METODOLOGÍA DE LA PROGRAMACIÓN
EJERCICIOS DEL TEMA 5
ESPECIFICACIÓN ECUACIONAL DE TAD

A) Operaciones sobre listas

Dar ecuaciones que definan las siguientes operaciones sobre los tipos de datos [Int] o [t] según el caso:

1) incr

Función que dada una lista de enteros devuelve la lista cuyos elementos son los elementos de la lista inicial incrementados en 1: *incr*.

Ejemplos:

incr ([4, 8, 5]) = [5, 9, 6]
incr ([]) = []

2) sumar

Función que dada una lista de enteros devuelve la suma de los elementos que conforman la lista: *sumar*.

Ejemplos:

sumar([4, 8, 5]) = 17
sumar([]) = 0

3) algun_par

Función que dada una lista de enteros devuelve True si hay algún par en la lista y False en caso contrario: *algun_par*.

Ejemplos:

algun_par([4, 8, 5]) = True
algun_par([11, 7, 5, 21]) = False
algun_par([]) = False

Utilizar las siguientes operaciones ya definidas: **par**, **impar**

4) coinciden_par

Función que dadas dos listas de enteros decide si siempre que en una posición de la primera lista hay un valor par en la misma posición de la segunda lista también hay un valor par y si siempre que en una posición de la primera lista hay un valor impar en la misma posición de la segunda lista también hay un valor impar. Si las dos listas no tienen la misma longitud, se genera un error: *coinciden_par*.

Ejemplos:
coinciden_par([4, 8, 3], [2, 8, 11]) = True
coinciden_par([4, 8, 3], [9, 2, 17]) = False
coinciden_par([5, 7], [1, 9, 3]) = error "distinta longitud"

Utilizar las siguientes operaciones ya definidas: **longitud**, **es_vacia, primero, resto, par**, **impar**

5) ultimo

Función que dada una lista devuelve el último elemento de la lista, es decir, el que está más a la derecha. Si la lista es vacía, ha de devolver un mensaje de error: *ultimo*.

Ejemplos: (los *di* son elementos de tipo t)

ultimo([d4, d8, d3, d5, d2]) = d2
ultimo([]) = error "Lista vacía"

Utilizar la siguiente operación ya definida: **es_vacia**

6) sin_ultimo

Función que dada una lista devuelve la lista que se obtiene eliminando el último elemento. Si la lista es vacía debe devolver un mensaje de error: *sin_ultimo*.

Ejemplos: (los *di* son elementos de tipo t)

sin_ultimo([d4, d8, d3, d5, d2]) = [d4, d8, d3, d5]
sin_ultimo([]) = error

Utilizar la siguiente operación ya definida: **es_vacia**

7) inversa

Función que dada una lista calcula su inversa: *inversa*.

Ejemplos: (los *di* son elementos de tipo t)

inversa([d4, d8, d3, d5, d2]) = [d2, d5, d3, d8, d4]
inversa([]) = []

Hacerlo de dos formas:

a) **Utilizando** la siguiente operación ya definida: ++
b) **Utilizar** las siguientes operaciones ya definidas: **ultimo**, **sin_ultimo**

8) son_inv

Función que dadas dos listas decide si son inversas: *son_inv*.

Ejemplos: (los *di* son elementos de tipo t)

son_inv ([d4, d8, d3], [d3, d8, d4]) = True
son_inv ([], []) = True
son_inv ([d1, d8, d6], [d3, d8]) = False
son_inv ([], [d3, d8]) = False

Hacerlo de dos formas:

a) **Utilizando** la siguiente función ya definida: **inversa**
b) **Utilizando** las siguientes operaciones ya definidas: **ultimo**, **sin_ultimo**, **longitud** y **es_vacia**

9) nveces

Función que cuenta el número de veces que un elemento dado aparece en una lista: *nveces*.

Ejemplos: (los *di* son elementos de tipo t)

nveces(d8, [d3, d7, d8, d20, d8, d7]) = 2
nveces(d10, [d3, d7, d8, d20, d8, d7]) = 0
nveces(d6, []) = 0

10) hay_rep

Función que decide si en una lista hay elementos repetidos: *hay_rep*.

Ejemplos: (los *di* son elementos de tipo t)

hay_rep([d5, d6, d20, d6, d6, d10, d34]) = True
hay_rep([d5, d6, d20, d6, d6, d5, d34]) = True
hay_rep([d5, d6, d20, d8, d7]) = False
hay_rep([]) = False

Utilizar la siguiente operación ya definida: **esta**

11) eliminar

Función que elimina de una lista todas las apariciones de un elemento dado: *eliminar*.

Ejemplos: (los *di* son elementos de tipo t)

eliminar(d6, [d5, d6, d20, d6, d6, d10, d34]) = [d5, d20, d10, d34]
eliminar(d6, [d5, d20, d34]) = [d5, d20, d34]
eliminar(d2, []) = []

12) eliminar_pares

Función que elimina de una lista de enteros todos los números pares: *eliminar_pares*.

Ejemplos:

eliminar_pares([5, 6, 20, 11, 7, 10, 34]) = [5, 11, 7]
eliminar_pares([4, 20, 18]) = []
eliminar_pares([]) = []

Utilizar las siguientes operaciones ya definidas: **par**, **impar**

13) eliminar_pos_pares

Función que elimina de una lista los elementos de las posiciones pares: *eliminar_pos_pares*.

Ejemplos: (los *di* son elementos de tipo t)

eliminar_pos_pares([d5, d6, d20, d7, d6, d10, d34]) = [d5, d20, d6, d34]
eliminar_pos_pares([d5, d20, d34]) = [d5, d34]
eliminar_pos_pares([]) = []

Utilizar las siguientes operaciones ya definidas: **es_vacia, resto**

14) eliminar_pos_impares

Función que elimina de una lista los elementos de las posiciones impares: *eliminar_pos_impares*.

Ejemplos: (los *di* son elementos de tipo t)

eliminar_pos_impares([d5, d6, d20, d7, d6, d10, d34]) = [d6, d7, d10]
eliminar_pos_impares([d5, d20, d34]) = [d20]
eliminar_pos_impares([]) = []

Utilizar las siguientes operaciones ya definidas: **es_vacia**, **primero, resto**

15) eliminar_rep

Función que elimina las repeticiones de elementos de una lista conservando la primera aparición de cada elemento: *eliminar_rep*.

