vez recorrido todo el árbol si no se ha encontrado el elemento se devuelve el valor de #f.

```

;; Busca un elemento dentro de un árbol cualquiera
;; Precondiciones:
;; ele: un nodo
;; arb: un árbol binario cualquiera
;;
(define (nodo? ele arb)
  (cond ( (null? arb)
           #f)
        ( (equal? ele (raíz arb))
           #t)
        ( (nodo? ele (hijo-izq arb))
           #t)
        ( (nodo? ele (hijo-der arb))
           #t)
        ( else
           #f)))

```

El código anterior se puede escribir uniendo todas las condiciones que producen valores de #t. De esta forma se produciría el siguiente código:

```

;; Busca un elemento dentro de un árbol cualquiera
;; Precondiciones:
;; ele: un nodo
;; arb: un árbol binario cualquiera
;;
(define (nodo? ele arb)
  (cond ( (null? arb)
           #f)
        ( (or (equal? ele (raíz arb))
              (nodo? ele (hijo-izq arb))
              (nodo? ele (hijo-der arb))))
           #t)
        ( else
           #f)))

```

```
( else  
  #f)))
```

Nodo? debe realizar los siguientes pasos:

```
> (nodo? 2 '( ))  
#f  
> (nodo? 2 7)  
(or (equal? 2 7)  
(nodo? 2 '( ))  
(nodo? 2 '( )))  
(or #f  
  #f  
  #f)  
#f  
> (or (equal? 2 10)  
  (nodo? 2 '(5 3 8))  
  (nodo? 2 '(15 13 18)))  
(or (equal? 2 10)  
  (or (equal? 2 5)  
    (nodo? 2 3)  
    (nodo? 2 8))  
  (or (equal? 2 15)  
    (nodo? 2 13)  
    (nodo? 2 18)))  
(or #f  
  (or #f  
    #f  
    #f)  
  (or #f  
    #f  
    #f))  
(or #f  
  #f  
  #f)  
#f
```

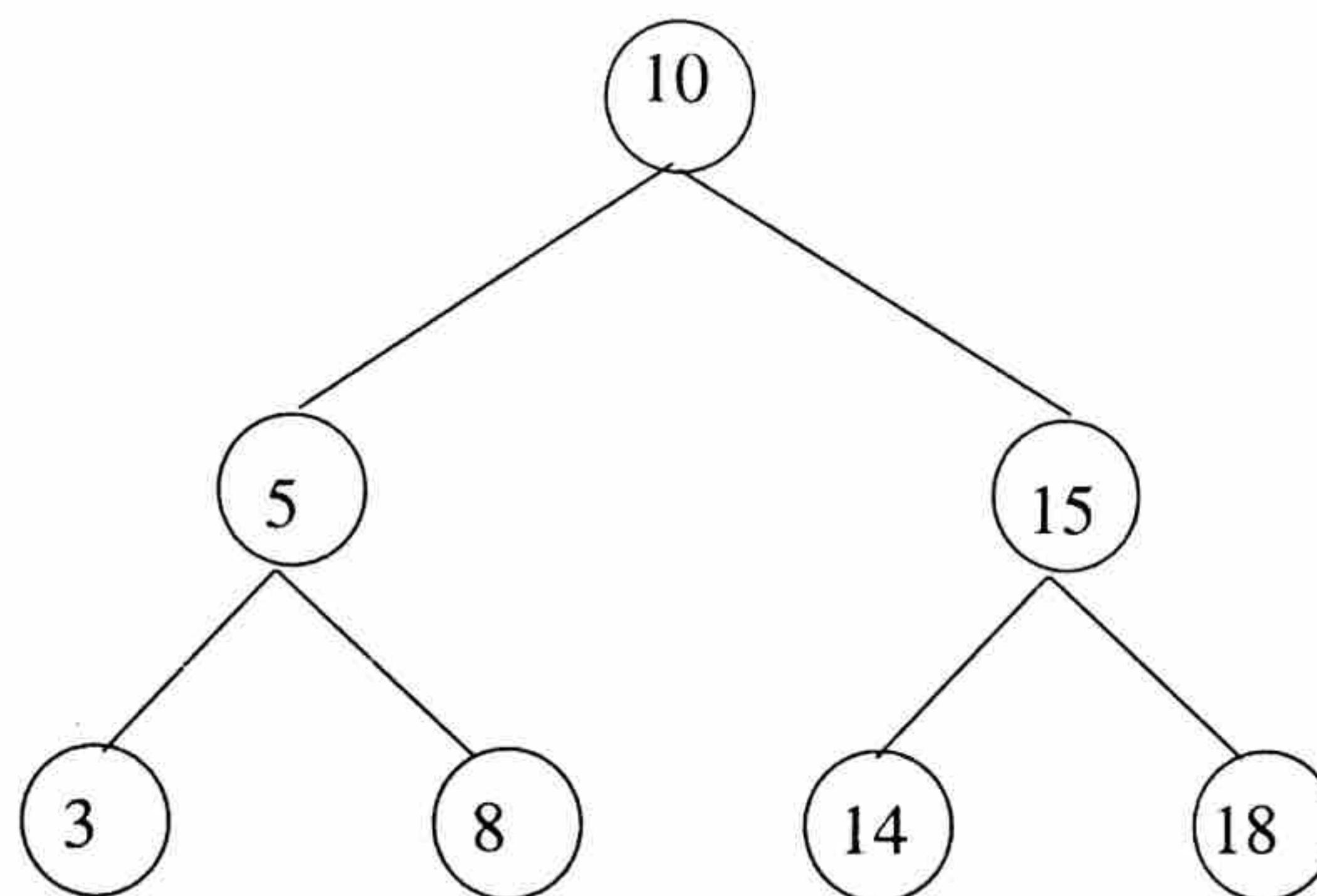
## Árboles binarios ordenados

Los árboles binarios ordenados son estructuras útiles para representar elementos que posean una relación de orden. Se utilizan para almacenar y buscar elementos. Además es una estructura sencilla de crear y mantener.

Para construir un árbol binario ordenado se mantiene el orden insertando siempre a la izquierda de la raíz los valores menores o iguales que ella y a la derecha los valores mayores. En caso que la existan hijos, se toma como nuevo árbol el subárbol correspondiente. Este proceso continua hasta encontrar una raíz sin hijos.

A continuación se presentan dos ejemplos de árboles ordenados. El primer ejemplo corresponde a

```
> '(10 ( 5 3 8)
      (15 14 8))
```