Ejemplos: (los *di* son elementos de tipo t)

eliminar_rep([d5, d6, d20, d6, d6, d10]) = [d5, d6, d20, d10]
eliminar_rep([d4, d4, d8, d22, d8, d22]) = [d4, d8, d22]
eliminar_rep([]) = []

Utilizar las siguientes operaciones ya definidas: **esta**, **eliminar**

16) eliminar_rep2

Función que elimina las repeticiones de elementos de una lista conservando la última aparición de cada elemento: *eliminar_rep2*.

Ejemplos: (los *di* son elementos de tipo t)

eliminar_rep2([d5, d6, d20, d6, d6, d10]) = [d5, d20, d6, d10]
eliminar_rep2([d4, d4, d8, d22, d8, d5]) = [d4, d22, d8, d5]
eliminar_rep2([]) = []

Utilizar las siguientes operaciones ya definidas: **esta**

17) par_igual

Función que decide si una lista tiene al menos un par de elementos consecutivos iguales: *par_igual*.

Ejemplos: (los *di* son elementos de tipo t)

par_igual([d5, d9, d3, d3, d2]) = True
par_igual([d5, d3, d3, d3, d2]) = True
par_igual([d5, d5, d1, d2, d2]) = True
par_igual([d5, d9, d3, d2, d3]) = False
par_igual([]) = False

Utilizar las siguientes operaciones ya definidas: **es_vacia**, **primero**

18) es_prefijo

Función que decide si una lista es prefijo de otra lista: *es_prefijo*.

Ejemplos: (los *di* son elementos de tipo t)

es_prefijo([d5, d9], [d5, d9, d3, d3, d2]) = True
es_prefijo([], [d5, d9, d3, d3, d2]) = True
es_prefijo([d7, d5, d10], [d5, d9, d3, d3, d2]) = False
es_prefijo([d5, d9, d8, d4], [d5, d9]) = False
es_prefijo([d5, d6], []) = False

Utilizar las siguientes operaciones ya definidas: **es_vacia**, **primero, resto**

En general una lista s es prefijo de una lista r si existe una lista w tal que s ++ w = r (por ello [] es prefijo de cualquier lista r ya que [] ++ r = r)

19) es_sublista

Función que decide si una lista es sublista de otra lista: *es_sublista*.

Ejemplos: (los *di* son elementos de tipo t)

es_sublista([d9, d3], [d5, **d9, d3**, d3, d2]) = True
es_sublista([d5, d9, d3], [d5, **d9, d3**, d3, d2]) = True
es_sublista([], [d5, d9, d3, d3, d2]) = True
es_sublista([d2, d5, d3], [d5, d9, d3, d3, d2]) = False
es_sublista([d5, d9, d8, d4], [d5, d9]) = False
es_sublista([d5, d6], []) = False

En general una lista s es sublista de una lista r si existen dos listas v y w tal que v ++ s ++ w = r. Por ello, [] es sublista de cualquier lista r ya que [] ++ [] ++ r = r (nótese que en ese caso v sería [] y w sería r).
Utilizar las siguientes operaciones ya definidas: **es_prefijo, es_vacia, resto**

20) elem_pos

Función que dado una posición (un número entero) y una lista, devuelve el elemento de la lista que ocupa dicha posición. Si la posición no está en el rango adecuado (entre 1 y la longitud de la lista), se produce un error: *elem_pos*.

Ejemplos: (los *di* son elementos de tipo t)

elem_pos(2, [d5, **d9**, d3, d3, d2]) = d9
elem_pos(8, [d5, d9, d3, d3, d2]) = error "No es adecuado"
elem_pos(0, [d5, d9, d3, d3, d2]) = error "No es adecuado"
elem_pos(2, []) = error "No es adecuado"

Utilizar la siguiente operación ya definida: **longitud**

21) insertar

Función que dado una posición (un número entero), un elemento de tipo t y una lista, devuelve la lista que se obtiene insertando el elemento en la posición indicada y desplazando los que quedan a la derecha. Si la posición no está en el rango adecuado (entre 1 y longitud de la lista más 1), se produce un error: *insertar*.

Ejemplos: (los *di* son elementos de tipo t)

insertar(2, d8, [d5, **d9**, d3, d3, d2]) = [d5, **d8**, d9, d3, d3, d2]
insertar(6, d8, [d5, d9, d3, d3, d2]) = [d5, d9, d3, d3, d2, **d8**]
insertar(1, d8, [d5, d9, d3, d3, d2]) = [**d8**, d5, d9, d3, d3, d2]
insertar(0, d8, [d5, d9, d3, d3, d2]) = error "No es adecuado"
insertar(9, d8, [d5, d9, d3, d3, d2]) = error "No es adecuado"
insertar(1, d8, []) = [d8]
insertar(2, d8, []) = error "No es adecuado"

Utilizar la siguiente operación ya definida: **longitud**

22) pos_primer_par

Función que, dada una lista de enteros devuelve la posición del primer elemento par de la lista. Si no hay ningún elemento par, devuelve el número de elementos de la lista más 1: *pos_primer_par*.

Ejemplos:

pos_primer_par([5, 9, 8, 7, 6]) = 3
pos_primer_par([9, 5, 5, 3, 79]) = 6

Desarrollar paso a paso los siguientes casos:

pos_primer_par([5, 9, 8, 6])
pos_primer_par([5, 9])

23) pos_ult_apar

Función que, dados un número entero y una lista de enteros devuelve la posición de la última aparición del número en la lista. Si el número no aparece en la lista ha de devolver el valor 0: *pos_ult_apar*.

Ejemplos:

pos_ult_apar(8, [8, 9, 8, 17, 6]) = 3
pos_ult_apar(8, [7, 3, 0, 0, 97]) = 0

Desarrollar paso a paso los siguientes casos:

pos_ult_apar(8, [8, 6, 8, 5])
pos_ult_apar(8, [3, 2])

Utilizar la siguiente operación ya definida: **esta**

24) maximo

Función que, dada una lista de enteros, devuelve el máximo de la lista. Si la lista es vacía se generará un mensaje de error: *maximo*.

Ejemplo:

maximo([5, 3, 8, 7, 8]) = 8

Desarrollar paso a paso el siguiente caso:

maximo([5, 3, 8, 7, 8])

Utilizar las siguientes operaciones ya definidas: **es_vacia, primero, resto**

25) unir

Función que dadas dos listas de tipo t, devuelve la lista que se obtiene uniendo las dos listas de tal forma que el primer elemento de la primera lista y el primero de la segunda lista quedan juntas.

Ejemplo (los *di* son elementos de tipo t):

unir([d2, d5, d4, d3], [**d8, d9**]) = [d3, d4, d5, d2, **d8, d9**]

Los elementos de la segunda lista quedan en el mismo orden, pero los elementos de la primera lista quedan en orden inverso.

Hacerlo de dos formas:

a) **Sin utilizar** operaciones definidas previamente
b) **Utilizando** las operaciones ya definidas **inversa** y ++

26) quitar

Función que dados un número entero y una lista de tipo t, devuelve la lista que se obtiene quitando de la lista inicial tantos elementos como indique el número. Si el número no está comprendido entre 0 y la longitud de la lista, se producirá un error. Si el número es justo 0, no se quitará ningún elemento.

Ejemplo:

quitar(3, [7, 6, 8, 5, 9]) = [5, 9]

Desarrollar paso a paso el siguiente caso:
quitar(3, [7, 6, 8, 5, 9])

Utilizar la siguiente operación ya definida: **longitud**

27) coger

Función que dados un número entero y una lista de tipo t, devuelve la lista que se obtiene cogiendo desde el principio de la lista inicial tantos elementos como indique el número. Si el número no está comprendido entre 0 y la longitud de la lista, se producirá un error. Si el número es justo 0, se devolverá la lista vacía.

Ejemplo:

coger(3, [7, 6, 8, 5, 9]) = [7, 6, 8]

Desarrollar paso a paso el siguiente caso:
coger(3, [7, 6, 8, 5, 9])

Utilizar la siguiente operación ya definida: **longitud**

28) colapsar

a) **Especificar ecuacionalmente** la función *colapsar* que dada una lista de enteros, devuelve la lista que se obtiene dejando sólo una copia cuando un elemento aparece repetido en posiciones contiguas, es decir, se eliminan las repeticiones contiguas. Si la lista inicial es vacía, la función ha de devolver una lista vacía (**utilizar** es_vacia y primero):

Ejemplo 1:

colapsar([0, 8, 8, 8, 4, 0, 0]) = [0, 8, 4, 0]

Ejemplo 2:

colapsar([3, 9, 9, 3]) = [3, 9, 3]

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el ejemplo colapsar([3, 9, 9, 3]), indicando en cada paso qué ecuación se ha utilizado.

29) propagar

a) **Especificar ecuacionalmente** la función *propagar* que, dada una lista de enteros, devuelve la lista que se obtiene propagando hacia la derecha los elementos pares por medio de la suma hasta encontrar un elemento impar o llegar al final de la lista. Si la lista es vacía la función ha de devolver la lista vacía (**utilizar** es_vacia, primero y resto):

Ejemplo:

propagar([4, 5, 2, 10, 8, 3, 6]) =
= [4, 5 + 4, 2, 10 + 2, 8 + 12, 3 + 20, 6]
= [4, 9, 2, 12, 20, 23, 6]

4 se propaga por ser par
9 no se propaga por ser impar
2 se propaga por ser par
12 se propaga por ser par
20 se propaga por ser par
23 no se propaga por ser impar
6 no se propaga por estar en la base

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el ejemplo propagar([8, 7, 2]), indicando en cada paso qué ecuación se ha utilizado.

30) arrollar

a) **Especificar ecuacionalmente** la función *arrollar* que, dada una lista de enteros, devuelve la lista que se obtiene empezando desde la izquierda y haciendo que un elemento arrolle a los que vienen a continuación hasta encontrar uno mayor. Cuando se encuentre uno mayor ese número mayor será el que empiece a arrollar a los que vienen a continuación y así hasta recorrer toda la lista. Si la lista inicial es vacía la función ha de devolver una lista vacía (**utilizar** es_vacia, primero y resto):

Ejemplo 1:

arrollar([4, 2, 3, 2, 8, 4, 9]) = [4, 4, 4, 4, 8, 8, 9]

Ejemplo 2:

arrollar([3, 9, 9, 3]) = [3, 9, 9,9]

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el ejemplo arrollar([3, 9, 9, 3]), indicando en cada paso qué ecuación se ha utilizado.

31) hundir

a) **Especificar ecuacionalmente** la función *hundir* que, dada una lista de enteros, devuelve la lista que se obtiene hundiendo el primer elemento hasta encontrar un valor que sea mayor o igual que él. Cuando se encuentre uno mayor o igual se termina el proceso, quedando los demás elementos igual. Si la lista inicial es vacía se ha de devolver una lista vacía y si la lista inicial tiene un solo elemento, se ha de devolver la misma lista (**utilizar** es_vacia, primero y resto):

Ejemplo 1:

hundir([**5**, 2, 4, 1, 8, 4, 3]) = [2, 4, 1, **5**, 8, 4, 3]

El primer elemento (el 5), se ha hundido (o desplazado hacia la derecha) hasta encontrar un valor mayor o igual que él (el 8).

Ejemplo 2:

hundir([**8**, 3, 2, 9]) = [3, 2, **8**, 9]

El primer elemento (el 8), se ha hundido (o desplazado hacia la derecha) hasta encontrar un valor mayor o igual que él (el 9).

Ejemplo 3:

hundir([**8**, 9, 1, 4]) = [**8**, 9, 1, 4]

El primer elemento (el 8), no se ha podido desplazar porque el segundo es ya mayor.

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el siguiente ejemplo, indicando en cada paso qué ecuación se ha utilizado:

hundir([**8**, 3, 2, 9])

32) borrar

a) **Especificar ecuacionalmente** la función *borrar* que dada una lista de enteros y un número entero que representa una posición de la lista empezando desde la derecha, devuelve la lista que se obtiene eliminando el elemento de la posición seleccionada. Si la lista es vacía se ha de generar un error y, siendo la lista no vacía, si la posición no es un número comprendido entre 1 y la longitud de la lista, también se ha de generar un error (**utilizar** la operación *longitud*):

Ejemplo 1:

borrar([5, 4, 1, 2, **4**, 8, 3], 3) = [5, 4, 1, 2, 8, 3]

Ejemplo 2:

borrar([1, 7, **5**, 8], 2) = [1, 7, 8]

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el siguiente ejemplo, indicando en cada paso qué ecuación se ha utilizado:

borrar([1, 7, **5**, 8], 2)

33) barrer (abril 2008 #1)

a) **Especificar ecuacionalmente** la función *barrer* que, dada una lista de enteros, devuelve la lista que se obtiene eliminando todos los elementos de la lista que son menores que el primero y que coloca dicho entero al final de la nueva lista. Si la lista es vacía ha de devolver un mensaje de error. Utilizar es_vacia, primero y resto:

Ejemplos:
barrer([5, 8, 8, 4, 9, 3]) = [8, 8, 9, 5]
barrer([5, 8, 5, 4, 9, 3]) = [8, 5, 9, 5]
barrer([5, 8, 8, 9, 6]) = [8, 8, 9, 6, 5]

La idea es que el primer elemento atraviesa toda la lista haciendo desaparecer los elementos que son menores que él y finalmente él mismo se queda en la última posición.