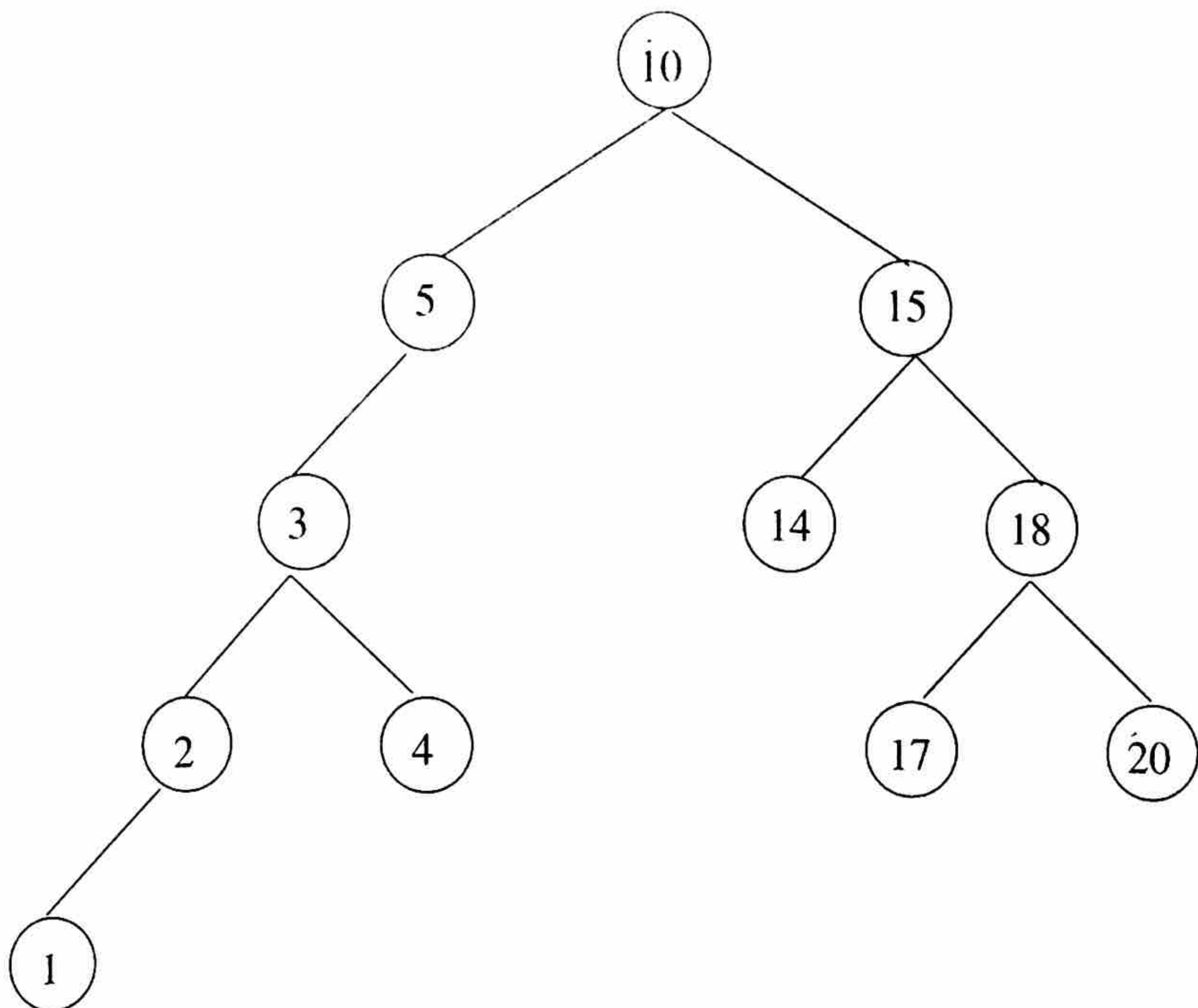


El segundo ejemplo es:

```
> '(10 ( 5 (3 (2 1 ( )) 4) ( ))
      (15 (14 ( ))(18 17 20))))
```

Una manera más fácil de visualizar el árbol es presentarlo de la siguiente manera:

```
> '(10 ( 5 (3 (2 1 ( ))
                  4)
                  ( ))
            (15 14
              (18 17 20))))
```



## Localizar elementos

La función (*localizar? ele arb*) recibe dos argumentos: un elemento y un árbol binario ordenado. Si el elemento se encuentra en el árbol binario devuelve como resultado #t, en caso contrario, devuelve el valor de #f. Su funcionamiento debe estar dado por:

```

> (localizar? 1 '(10 (5 3 8) (15 14 18)))
#f
> (localizar? 5 '(10 (5 3 8) (15 14 18)))
#t
> (localizar? 3 '(10 (5 3 8) (15 14 18)))
#t
  
```

Para construir este algoritmo se utiliza el siguiente método. Si se recibe un árbol nulo se devuelve el valor de #f, si el elemento buscado es igual que la raíz se devuelve el árbol completo, si el elemento es menor que la raíz se invoca nuevamente la función con el subárbol izquierdo, si el elemento es mayor que la

raíz se invoca nuevamente la función con el subárbol derecho. A continuación se presenta el código de la función.

```
;; Localiza un elemento dentro de
;; un árbol binario ordenado
;; ele : un símbolo
;; arb : árbol binario ordenado
;;
(define (localizar? ele arb)
  (cond ( (null? arb)
           #f)
        ( (equal? ele (raíz arb))
           #t)
        ( (< ele (raíz arb))
           (localizar? ele (hijo-izq arb)))
        ( (> ele (raíz arb))
           (localizar? ele (hijo-der arb)))))
```

Veamos como funciona el código de localizar?:

```
> (localizar? 1 '(10 (5 3 8) (15 14 18)))
(localizar? 1 '(5 3 8))
(localizar? 1 3)
(localizar? 1 '())
#f
> (localizar? 14 '(10 (5 3 8) (15 14 18)))
(localizar? 14 '(15 14 18))
(localizar? 14 14)
#t
```

Es sencillo cambiar el código anterior para que en lugar de devolver `#t` devuelva el subárbol del cual el elemento buscado es la raíz. Así se obtendría el siguiente comportamiento.

```
> (localizar 1 '(10 (5 3 8) (15 14 18)))
#f
```

```
> (localizar 5 '(10 (5 3 8) (15 14 18)))
(5 3 8)
> (localizar 3 '(10 (5 3 8) (15 14 18)))
3
```

El código correspondiente será:

```
;; Localiza un elemento dentro de
;; un árbol binario ordenado
;; ele : un símbolo
;; arb : árbol binario ordenado
;;
(define (localizar ele arb)
  (cond ( (null? arb)
           #f)
        ( (equal? ele (raíz arb))
           arb)
        ( (< ele (raíz arb))
           (localizar ele (hijo-izq arb)))
        ( (> ele (raíz arb))
           (localizar ele (hijo-der arb)))))
```

Veamos como funciona el código de localizar:

```
> (localizar 1 '(10 (5 3 8) (15 14 18)))
(localizar 1 '(5 3 8))
(localizar 1 3)
(localizar 1 '())
#f
> (localizar 15 '(10 (5 3 8) (15 14 18)))
(localizar 15 '(15 14 18))
(15 14 18)
> (localizar 14 '(10 (5 3 8) (15 14 18)))
(localizar 14 '(15 14 18))
(localizar 14 14)
```

## Insertar elementos

La función Insertar del árbol binario ordenado, se debe devolver como resultado el árbol que el elemento se encuentre en concordancia con la inserción de ordenadas.