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el siguiente ejemplo indicando en cada paso qué ecuación se ha utilizado: barrer([5, 8, 4, 1, 9])

34) recorrer (abril 2008 #2)

a) **Especificar ecuacionalmente** la función *recorrer* que, dada una lista de enteros, devuelve la lista que se obtiene desplazando el primer elemento hacia la derecha hasta:

- el final si ese primer elemento no aparece en toda la lista y
- hasta que aparezca el propio número, en cuyo caso las dos copias han de ser sustituidas por 0.

Si la lista es vacía ha de devolver una lista vacía. Utilizar es_vacia, primero y resto:

Ejemplos:
recorrer([**5**, 8, 4, **5**, 9, 3]) = [8, 4, **0**, **0**, 9, 3]
recorrer([**5**, 8, 8, 4, 9, 3]) = [8, 8, 4, 9, 3, **5**]
recorrer([**5**, **5**, 5, 5]) = [0, 0, **5**, **5**]

La idea es que el primer elemento va atravesando la lista hasta encontrar un elemento igual y entonces los dos valores son sustituidos por 0. Si en ese recorrido el valor inicial no aparece, éste llegará hasta el final de la lista y se quedará ahí.

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso los siguientes ejemplos indicando en cada paso qué ecuación se ha utilizado:

recorrer([5, 8, 4])

recorrer([5, 5, 4])

35) dividir (junio 2008)

a) **Especificar ecuacionalmente** la función *dividir* que, dada una lista de enteros, devuelve una lista que tiene un elemento menos que el de entrada y se obtiene considerando los elementos que van a partir del segundo:

- Aquellos elementos divisibles por el primero (los que dan resto 0) son sustituidos por el resultado que se obtiene al dividirlos por el primero.
- Aquellos elementos no divisibles por el primero se mantienen.

Si la lista inicial es vacía, ha de devolver un mensaje de error. Si la lista no es vacía pero su primer elemento es 0, ha de devolver un mensaje de error. Si la lista contiene sólo un elemento, ha de devolver la lista vacía. **Utilizar** es_vacia, primero y resto:

Ejemplos:

dividir([**2**, 8, 6, 5, 9, 20]) = [4, 3, 5, 9, 10]	dividir([**3**, 15]) = [5]
dividir([**5**, 8, 8, 15, 9]) = [8, 8, 3, 9]	dividir([**3**, 14]) = [14]

La idea es que el primer elemento va atravesando la lista y dividiendo aquellos que son divisibles y manteniendo igual los que no son divisibles.

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el siguiente ejemplo indicando en cada paso qué ecuación se ha utilizado: dividir([5, 20, 6, 15])

36) superpar (septiembre 2008)

a) **Especificar ecuacionalmente** la función *superpar* que, dada una lista de enteros, devuelve la lista que se obtiene calculando la suma de todos los pares y dejándola al final de la nueva lista. Los elementos impares se mantienen.

Si la lista inicial es vacía ha de devolver la lista vacía y si la lista inicial sólo contiene un elemento, se ha de devolver la misma lista. **Utilizar** *es_vacia*, *primero* y *resto*:

Ejemplos:

superpar([**2**, **8**, 5, **14**, 9, **10**]) = [5, 9, **34**]	superpar([5, 3, 7, 1]) = [5, 3, 7, 1]

La idea es que los pares van desplazándose hacia la izquierda y al encontrar un nuevo número par, los dos son sustituidos por la suma de ellos, y ese nuevo número par es el que sigue desplazándose hacia la derecha. Los impares se mantienen.

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el siguiente ejemplo indicando en cada paso qué ecuación se ha utilizado: superpar([3, 10, 8, 9])

37) acumular (abril 2009 #1)

a) **Especificar ecuacionalmente** la función *acumular* que, dada una lista de enteros, devuelve la lista que contiene en cada posición la suma de los elementos que van desde la primera posición hasta esa posición. Si la lista es vacía se ha de devolver la lista vacía. Utilizar es_vacia, primero y resto.

Ejemplos:
acumular([10, 8, 15]) = [10, 18, 33]
acumular([10, 0, 8]) = [10, 10, 18]

La idea es que se va recorriendo la lista y calculando la suma acumulada hasta esa posición.

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el siguiente ejemplo indicando en cada paso qué ecuación se ha utilizado: **acumular([10, 8, 15])**

38) adelantar (abril 2009 #2)

a) **Especificar ecuacionalmente** la función *adelantar* que dadas una lista de booleanos y otra de enteros, devuelve la lista que se obtiene recorriendo las dos listas a la vez de tal forma que

- Si las dos listas no tienen la misma longitud se genera un mensaje de error.
- Si las dos listas son vacías se devolverá la lista vacía.
- Si las dos listas tienen un único elemento cada una, el resultado es la segunda lista tal cual.
- Cada vez que haya un True en la primera lista, el elemento correspondiente de la segunda lista adelanta una posición intercambiándose por el de la siguiente posición.
- Cuando hay un False en la primera lista, el elemento correspondiente se queda en la misma posición y se sigue recorriendo las listas.

Es conveniente darse cuenta de que un elemento puede adelantar varias posiciones por haber varios Trues seguidos pero un elemento que se mueve hacia atrás ya no adelantará nunca.

Utilizar las funciones *longitud*, *es_vacia*, *primero* y *resto*.

Ejemplos:
- adelantar([False, **True**, **True**, False, **True**], [8, 3, 9, 5, 2]) = [8, 9, 5, 3, 2]

  En este caso el 8 se queda en la primera posición porque hay un False. El 3 primero adelanta una posición intercambiándose con el 9 por haber un True en la segunda posición y una vez situado en la tercera posición el 3 vuelve a adelantar otra posición intercambiándose con el 5 por haber otro True en la tercera posición. Como en la cuarta

posición hay un False el 3 no puede seguir adelantando. El 2 se queda donde estaba porque, aunque haya un True en la última posición, ya no hay sitio para moverse hacia la derecha.

- adelantar([**True**, **True**, **True**, False, False], [8, 3, 9, 5, 2]) = [3, 9, 5, 8, 2]

  En este caso el 8 adelanta una posición intercambiándose con el 3 porque en la primera posición hay un True. Una vez situado en la segunda posición, el 8 vuelve a adelantar una posición, intercambiándose con el 9, porque en la segunda posición hay otro True. Una vez situado en la tercera posición, el 8 vuelve a adelantar una posición, intercambiándose con el 5, porque en la tercera posición hay otro True. Como en la cuarta posición hay un False el 8 ya no puede seguir adelantando. El 2 se queda donde estaba porque en la quinta posición hay un False.

- adelantar([**True**, False, **True**, False, **True**], [8, 3, 9, 5, 2]) = [3, 8, 5, 9, 2]
- adelantar([**True**, False, **True**, **True**, **True**], [8, 3, 9, 5, 2]) = [3, 8, 5, 2, 9]

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el siguiente ejemplo indicando en cada paso qué ecuación se ha utilizado:

adelantar([**True**, False, **True**, **True**, **True**], [8, 3, 9, 5, 2])

39) elimparpos (junio 2009)

a) **Especificar ecuacionalmente** la función *elimparpos* que dados una lista de enteros y un entero que representa una posición dentro de la lista, devuelve la lista que se obtiene al eliminar el elemento de la lista que ocupa dicha posición en caso de que ese elemento sea par. Si el elemento que ocupa la posición indicada no es par, no se elimina. Si la lista dada es vacía o, no siendo vacía la lista, la posición indicada no está entre 1 y la longitud de la lista, la función ha de devolver un mensaje de error.

Utilizar la función *longitud*.

Ejemplos:

- elimparpos([8, 5, **9**, 7, 10, 4], **3**) = [8, 5, **9**, 7, 10, 4]

  En este caso se devuelve la misma lista porque el elemento que ocupa la posición 3 no es par y por tanto no hay que eliminarlo.

- elimparpos([8, 5, **16**, 7, 10, 4], **3**) = [8, 5, 7, 10, 4]

  En este caso se ha devuelto la lista que se obtiene eliminando el elemento de la posición 3 porque es par.

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el siguiente ejemplo indicando en cada paso qué ecuación se ha utilizado:

elimparpos([8, 5, 16, 7, 10, 4], 3)

40) ult_primo (septiembre 2009)

a) **Especificar ecuacionalmente** la función *ult_primo* que, dada una lista de enteros, devuelve el último primo de la lista, es decir, el que está más a la derecha. Si la lista es vacía o si no hay ningún primo en la lista, la función ha de devolver el valor -1.

Utilizar las funciones *primero*, *resto* y *es_vacia*.

Además, hay que utilizar la función auxiliar *es_primo*, que dado un entero devuelve true si es primo y false en caso contrario, es decir, dado un entero devuelve true si el entero es positivo y tiene justo dos divisores. Se considera que la función *es_primo* ya está definida, no hay que definirla, sólo utilizarla

Ejemplos:

- ult_primo([8, 11, -7, 9, 3, 6]) = 3

  En este caso la lista contiene dos primos, el 11 y el 3, y la función devuelve el último, es decir, el 3.

- ult_primo([8, -7, 16, 4, 10]) = -1

  En este caso la lista no contiene ningún número primo y por tanto devuelve -1.

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el siguiente ejemplo indicando en cada paso qué ecuación se ha utilizado:

ult_primo([8, 11, -7, 9, 3, 6])

41) sublongparponer (Abril 2010 #1)

a) Dar la especificación ecuacional de la función "sublongparponer" que dada una lista de tipo Int, devuelve una nueva lista generada de tal forma que para que cada sublista formada por elementos iguales tenga longitud par, se añade o pone un elemento igual al final de cada sublista de longitud impar. Si la lista inicial es vacía, la función devolverá la lista vacía. Utilizar las funciones *es_vacia* (que decide si una lista es vacía o no), *primero* (que devuelve el primer elemento de una lista) y *resto* (que devuelve la lista que queda tras eliminar el primer elemento):

Ejemplos:
sublongparponer([10, 10, 10, 8, 8, 15, 8]) = [10, 10, 10, **10**, 8, 8, 15, **15**, 8, **8**]
sublongparponer([15]) = [15, **15**]
sublongparponer([15, 10, 15, 10]) = [15, **15**, 10, **10**, 15, **15**, 10, **10**]

- En el primer ejemplo hay cuatro sublistas formadas por elementos iguales. En la primera sublista el valor 10 aparece 3 veces y para que la longitud de esa sublista sea par, se añade otro 10. En la tercera y cuarta sublista también se ha añadido un elemento para obtener sublistas de longitud par.
- En el segundo ejemplo hay una única sublista y está formada por un único 15. Como la longitud de esa sublista es impar se ha añadido otro 15.
- En el tercer ejemplo hay cuatro sublistas. Como cada una de ellas tiene longitud impar, en cada caso se ha añadido un elemento para tener sublistas de longitud par.

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el siguiente ejemplo indicando en cada paso qué ecuación se ha utilizado:

sublongparponer([10, 10, 10, 8, 8, 15, 8])

42) mayor_de_cada_par (Abril 2010 #2)

a) Dar la especificación ecuacional de la función "mayor_de_cada_par" que dada una lista de tipo Int, devuelve una nueva lista generada de tal forma que para cada par de elementos se elimina el menor y se duplica el mayor. Los pares no se mezclan, es decir, el primer par lo forman los elementos de las posiciones 1 y 2, el segundo par lo forman los elementos de las posiciones 3 y 4, etc. Si la lista inicial es vacía, la función devolverá la lista vacía. Si la lista inicial tiene longitud impar, la función devolverá un mensaje de error. Utilizar las funciones *longitud* (que devuelve el número de elementos de una lista), *primero* (que devuelve el primer elemento de una lista) y *resto* (que devuelve la lista que queda tras eliminar el primer elemento):

Ejemplos:

mayor_de_cada_par([10, 8, 5, 7, 7, 20, 5, 5]) = [10, 10, 7, 7, 20, 20, 5, 5]

En la lista del ejemplo hay cuatro pares. El primer par está formado por 10 y 8 y como el mayor es 10, el primer par de la nueva lista está formado por dos dieces. El segundo par está formado por 5 y 7 y como el mayor es 7, el segundo par de la nueva lista está formado por dos sietes. En el tercer par el mayor es 20 y por ello el tercer par de la nueva lista está formado por dos veintes. En el cuarto par el mayor es 5 y por ello el cuarto par de la nueva lista está formado por dos cincos.