Esta función debe producir los siguientes resultados:

```
> (insertar 10 '())
10

> (insertar 5 10)
(10 5 '())

> (insertar 5 '(10 '())())
(10 5 '())

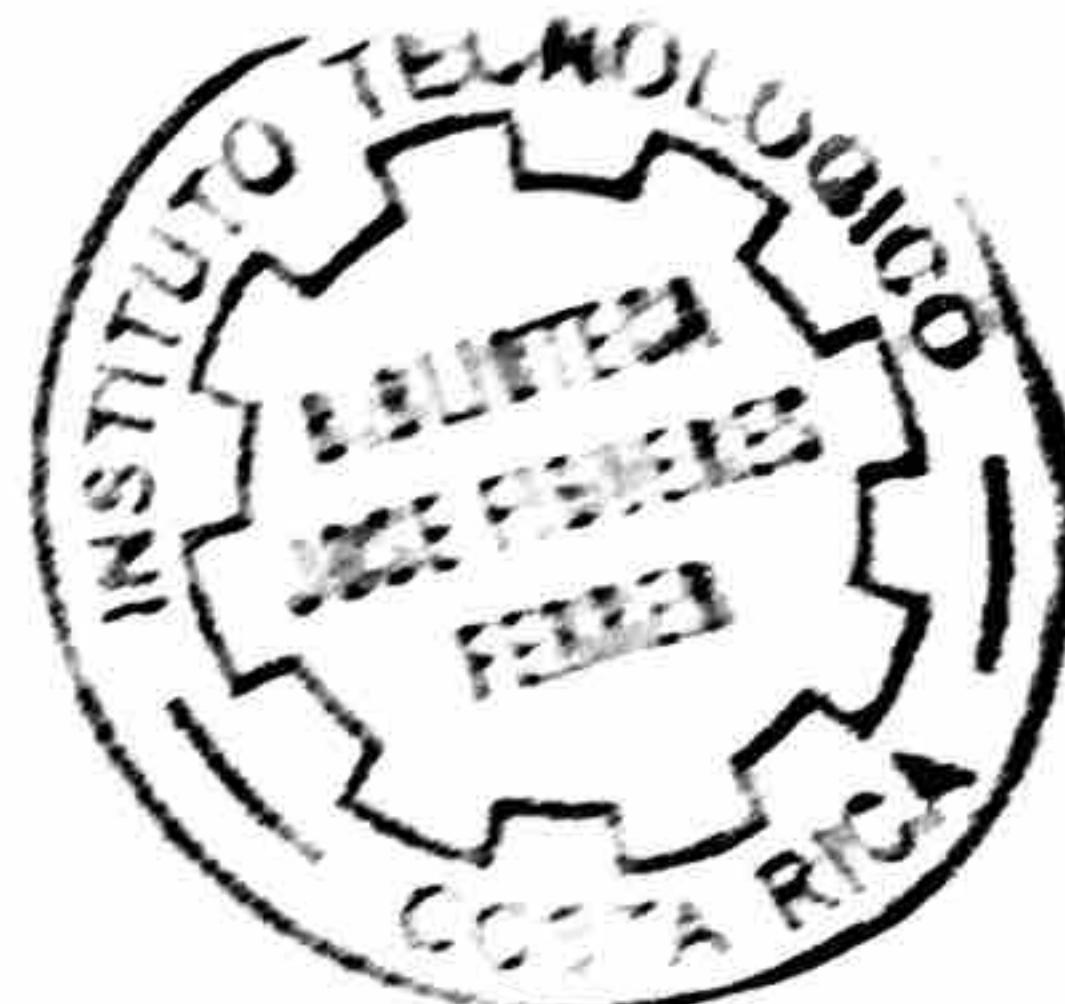
> (insertar 3 '(10 5 '())())
(10 (5 3 '())())

> (insertar 8 '(10 (5 3 '())()))
(10 (5 3 8) '())

> (insertar 15 '(10 (5 3 8) '())())
(10 (5 3 8) 15)

> (insertar 14 '(10 (5 3 8) 15))
(10 (5 3 8) (15 14 '()))

> (insertar 18 '(10 (5 3 8) (15 14 '())))
(10 (5 3 8) (15 14 18))
```



Para construir esta función se utiliza el siguiente algoritmo. Insertar un elemento en un árbol vacío devuelve dicho elemento como raíz. Si el elemento es menor o igual que la raíz se debe construir un árbol con esa misma raíz al subárbol izquierdo con el nuevo elemento a insertar y el subárbol derecho. Si el elemento es mayor que la raíz se debe construir un árbol con esa misma raíz al subárbol izquierdo y el subárbol derecho con el nuevo elemento a insertar.

```
;; Insertar un elemento en un árbol binario ordenado
;; ele: un símbolo
;; arb: árbol binario ordenado
;;
```

```
(define (insertar cle arb)
  (cond ((null? arb)
          (árbol cle '() '()))
        ((<= cle (raíz arb))
         (árbol (raíz arb)
                (insertar ele (hijo-izq arb))
                (hijo-der arb))))
        ((> cle (raíz arb))
         (árbol (raíz arb)
                (hijo-izq arb)
                (insertar ele (hijo-der arb)))))))
```

Si se ejecuta el código de insertar produciría los siguientes resultados:

```
> (insertar 10 '())
(árbol 10 '() '())
10
> (insertar 5 10)
(árbol 10
      (insertar 5 '()
                  '()))
(árbol 10
      '(5)
      '())
(10 5 ( ))
> (insertar 15 '(10 5 ( )))
(árbol 10
      5
      (insertar 15 '()))
(árbol 10
      5
      15)
(10 5 15)
```

> (insertar 3 '(10 5 15))

(árbol 10

(insertar 3 5)

15)

(árbol 10

(árbol 5

(insertar 3 '( ))

'( ))

15)

(árbol 10

(árbol 5

3

'( ))

15)

(árbol 10

'(5 3 ( ))

15)

(10 (5 3 ( )) 15)

> (insertar 7 '(10 (5 3 ( )) 15))

(árbol 10

(insertar 7 '(5 3 ( )) )

15)

(árbol 10

(árbol 5

3

(insertar 7 '( )) )

15)

(árbol 10

(árbol 5

3

7)

15)

(árbol 10  
 (5 3 7)  
 15)

(10 (5 3 7) 15)

## Eliminar elementos

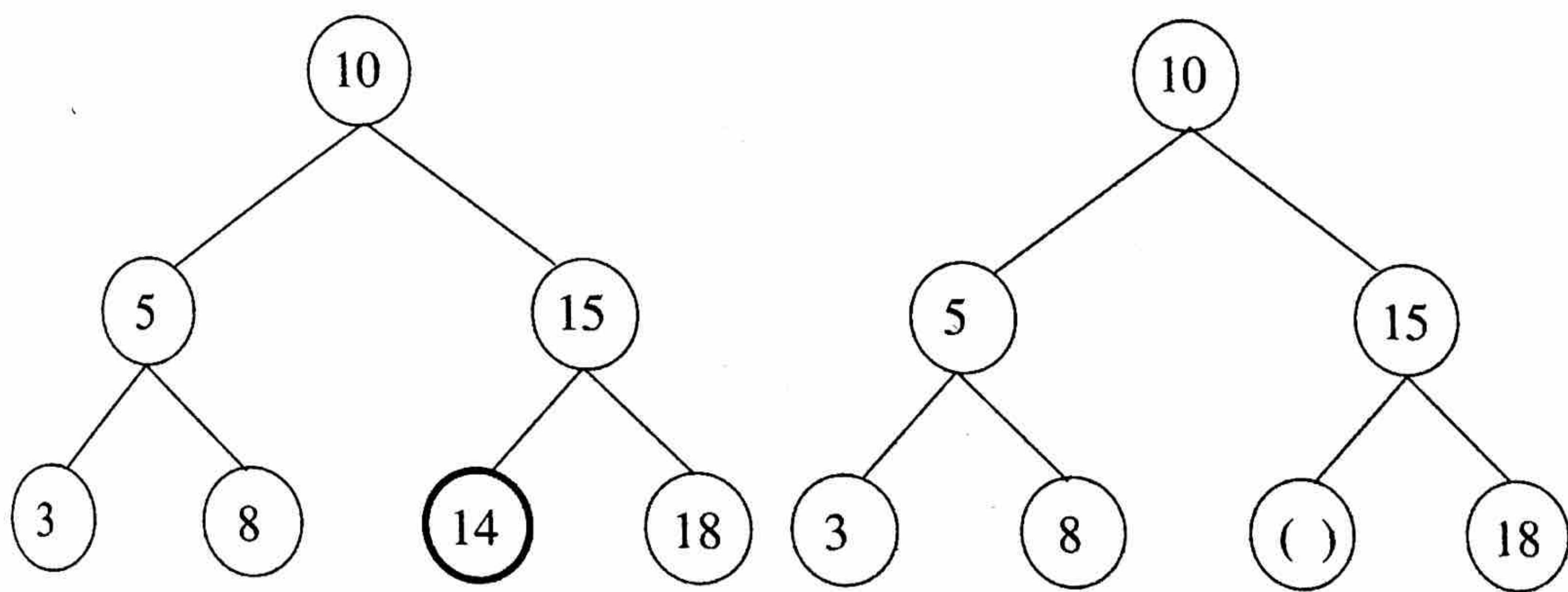
La función (*eliminar ele arb*) recibe dos argumentos un elemento y un árbol binario ordenaros. Se debe devolver como solución un nuevo árbol binario donde el elemento solicitado haya sido eliminado o borrado del árbol.