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el siguiente ejemplo indicando en cada paso qué ecuación se ha utilizado:

mayor_de_cada_par([10, 8, 5, 7, 7, 20, 5, 5])

43) colocar (Junio 2010)

a) **Especificar ecuacionalmente** la función *colocar* que, dadas dos listas de enteros, devuelve la lista que contiene todos los elementos de la primera lista pero habiendo colocado además un elemento de la segunda lista (por orden de aparición, de izquierda a derecha) cada vez que en la primera lista haya un 0.

Si la primera lista es vacía se devuelve la lista vacía.

Siendo la primera lista no vacía, si el número de ceros de la primera lista es mayor que el número de elementos de la segunda lista, se mostrará un mensaje de error.

Utilizar las siguientes funciones:

- *nveces*, que, dados un elemento y una lista, devuelve el número de veces que aparece ese elemento en la lista.
- *longitud*, que dada una lista devuelve el número de elementos de la lista.
- *primero*, que, dada una lista, devuelve el primer elemento de la lista.
- *resto*, que, dada una lista, devuelve la lista que se obtiene quitando el primer elemento de la lista.

Ejemplos:

- colocar([8, **0**, **0**, 7, **0**, 6], [**3**, **20**, **12**, 45, 28]) = [8, **0**, **3**, **0**, **20**, 7, **0**, **12**, 6]

  Como en la primera lista hay tres ceros, en la nueva lista se han colocado los tres primeros elementos de la segunda lista, cada uno detrás de un 0.

- colocar([8, **0**, **0**, 7, **0**, 6], [**3**, **20**])

  En este caso la función mostraría un mensaje de error por no haber suficientes elementos en la segunda lista, ya que en la primera lista hay más ceros que elementos en la segunda.

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el siguiente ejemplo indicando en cada paso qué ecuación se ha utilizado:

colocar([8, 0, 0, 7], [3, 20, 12, 28])

44) sublongparquitar (Septiembre 2010)

a) Dar la especificación ecuacional de la función "sublongparquitar" que dada una lista de tipo Int, devuelve una nueva lista generada de tal forma que para que cada sublista formada por elementos iguales tenga longitud par, se elimina o quita un elemento al final de cada sublista de longitud impar. Si la lista inicial es vacía, la función devolverá la lista vacía. Utilizar las funciones *es_vacia* (que decide si una lista es vacía o no), *primero* (que devuelve el primer elemento de una lista) y *resto* (que devuelve la lista que queda tras eliminar el primer elemento):

Ejemplos:

sublongparquitar([10, 10, **10**, 8, 8, **15**, **8**]) = [10, 10, 8, 8]
sublongparquitar([**15**]) = []
sublongparquitar([**15**, **10**, **15**, **10**]) = []
sublongparquitar([10, 10, 10, 10, 8, 8]) = [10, 10, 10, 10, 8, 8]

- En el primer ejemplo hay cuatro sublistas formadas por elementos iguales. En la primera sublista el valor 10 aparece 3 veces y para que la longitud de esa sublista sea par, se quita un 10. En la tercera y cuarta sublista también se ha de quitar un elemento para obtener sublistas de longitud par. Como esas dos sublistas sólo contienen un elemento, desaparecen.
- En el segundo ejemplo hay una única sublista y está formada por un único 15. Como la longitud de esa sublista es impar se ha quitado el 15 y ha quedado la lista vacía.
- En el tercer ejemplo hay cuatro sublistas. Como cada una de ellas tiene longitud impar, en cada caso se ha quitado un elemento para tener sublistas de longitud par. Como todas las sublistas son de longitud 1, ha quedado una lista vacía.
- En el cuarto ejemplo hay dos sublistas formadas por elementos iguales. Como las dos sublistas tienen longitud par, no se ha quitado ningún elemento y al final ha quedado la lista inicial.

b) Una vez dadas las ecuaciones, **desarrollar** paso a paso el siguiente ejemplo indicando en cada paso qué ecuación se ha utilizado:

sublongparquitar([10, 10, 10, 8, 8, 15, 8])

B) Pruebas por inducción para listas
Demostrar los siguientes **teoremas** inductivos:

1) longitud(s) = longitud(inversa(s))

Probar por inducción que para toda lista s se cumple lo siguiente:

longitud(s) = longitud(inversa(s))

La inducción se ha de realizar sobre s considerando el caso básico s = [] y el caso general s = x:r. La hipótesis de la inducción consistirá en suponer que para r se cumple la propiedad. Utilizar la versión 7.a (con ++) para la función *inversa*. Hay que utilizar la propiedad que dice que *longitud(u ++ v) = longitud(u) + longitud(v)* para cualquier par de listas u y v.

2) longitud(s) ≥ longitud(eliminar(x, s))

Probar por inducción que para cualquier elemento x y cualquier lista s se cumple lo siguiente:

longitud(s) ≥ longitud(eliminar(x, s))

La inducción se ha de realizar sobre s considerando el caso básico s = [] y el caso general s = z:r. La hipótesis de la inducción consistirá en suponer que para r se cumple la propiedad.

3) s ++ [ ] = s

Probar que para cualquier lista s se cumple lo siguiente:

s ++ [] = s

La inducción se ha de realizar sobre s considerando el caso básico s = [] y el caso general s = x:r. La hipótesis de la inducción consistirá en suponer que para r se cumple la propiedad.

4) inversa(s ++ r) = inversa(r) ++ inversa(s)

Probar que para dos listas cualesquiera s y r se cumple lo siguiente:

inversa(s ++ r) = inversa(r) ++ inversa(s)

La inducción se ha de realizar sobre s considerando el caso básico s = [] y el caso general s = x:w. La hipótesis de la inducción consistirá en suponer que para w y r se cumple la propiedad.

Para probar que la propiedad se cumple en el caso básico es necesario utilizar o considerar la propiedad probada en el ejercicio 3 (para cualquier s se cumple s ++ [] = s).

5) nveces(z, s ++ r) = nveces(z, s) + nveces(z, r)

Probar por inducción que para cualquier elemento z y dos listas cualesquiera s y r se cumple lo siguiente:

nveces(z, s ++ r) = nveces(z, s) + nveces(z, r)

La inducción se ha de realizar sobre s considerando el caso básico s = [] y el caso general s = x:w. La hipótesis de la inducción consistirá en suponer que para w y r se cumple la propiedad.

6) nveces(x, eliminar(x, s)) = 0

Probar por inducción que para cualquier elemento x y cualquier lista s, se cumple la siguiente propiedad:

nveces(x, eliminar(x, s)) = 0

La inducción debe realizarse sobre s considerando el caso básico s = [] y el caso inductivo s = z:r, donde la hipótesis de la inducción consistirá en suponer que x y r cumplen la propiedad.

7) eliminar(x, s ++ r) = eliminar(x, s) ++ eliminar(x, r)

Probar por inducción que para cualquier elemento x (de tipo t) y cualquier par de listas s y r (de tipo [t]), se cumple la siguiente propiedad:

eliminar(x, s ++ r) = eliminar(x, s) ++ eliminar(x, r)

La inducción debe realizarse sobre s considerando el caso básico s = [] y el caso inductivo s = z:w, donde la hipótesis de la inducción consistirá en suponer que x, w y r cumplen la propiedad.

8) longitud(eliminar(x, s ++ r)) = longitud(eliminar(x, s)) + longitud(eliminar(x, r))

Probar por inducción que para cualquier elemento x (de tipo t) y cualquier par de listas s y r (de tipo [t]), se cumple la siguiente propiedad:

longitud(eliminar(x, s ++ r)) = longitud(eliminar(x, s)) + longitud(eliminar(x, r))

La inducción debe realizarse sobre s considerando el caso básico s = [] y el caso inductivo s = z:w, donde la hipótesis de la inducción consistirá en suponer que x, w y r cumplen la propiedad.

9) eliminar(x, eliminar(x, s)) ++ r = eliminar(x, s) ++ r

Probar por inducción que para cualquier elemento x (de tipo t) y cualquier par de listas s y r (de tipo [t]), se cumple la siguiente propiedad:

eliminar(x, eliminar(x, s)) ++ r = eliminar(x, s) ++ r

La inducción debe realizarse sobre s considerando el caso básico s = [] y el caso inductivo s = z:w, donde la hipótesis de la inducción consistirá en suponer que x, w y r cumplen la propiedad.

10) sumar(incr (s)) = sumar(s) + longitud(s)

Probar por inducción que para cualquier lista s (de tipo [Int]), se cumple la siguiente propiedad:

sumar(incr(s)) = sumar(s) + longitud(s)

La inducción debe realizarse sobre s considerando el caso básico s = [] y el caso inductivo s = x:r, donde la hipótesis de la inducción consistirá en suponer que r cumple la propiedad.

11) eliminar_pares(s ++ r) = eliminar_pares(s) ++ eliminar_pares(r)

Probar por inducción que para dos listas cualesquiera s y r se cumple lo siguiente:

eliminar_pares(s ++ r) = eliminar_pares(s) ++ eliminar_pares(r)

La inducción se ha de realizar sobre s considerando el caso básico s = [] y el caso general s = x:w. La hipótesis de la inducción consistirá en suponer que para w y r se cumple la propiedad.

12) longitud(eliminar_pos_pares(s)) = longitud(s) `div` 2 cuando la longitud de s es par

Probar por inducción que para cualquier lista s cuya longitud sea par, se cumple la siguiente propiedad:

longitud(eliminar_pos_pares(s)) = longitud(s) `div` 2

siendo `div` la división entera.

La inducción debe realizarse sobre s considerando el caso básico s = [] y el caso inductivo s = x:r, pero recordando que s tiene un número par de elementos y que por tanto r no es vacía (tiene al menos un elemento). La hipótesis de la inducción consistirá en suponer que resto(r) (cuya longitud también será par) cumple la propiedad.

13) longitud(eliminar_pos_pares(s)) = (longitud(s) `div` 2) + 1 cuando impar(longitud(s))

Probar por inducción que para cualquier secuencia s cuya longitud sea impar, se cumple la siguiente propiedad:

longitud(eliminar_pos_pares(s)) = (longitud(s) `div` 2) + 1

siendo `div` la división entera.

La inducción debe realizarse sobre s considerando el caso básico s = x:[] y el caso inductivo s = x:(y:r), pero teniendo en cuenta que s tiene un número impar de elementos y que el número de elementos de s es por lo menos 3. La hipótesis de la inducción consistirá en suponer que r (cuya longitud también será impar) cumple la propiedad. Es necesario tener en cuenta que toda lista no vacía r cumple la siguiente propiedad:

longitud(r) = 1 + longitud(resto(r))

14) *eliminar_pos_pares(s ++ r) = eliminar_pos_pares(s) ++ eliminar_pos_pares(r) cuando par(longitud(s))*

Probar por inducción que para cualquier par de listas s y r, siendo la longitud de s par, se cumple la siguiente propiedad:

eliminar_pos_pares(s ++ r) = eliminar_pos_pares(s) ++ eliminar_pos_pares(r)

La inducción debe realizarse sobre s considerando el caso básico s = [] y el caso inductivo s = x:(y:w), donde la hipótesis de la inducción consistirá en suponer que w y r (donde la longitud de w también será par) cumplen la propiedad.

15) *eliminar_pos_pares(s ++ r) ≠ eliminar_pos_pares(s) ++ eliminar_pos_pares(r) cuando impar(longitud(s))*

Comprobar que cuando la longitud de s no es par la propiedad del ejercicio anterior no se cumple. Calcular para ello las dos siguientes expresiones (no es necesario dar todos los pasos siguiendo las ecuaciones):

- eliminar_pos_pares([7, 4, 8] ++ [5, 2])
- eliminar_pos_pares([7, 4, 8]) ++ eliminar_pos_pares([5, 2])

Nota: Para probar que todas las listas cumplen una determinada propiedad hay que utilizar una técnica como la de la inducción. En cambio, para probar que no todas las listas cumplen una determinada propiedad basta con dar un ejemplo en el que se vea que la propiedad no se cumple.