Esta función, aparentemente sencilla, posee cierto nivel de complejidad debido principalmente a los casos que se pueden presentar. A continuación se explicará cada posible caso.

- a) Un primer caso se presenta cuando se desea eliminar un nodo que no posee hijos. En ese caso es suficiente con borrar dicho nodo. Por ejemplo:

```
> (eliminar 14 '(10 (5 3 8) (15 14 18)))
(10 (5 3 8) (15 () 18))
```

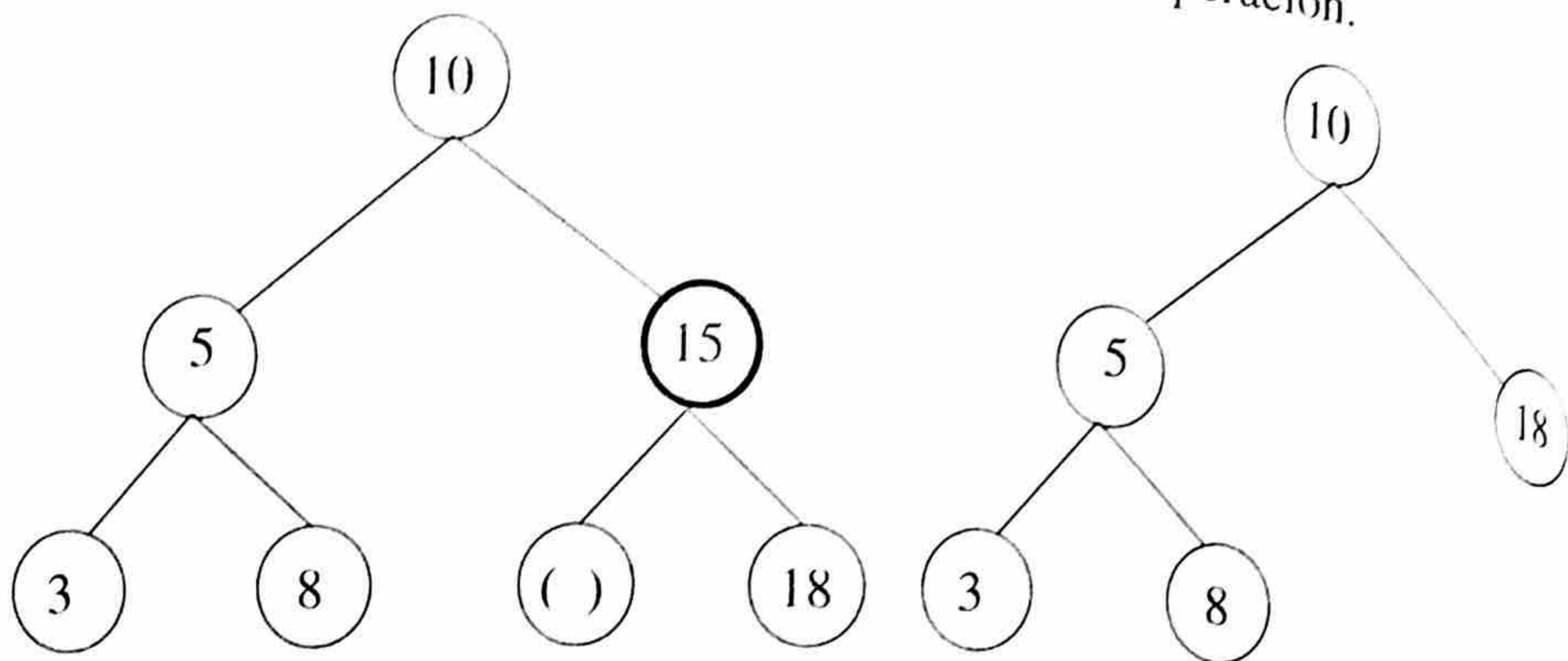
A continuación se representa gráficamente esta operación.



- b) Un segundo caso se presenta cuando se desea eliminar un nodo que posee un único hijo. Si posee sólo el hijo izquierdo, se sustituye por este, en caso contrario se substituye por el hijo derecho. Por ejemplo:

```
> (eliminar 15 '(10 (5 3 8) (15 () 18)))
(10 (5 3 8) 18)
```

A continuación se representa gráficamente esta operación.

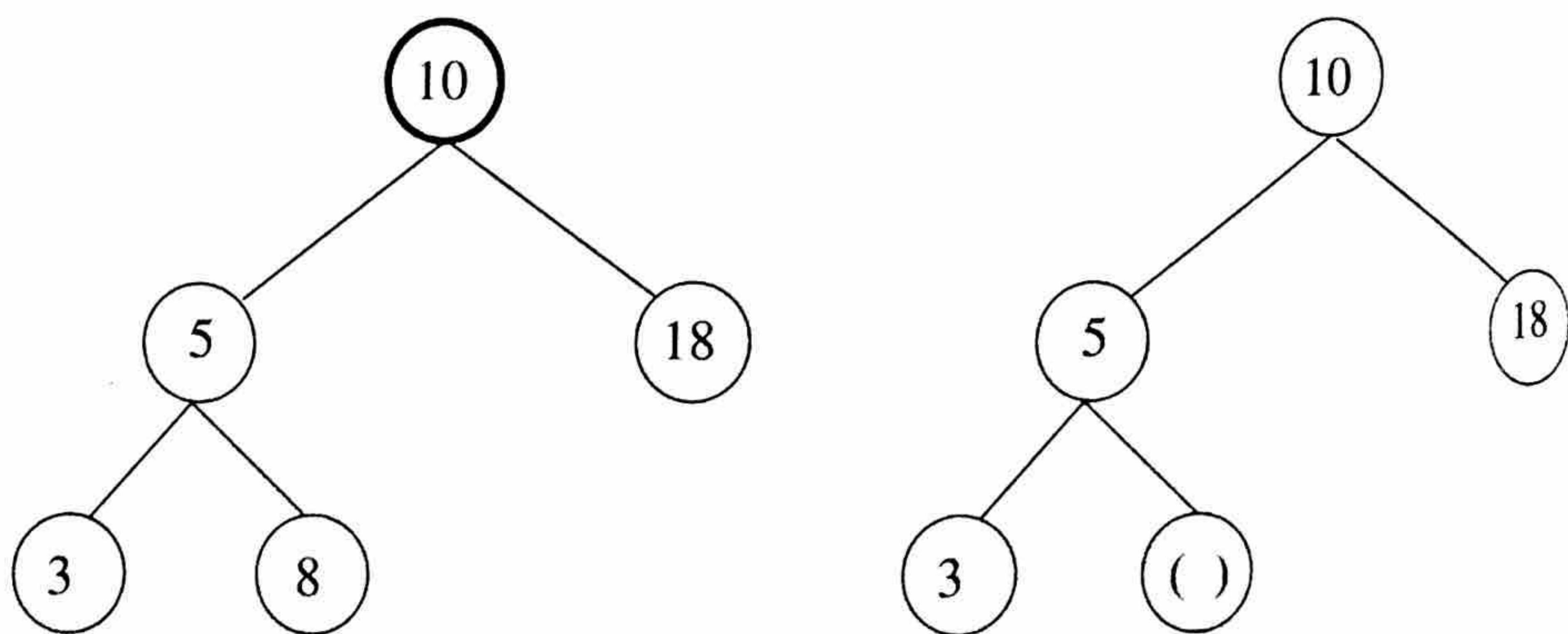


c) Un tercer caso se presenta cuando se desea eliminar un nodo que posee tanto un hijo izquierdo como un hijo derecho. En esta caso hay dos posibles cursos de acción. El primer curso de acción consiste en tomar el mayor elemento del subárbol izquierdo, o sea tomar el el mayor elemento de los menores y ponerlo como raíz. Así se garantiza que todos los elementos a la izquierda son menores que este y todos los elementos a la derecha son mayores que este. Como segunda posibilidad se puede tomar el menor elemento del subárbol derecho y se obtiene un razonamiento similar al anterior. En este caso se ha optado por la primera posibilidad. Por ejemplo:

```

> (eliminar 10 '(10 (5 3 8) 18))
(8 (5 3 ( )) 18)
  
```

A continuación se representa gráficamente esta operación.



Se presenta ahora el código de esta función. Observe como se toman cada uno de los casos descritos. Para el último caso se construye una función auxiliar

llamada (elemento-mayor arb), que devuelve el mayor elemento de un árbol o subárbol cualquiera.

;; Saca el mayor elemento de un árbol

;; arb: árbol binario ordenado

;; (define (mayor arb)

(cond ( (null? arb)

#f)

( (null? (hijo-der arb))

(raíz arb))

( else

(mayor (hijo-der arb))))

;; Elimina un elemento dentro de un árbol ordenado

;; ele: símbolo

;; arb: árbol binario ordenado

;; (define (eliminar ele arb)

(cond ( (null? arb)

'( ))

;; Se inicia la búsqueda del nodo

( (< ele (raíz arb))

(árbol (raíz arb)

(eliminar ele (hijo-izq arb))

(hijo-der arb)))

( (> ele (raíz arb))

(árbol (raíz arb)

(hijo-izq arb)

(eliminar ele (hijo-der arb))))

;; El nodo no tiene hijos

( (and (null? (hijo-izq arb))

(null? (hijo-der arb))))

'( ))

;; El nodo no tiene hijo-izq

( (null? (hijo-izq arb))

(hijo-der arb))

;; El nodo no tiene hijo-der

```

  (null? (hijo-der arb))
    (hijo-izq arb))
  ; El nodo tiene dos hijos
  (else
    (árbol (mayor (hijo-izq arb))
           (eliminar (mayor (hijo-izq arb))
                      (hijo-izq arb)))
           (hijo-der arb))))))

```

El algoritmo anterior presenta una inefficiencia al buscar el elemento mayor del subárbol izquierdo dos veces. Esto se puede corregir utilizando una instrucción de let de la siguiente manera:

```

;; Elimina un elemento dentro de un árbol ordenado
;; ele: símbolo
;; arb: árbol binario ordenado
;;
(define (eliminar ele arb)
  (cond ( (null? arb)
           '())
        ;; Se inicia la búsqueda del nodo
        ( (< ele (raíz arb))
            (árbol (raíz arb)
                  (eliminar ele (hijo-izq arb))
                  (hijo-der arb)))
        ( (> ele (raíz arb))
            (árbol (raíz arb)
                  (hijo-izq arb)
                  (eliminar ele (hijo-der arb)))))
        ;; El nodo no tiene hijos
        ( (and (null? (hijo-izq arb))
               (null? (hijo-der arb)))
               '())
        ;; El nodo no tiene hijo-izq
        ( (null? (hijo-izq arb))
            (hijo-der arb))
        ;; El nodo no tiene hijo-der
        ( (null? (hijo-der arb))

```

```

(hijo-izq arb))
; El nodo tiene dos hijos
; else
( else
  (let ( (nueva-raíz (mayor (hijo-izq arb)))
         )
    (árbol nueva-raíz
      (eliminar nueva-raíz (hijo-izq arb))
      (hijo-der arb))))))

```

Repasemos el proceso de borrado para el primer caso:

> (eliminar 14 '(10 (5 3 8) (15 14 18)))

(árbol 10  
 '(5 3 8)  
 (eliminar 14 '(15 14 18)))

(árbol 10  
 '(5 3 8)

(árbol 15  
 (eliminar 14 14)  
 18))

(árbol 10  
 '(5 3 8)  
 (árbol 15  
 '( )  
 18))

(árbol 10  
 '(5 3 8)  
 '(15 ( ) 18))

(10 (5 3 8) (15 ( ) 18))

Repasemos el proceso de borrado para el segundo caso:

> (eliminar 15 '(10 (5 3 8) (15 ( ) 18)))

(árbol 10  
 '(5 3 8)  
 (eliminar 15 '(15 ( ) 18)))

```
(árbol 10
      '(5 3 8)
      18)

(10 (5 3 8) 18)
```

Repasemos el proceso de borrado para el tercer caso:

```
> (eliminar 10 '(10 (5 3 8) 18))
```

```
(árbol 8
      (eliminar 8 (5 3 8))
      18)
```

```
(árbol 8
      (árbol 5
      3
      (eliminar 8 8))
      18)
```

```
(árbol 8
      (árbol 5
      3
      '(  ))
      18)
```

```
(árbol 8
      '(5 3(  ))
      18)
```

```
(8 (5 3 ( )) 18)
```

## Resumen

- Las listas se pueden utilizar para representar conjuntos matemáticos. Se pueden realizar las funciones básicas de conjuntos, pertenencia, unión, intersección.
- Las listas pueden representar vectores y matrices. Las matrices se pueden representar por vectores fila o por vectores columna. Se presentan las operaciones básicas entre vectores y matrices.

- Los árboles son estructuras de listas de listas. En especial los árboles binarios sirven para representar eficientemente elementos que posean alguna relación de orden mediante árboles binarios ordenados.
- Las principales operaciones sobre los árboles binarios ordenados son localizar elementos, insertar nuevos elementos y borrar elementos.

## Ejercicios

### Ejercicio N° 1

Construya una función que se llame (*insertar-elemento ele conj*). Esta función recibe dos argumentos, el primer argumento es un elemento y el segundo elemento es un conjunto. Recuerde que un conjunto no puede tener elementos repetidos. Por ejemplo:

- > (insertar-en-un-conjunto ‘a ‘( ))  
(a)
- > (insertar-en-un-conjunto ‘b ‘(a))  
(a b)
- > (insertar-en-un-conjunto ‘c ‘(a b))  
(a b c)
- > (insertar-en-un-conjunto ‘a ‘(a b c d e))  
(a b c d e)

### Ejercicio N° 2

Construya una función que se llame (*borrar-elemento ele conj*) Debe construir un nuevo conjunto donde no aparezca el elemento ele. A continuación se presentan algunos ejemplos:

- > (borrar-elemento ‘a ‘( ))  
( )
- > (borrar-elemento ‘a ‘(a b c))  
(b c)
- > (borrar-elemento ‘x ‘(a b c b))  
(a b c d)

*Ejercicio N° 3*

Construya una función que se llame (*subconjunto? conj1 conj2*). La función recibe dos argumentos y debe producir el valor de #t cuando el 1er argumento es subconjunto del segundo argumento, en caso contrario, producir el valor de #f.

Recuerde que para que el conj1 sea subconjunto del conj2, todos los elementos del 1er conjunto deben estar contenidos en el 2do. Por ejemplo:

- > (subconjunto? '() '(a b c d e f))  
#t
- > (subconjunto? '(a b c) '(a b c d e f))  
#t
- > (subconjunto? '(a b x) '(a b c d e f))  
#f

*Ejercicio N° 4*

Construya una función que se llame (*diferencia conj1 conj2*). Esta función recibe dos conjuntos como sus argumentos, la función debe construir la diferencia del primer conjunto con respecto al segundo. Recuerde que la diferencia entre conjuntos son todos los elementos que pertenecen al 1er conjunto, pero no pertenecen al segundo. Por ejemplo:

- > (diferencia '(a b c) '(a b c d e f))  
( )
- > (diferencia '(a b c) '(b c d e))  
(a)
- > (diferencia '(b c d e) '(a b c))  
(d e)
- > (diferencia '(a b x y z) '(a b c d e f))  
(x y z)

*Ejercicio N° 5*

Construya una función que se llame (*función-car eles conj*). Esta función realiza un proceso similar al de las funciones características para conjuntos, los

elementos que aparecen en el segundo conjunto deben ser sustituidos por 0s y 1s, para ello si el elemento aparece en el primer conjunto debe sustituirse por un 1, en caso contrario por un 0. A continuación se muestra su comportamiento:

> (función-car ‘(a) ‘(a b c))

(1 0 0)

> (función-car ‘(a x) ‘(a b c))

(1 0 0)

> (función-car ‘(a b d e) ‘(a e f g h b))

(1 1 0 0 0 1)

### Ejercicio N° 6

Programe una operación de nombre (*unión-múltiple lista*). Esta función recibe una lista de conjuntos y construye la unión de todos ellos. Por ejemplo:

> (unión-múltiple ‘( (a b c d e)

          (a b c x y z)

          (a b x y z t)

      ))

(a b c d e t x y z)

> (unión-múltiple ‘( (1 2 3 4)

          (1 2 5 6 7 8)

          (1 9)

          (3 4 5 6 7)

      ))

(1 2 3 4 5 6 7 8 9)

### Ejercicio N° 7

Escriba una función para realizar el producto cartesiano de conjuntos, de nombre (*prod conj1 conj2*). Esta función recibe dos conjuntos. Para cada elemento del primer conjunto crea un par con cada uno de los elementos del segundo conjunto. Esta función debe realizar la siguiente operación:

> (prod ‘(1) ‘(a))

((1 a))

```

> (prod '(1 2) '(a b))
((1 a) (1 b) (2 a) (2 b))

> (prod '(1 2 3) '(a b))
((1 a) (1 b) (2 a) (2 b) (3 a) (3 b))

> (prod '(1 2) '(a b c d))
((1 a) (1 b) (1 c) (1 d) (2 a) (2 b) (2 c) (2 d))

> (prod '(1 2 3 4 5) '(a b))
((1 a) (1 b) (2 a) (2 b) (3 a) (3 b) (4 a) (4 b) (5 a) (5 b))

```

### Ejercicio N° 8

En la representación de conjuntos utilizada, estos no poseen ninguna relación de orden. Para el desarrollo de este ejercicio se utilizarán conjuntos ordenados, por ello se utilizarán únicamente números.

Se representarán los conjuntos como conjuntos ordenados de menor a mayor. De esta forma un conjunto es de la forma:

```

> (es-conjunto? '(2 5 7 9 11))
#t

> (es-conjunto? '(2 5 11 9))
#f

> (unión '(2 5 7 9 11) '(3 5 7 10 12))
(2 3 5 7 9 10 11 12)

```

Programe las funciones siguientes sacando provecho de la nueva representación ordenada.

- (es-conjunto? conj)
- (equal-set? conj1 conj2)
- (unión conj1 conj2)
- (intersección conj1 conj2)

### Ejercicio N° 9

Construya una función que se llame (*extraer-vec vec*). Esta función extrae el elemento n-ésimo de un vector. Por ejemplo:

*José E. Hele Glitzman*

```
> (extraer-vec 1 '(11 12 13 14 15))
11
> (extraer-vec 3 '(10 11 12 13 14))
13
> (extraer-vec 7 '(10 11 12 13 14))
#f
```

### Ejercicio N° 10

Construya una función que se llame (*mul-esc-vec num vec*). Esta función realiza la multiplicación de un valor escalar por un vector. A continuación se presentan algunos ejemplos:

```
> (mul-esc-vec 2 '(1 1 1))
(2 2 2)
> (mul-esc-vec 2 '(1 2 3))
(2 4 6)
> (mul-esc-vec 3 '(4 5 7 2))
(12 15 21 6)
```

### Ejercicio N° 11

Construya una función que se llame (*vector-inverso num vector*). Esta función recibe dos argumentos, un número num y un vector con valores entre 0 y num. Debe producir un vector donde cada entrada ha sido restada al valor de num. Por ejemplo:

```
> (vector-inverso 10 '())
()
> (vector-inverso 10 '(1 2 3))
(9 8 7)
> (vector-inverso 10 '(3 9 7 0))
(7 1 3 10)
> (vector-inverso 20 '(3 9 7 0))
(17 11 13 20)
```

**Ejercicio N° 12**

Construya una función que se llame `mult-mat`. Esta función recibe como argumentos de entrada dos matrices y produce como salida la multiplicación de las matrices. Para multiplicar matrices es necesario que el número de columnas de la primera sea igual al número de filas de la segunda. A continuación se presentan algunas ejemplos:

> `mult-mat (1 2 3 4) (5 6 7 8)`

(5 6 7 8) (11 12 13 14)

> `mult-mat (1 2 3 4) (5 6 7 8)`

(5 6 7 8) (11 12 13 14) (15 16 17 18) (19 20 21 22)

**Ejercicio N° 13**

Construya una función que se llame `extraer-mat`. Esta función recibe como argumento un elemento ( $i, j$ ) de una matriz. La primera coordenada representa la fila y la segunda coordenada representa la columna. Por ejemplo:

> `extraer-mat 1 2 (11 12 13)`

(14 15 16)

(17 18 19) ()

12

> `extraer-mat 2 3 (11 12 13)`

(14 15 16)

(17 18 19) ()

16

> `extraer-mat 3 4 (11 12 13)`

(14 15 16)

(17 18 19) ()

#f

**Ejercicio N° 14**

Construya una función que se llame `(sumar-mat-mat mat mat)`. Esta función recibe como argumentos de entrada dos matrices y produce como salida la matriz resultante de la suma de los argumentos. Si M y N son dos matrices, el resultado de la suma se define como:

*José E. Hélio Guzman*

$$(M + N)[i,j] = M[i,j] + N[i,j]$$

Por ejemplo:

```
> (sumar-mat-mat `((1 1 1)
  (2 2 2)
  (1 2 3))
 `((1 0 0)
  (2 0 0)
  (3 3 3)))
```

```
((2 1 1)
 (4 2 2)
 (4 5 6))
```

### Ejercicio N° 15

Construya una función que se llame (*diagonal mat*). Esta función recibe como argumento una matriz de tamaño NxN y produce como resultado un vector con los valores de la diagonal. Un elemento pertenece a la diagonal si se encuentra en la i-ésima fila y la i-ésima columna. A continuación se presentan algunos ejemplos.

```
> (diagonal `((1 2)
  (3 4)))
 (1 4)

> (diagonal `((1 2 3)
  (1 2 3)
  (1 2 3)))
 (1 2 3)

> (diagonal `((1 2 3)
  (4 5 6)
  (7 8 9)))
 (1 5 9)

> (diagonal `(( 1 2 3 4)
  ( 5 6 7 8)
  ( 9 10 11 12)
  (13 14 15 16)))
 (1 6 11 16)
```

*Ejercicio N° 16*

Construya una función que se llame (*anti-diagonal mat*). Esta función recibe como argumento una matriz de tamaño NxN y debe extraer la diagonal de una matriz en el sentido derecha-izquierda. Por ejemplo:

> (anti-diagonal '((1 2  
                      (3 4)))

(2 3)

> (anti-diagonal '((1 2 3)  
                      (1 2 3)  
                      (1 2 3) ))

(3 2 1)

> (anti-diagonal '((1 2 3)  
                      (4 5 6)  
                      (7 8 9)))

(3 5 7)

> (anti-diagonal '(( 1 2 3 4)  
                      ( 5 6 7 8)  
                      ( 9 10 11 12)  
                      (13 14 15 16)))

(1 6 11 16)

*Ejercicio N° 17*

Utilizando las funciones anteriores de diagonal y antidiagonal, construya un proceso denominado (*qdet mat*). Qdet multiplica cada uno de los elementos de la diagonal, multiplica cada uno de los elementos de la anti-diagonal y luego suma esos dos valores. Por ejemplo:

> (qdet '((1 2 3)  
                      (4 5 6)  
                      (7 8 9)))

150

La diagonal (1 5 9), se multiplica y se obtiene.

$$1 * 5 * 9 = 45$$

La anti-diagonal  $(3 \ 5 \ 7)$ , se multiplica y se obtiene:

$$3 * 5 * 7 = 105$$

Finalmente se suman ambos valores:

$$45 + 105 = 150$$

### Ejercicio N° 18

Una matriz cuadrada, de  $N$  filas y  $N$  columnas se dice que es triangular superior si y solo sí posee ceros en todas las entradas inferiores a su diagonal. Construya una función que se llame (*triangular?* mat), que recibe un argumento y produce el valor de #t si recibe una matriz triangular superior y produce el valor de #f en cualquier otro caso. Por ejemplo:

```
> (triangular? '((1 4 1)
      (0 8 1)
      (0 0 1)))
#t
> (triangular? '((1 1 1 1)
      (0 1 1 1)
      (0 0 1 1)
      (0 0 0 1) ))
#t
> (triangular? '((1 7 8 0)
      (0 1 1 1)
      (0 0 2 1)
      (0 0 0 1)))
#t
```

### Ejercicio N° 19

Una matriz cualquiera, de  $N$  filas y  $M$  columnas se dice que es escalonada si y solo sí el número de ceros anteriores al primer elemento distinto de cero crece o se mantiene igual de fila a fila. Construya una función que se llame (*escalonada?* mat), que recibe un argumento y produce el valor de #t si recibe una matriz escalonada y produce el valor de #f en cualquier otro caso. Por ejemplo:

```

> (escalonada? '((2 3 2)
      (0 0 4)
      (0 0 0)
      (0 0 0)))
#t

> (escalonada? '((1 1 1 1)
      (0 1 1 1)
      (0 0 1 1)
      (0 0 0 1)))
#t

> (escalonada? '((2 3 2 0 4 5 6)
      (0 0 7 1 0 0 0)
      (0 0 0 0 0 1 2)
      (0 0 0 0 0 0 7)))
#t

> (escalonada? '((2 3 2 0 4 5 6)
      (0 0 7 1 3 2 0)
      (0 0 0 0 0 1 2)
      (0 0 0 0 0 0 0)))
#t

> (escalonada? '((2 3 2 0 4 5 6)
      (0 0 0 0 0 2 0)
      (0 0 0 7 2 1 2)
      (0 0 0 0 0 0 0)))
#f

```

### Ejercicio N° 20

Construya una función de nombre (*rotar matriz*). Esta función toma la primera columna y la pone como la primera.

```

> (rotar '(1 2 3)
      (4 5 6)
      (7 8 9)))
((2 3 1)
 (5 6 4)
 (8 9 7))

```

> (rotar '(( 1 2 3)  
       ( 4 5 6)  
       ( 7 8 9)  
       (10 11 12)))

(( 2 3 1)  
   ( 5 6 4)  
   ( 8 9 7)  
   (12 11 10))

### Ejercicio N° 21

Construya una función de nombre (*virus k matriz*). Esta función sustituye k elementos de la matriz iniciando en la primera fila.

> (virus 5 '((1 2 3)  
          (4 5 6)  
          (7 8 9)))

((0 0 0)  
   (0 0 6)  
   (7 8 9))

> (virus 9 '(( 1 2 3 4)  
          ( 5 6 7 8)  
          ( 9 10 11 12)  
          (13 14 15 16)))

(( 0 0 0 0)  
   ( 0 0 0 0)  
   ( 0 10 11 12)  
   (13 14 15 16)))

### Ejercicio N° 22

Construya una función de nombre (*virus k matriz*). Esta función sustituye k diagonales de la matriz iniciando en esquina superior izquierda y continuando hacia la esquina inferior derecha.

> (virus 2 '((1 2 3)  
          (4 5 6)  
          (7 8 9)))

```
((0 0 3)
 (0 5 6)
 (7 8 9))

> (virus 3 '(( 1 2 3 4)
   ( 5 6 7 8)
   ( 9 10 11 12)
   (13 14 15 16)))

(( 0 0 0 4)
 ( 0 0 7 8)
 ( 0 10 11 12)
 (13 14 15 16))

> (virus 5 '(( 1 2 3 4)
   ( 5 6 7 8)
   ( 9 10 11 12)
   (13 14 15 16)))

(( 0 0 0 0)
 ( 0 0 0 0)
 ( 0 0 0 12)
 ( 0 0 15 16))
```

### Ejercicio N° 23

Construya una función que se llame (*contar-nodos arb*). Esta función debe recibir como argumento un árbol binario cualquiera y debe producir como resultado en número total de nodos que tiene ese árbol. Por ejemplo:

```
> (contar-nodos '())
0
> (contar-nodos '(10 5 15))
3
> (contar-nodos '(10 ( 5 3 8)
   (15 14 18)))
7
> (contar-nodos '(10 ( 5 (3 (2 1 ( ))
   4)
 ( ))
```

11  
 (15 11  
 (18 17 20)))

11

## Ejercicio N° 24.

Construya una función que se llame (*contar-hojas arb*). Esta función debe recibir como argumento un árbol binario cualquiera y debe producir como resultado en número total de hojas que tiene ese árbol. En este caso una hoja es un nodo que no posee ningún hijo. Por ejemplo:

0  
 > (contar-hojas '( ))  
 1  
 > (contar-hojas '(10 5 15))  
 2  
 > (contar-hojas '(10 ( 5 3 8)  
 (15 14 18)))  
 3  
 > (contar-hojas '(10 ( 5 (3 (2 1 ( ))  
 4  
 (4)  
 ( ))  
 (15 11  
 (18 17 20))))  
 5

## Ejercicio N° 25

Construya una función que se llame (*contar-nr arb*). Esta función debe contar el número de elementos NO repetidos de un árbol binario cualquiera. Así mismo, debe realizar el conteo recorriendo únicamente una vez el árbol. Para realizar este proceso puede utilizar una lista lineal donde puede colocar cada uno de los elementos no repetidos que encuentre. Al final debe contar la cantidad de elementos que existe en esa lista.

6  
 > (contar-nr '(a (b d e) (c f g)))  
 7  
 > (contar-nr '(a (b a x) (c a y)))  
 5

*Ejercicio N° 26*

Construya una función que se llame (*altura arb*). Esta función recibe como argumento un árbol binario cualquiera e indica el número máximo de niveles que tiene. Debe encontrar la altura recorriendo únicamente una vez el árbol. Por ejemplo:

```
> (altura '(10 (5 2 6)
              (8 5 7)))
  3

> (altura '(a (b (f g h) ( ))
              (c d e)))
  4

> (altura '(a (b (c (d e ( ))
                  ( ))
                  ( ))
              x)))
  5
```

*Ejercicio N° 27*

Construya una función que se llame (*peso arb*). Esta función recibe como argumento un árbol binario cualquiera y devuelve un valor. Para calcular este valor se numeran los subárboles nulos como 0 y las hojas con el valor de 1. El padre se numera de la siguiente manera, si sus hijos tienen igual peso el peso del padre es igual a cualquiera de ellos más 1, si los hijos tienen pesos diferentes el peso del padre es el mayor de los dos. Por ejemplo:

```
> (peso '(a (b d e)
              (c f g)))
  3

> (peso '(a (b (f g h) ( ))
              (c d e)))
  3

> (peso '(a (b (c (d e ( ))
                  ( ))
                  ( ))
              x)))
  4
```

*José E. Helo Guzmán*

x))

2

Ejercicio N° 28

Dado un árbol binario ORDENADO, se desea programar una función de nombre (padre nodo arb), que devuelve el padre de un nodo, por ejemplo:

> (padre 10 '(10 ( 5 3 8)  
                     (15 14 18)))

#f

> (padre 5 '(10 ( 5 3 8)  
                     (15 14 18)))

10

> (padre 3 '(10 ( 5 3 8)  
                     (15 14 18)))

5

> (padre 14 '(10 ( 5 3 8)  
                     (15 14 18)))

15

> (padre 18 '(10 ( 5 3 8)  
                     (15 14 18)))

15

Ejercicio N° 29

Dado un árbol binario NO ORDENADO, se desea programar una función de nombre (padre nodo arb), que devuelve el padre de un nodo, por ejemplo:

> (padre 10 '(10 (12 13 19)  
                     (15 14 18)))

#f

> (padre 12 '(10 (10 (12 13 19)  
                     (15 14 18))))

10

> (padre 13 '(10 (12 13 19)  
                     (15 14 18))))

12



226

> (padre 14 '(10 (12 13 19)  
                       (15 14 18)))

15  
 > (padre 18 '(10 (12 13 19)  
                       (15 14 18)))

15

**Ejercicio N° 30**

Utilice la función para (*insertar ele arb*) elementos en un árbol binario ordenado con los siguientes datos en el orden establecido: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1. Analice el árbol resultante. ¿Nota algo extraño en la estructura de ese árbol?

**Ejercicio N° 31**

Construya una función que se llame (*insertar-nr ele arb*). Esta función inserta un elemento en un árbol binario ordenado si y solo si el elemento que va a ser insertado no se encuentra en el árbol.

**Ejercicio N° 32**

Utilizando como base la función para eliminar un elemento de un árbol binario ordenado, construya una nueva función que al eliminar un nodo con dos hijos lo sustituya por el menor elemento del hijo derecho, en lugar de sustituirlo por el mayor elemento del hijo izquierdo.

**Ejercicio N° 33**

Construya una función que se llame podar-árbol. Esta función debe recibir como argumento un árbol binario ordenado y debe producir como resultado un árbol binario que no posea elementos repetidos. Por ejemplo:

> (podar-árbol '(10 ( 5 5 8)  
                       (15 14 18)))

(10 (5 ( ) 8)  
                       (15 14 18)))

**Ejercicio N° 34**

Construya una función que se llame (*invertir-árbol arb*). Esta función debe recibir como argumento un árbol binario ordenado y debe producir como

~~Hele' Cilt. 111~~  
resultante de un árbol binario con todos sus hijos invertidos, todo aquél que era hijo de un árbol binario debe ser un hijo derecho y viceversa. La relación de orden del resultado, ahora debe ser inversa a la del árbol original, a la izquierda de la raíz resultante debe ser inversa a la del árbol original, a la izquierda de la raíz resultante deben ser elementos mayores y a la derecha elementos menores o iguales. Por ejemplo:

es. por ejem.  
> (invertir-árbol .( ))

( ) cartir-árbol '(10 5 15))

> (invert  
(10 15 5))

(10 15 2) invertir-árbol (10 (5 3 8) (15 14 1)

```
> (invertin (15 14 18)))
```

$$\begin{pmatrix} 10 & (15 & 18 & 14) \\ & (5 & 8 & 3) \end{pmatrix}$$

## Ejercicio N° 35

*Ejercicio IV*

Todos los árboles binarios presentados en este capítulo guardaban en sus nodos valores numéricos. Cuando se permite insertar elementos repetidos, estos se almacenan en lugares diferentes del árbol. Construya una nueva estructura de árboles binarios ordenados donde en lugar de un número un nodo está constituido por un lista de dos elementos, el 1er elemento indica el valor guardado, el segundo elemento el número de veces que ha sido guardado dicho valor. Por ejemplo:

```
> (insertar 10 '( ))
```

((10 1) ( ))

```
> (insertar 5 '( (10 1) ))
```

$$\begin{pmatrix} 10 & 1 \\ 5 & 1 \end{pmatrix}$$

```
> (insertar 15 '((10 1) (5 1)
   ( ))
```

$$\begin{pmatrix} (10 & 1) & (5 & 1) \\ & (15 & 1) \end{pmatrix}$$

```
> (insertar 5 '((10 1) (5 1)
   (15 1)))
```

$((10\ 1)\ (5\ 2)$   
 $\quad (15\ 1))$

```
> (insertar 5 `((10 1) (5 2)
((10 1) (5 3)
(15 1)))
```

## Ejercicio N° 36. Programación Avanzada. Operaciones con Árboles N-arios

Todos los árboles considerados hasta el momento han sido árboles binarios. Si se permite que un padre tenga más de dos hijos se denominan simplemente **árboles N<sub>0</sub>** (sugerencia, puede utilizar la instrucción MAP):

- a) (contar-nodos arb)
  - b) (contar-hojas arb)
  - c) (altura arb)

## *Ejercicio N° 37. Programación Avanzada. Partes de un Conjunto*

**Subconjunto**  
Construya una función en Scheme que construya todos los subconjuntos de una lista. Dicha función se debe llamar (`partes conj`) y debe tener el siguiente comportamiento.

```
> (partes '( ))
(( ))  
  
> (partes '(1 2))
(( )
(1) (2)
(1 2)
)  
  
> (partes '(1 2 3))
(( )
(1) (2) (3)
```

$$\begin{pmatrix} 1 & 2 \\ 1 & 2 & 3 \end{pmatrix} (1 \ 3) \ (2 \ 3) \ (2 \ 3)$$

)  
> (partes '(1 2 3 4))

$$\begin{pmatrix} ( ) \\ (1) & (2) & (3) & (4) \\ (1 \ 2) & (1 \ 3) & (1 \ 4) & (2 \ 3) & (2 \ 4) & (3 \ 4) \\ (1 \ 2 \ 3) & (1 \ 2 \ 4) & (1 \ 3 \ 4) & (2 \ 3 \ 4) \\ (1 \ 2 \ 3 \ 4) \end{pmatrix}$$

)