16) *nveces(x, s) = nveces(x + 1, incr(s)) (abril 2008 #1)*

a) **Especificar ecuacionalmente** la función nveces que, dados un elemento de tipo t y una lista de tipo t devuelve el número de apariciones del elemento en la lista:

nveces:: (t, [t]) → Int

Ejemplos:
nveces(5, [2, 6]) = 0
nveces(8, [3, 4, 8, 9, 8]) = 2

b) **Especificar ecuacionalmente** la función incrementar que, dada una lista de enteros devuelve la lista que se obtiene incrementando en 1 todos los elementos de la lista. Si la lista es vacía devuelve una lista vacia:

incr:: ([t]) → [t]

Ejemplos:
incr([2, 6]) = [3, 7]
incr([3, 4, 8, 9, 8]) = [4, 5, 9, 10, 9]

c) **Probar por inducción** que para cualquier lista s, se cumple la siguiente propiedad:

nveces(x, s) = nveces(x + 1, incr(s))

La inducción debe realizarse sobre s considerando el caso básico s = [] y el caso inductivo s = z:r, donde la hipótesis de la inducción consistirá en suponer que x y r cumplen la propiedad.

17) *sumar(s) = ultimo(s) + sumar(sin_ultimo(s)) (abril 2008 #2)*

a) **Especificar ecuacionalmente** la función *sumar* que, dada una lista de enteros devuelve la suma de todos los elementos de la lista. Si la lista es vacía devuelve 0:

sumar:: ([Int]) → Int

Ejemplos:
sumar([5, -1, 8]) = 12
sumar([8, 3, 7, 8]) = 26

b) **Especificar ecuacionalmente** la función *ultimo* que, dada una lista de tipo t devuelve el último elemento de la lista. Si la lista es vacía devuelve error. Se ha de utilizar la operación es_vacia:

ultimo:: ([t]) → t

Ejemplos:
ultimo([5, 2, 6]) = 6
ultimo([3, 8, 4, 4, 9, 8]) = 8

c) **Especificar ecuacionalmente** la función *sin_ultimo* que, dada una lista de tipo t devuelve la lista que se obtiene eliminando el último elemento de la lista. Si la lista es vacía devuelve error. Se ha de utilizar la operación es_vacia:

sin_ultimo:: ([t]) → [t]

Ejemplos:
sin_ultimo([5, 2, 6]) = [5, 2]
sin_ultimo([3, 8, 4, 4, 9, 8]) = [3, 8, 4, 4, 9]

d) **Probar por inducción** que para cualquier lista no vacía s (de tipo [Int]), se cumple la siguiente propiedad:

sumar(s) = ultimo(s) + sumar(sin_ultimo(s))

La inducción debe realizarse sobre s considerando el caso básico s = x:[] y el caso inductivo s = z:r, siendo r una lista no vacía y donde la hipótesis de la inducción consistirá en suponer que r cumple la propiedad.

18) *esta(x, s) = esta(x + 1, incr(s)) (junio 2008)*

a) **Especificar ecuacionalmente** la función esta que, dados un elemento de tipo t y una lista de tipo t devuelve True si el elemento está en la lista y False en caso contrario:

esta:: (t, [t]) → Bool

Ejemplos:

esta(5, [2, 6]) = False	esta(8, [3, 4, 8, 9, 8]) = True

b) **Especificar ecuacionalmente** la función incr que, dada una lista de enteros devuelve la lista que se obtiene incrementando en 1 todos los elementos de la lista. Si la lista es vacía devuelve una lista vacía:

incr:: ([t]) → [t]

Ejemplos:

incr([2, 6]) = [3, 7]	incr([3, 4, 8, 9, 8]) = [4, 5, 9, 10, 9]

c) **Probar por inducción** que para cualquier lista s, se cumple la siguiente propiedad:

esta(x, s) = esta(x + 1, incr(s))

La inducción debe realizarse sobre s considerando el caso básico s = [] y el caso inductivo s = z:r, donde la hipótesis de la inducción consistirá en suponer que x y r cumplen la propiedad.

19) *longitud(eliminar(e, s)) = longitud(s) – nveces(e, s) (septiembre 2008)*

a) **Especificar ecuacionalmente** la función longitud que, dada una lista de tipo t, devuelve el número de elementos de la lista:

longitud:: ([t]) → Int

Ejemplos:

longitud([d5, d2, d6]) = 3	longitud([d8, d3, d4, d8, d9, d8]) = 6

b) **Especificar ecuacionalmente** la función eliminar que, dados un elemento de tipo t y una lista de tipo t, devuelve la lista que se obtiene eliminando todas las apariciones de dicho elemento en la lista:

eliminar:: (t, [t]) → [t]

Ejemplos:

eliminar(d8, [d5, d2, d6]) = [d5, d2, d6]	eliminar(d8, [**d8**, d3, d4, **d8**, d9, **d8**]) = [d3, d4, d9]

c) **Especificar ecuacionalmente** la función nveces que, dados un elemento de tipo t y una lista de tipo t, devuelve el número de apariciones del elemento en la lista:

nveces:: (t, [t]) → Int

Ejemplos:

nveces(d5, [d2, d6]) = 0	nveces(d8, [d3, d4, d8, d9, d8]) = 2

d) **Probar por inducción** que para cualquier lista s, se cumple la siguiente propiedad:

longitud(eliminar(e, s)) = longitud(s) – nveces(e, s)

La inducción debe realizarse sobre s considerando el caso básico s = [] y el caso inductivo s = x:r, donde la hipótesis de la inducción consistirá en suponer que para el elemento e y la lista r se cumple la propiedad.

20) *inversa(inversa(s)) = s (abril 2009 #1)*

a) **Especificar ecuacionalmente** la función ++ que, dadas dos listas de tipo t devuelve la lista que se obtiene concatenándolas.

++:: ([t], [t]) → [t]

Ejemplos: [1, 7] ++ [8, 5, 9] = [1, 7, 8, 5, 9]
[] ++ [8, 5, 9] = [8, 5, 9]

b) **Especificar ecuacionalmente** la función "inversa" que, dada una lista de tipo t devuelve la lista que se obtiene poniendo los elementos en orden inverso. Si la lista es vacía devuelve una lista vacia. Utilizar la operación de concatenación ++:

inversa:: ([t]) → [t]

Ejemplos:
inversa([2, 6]) = [6, 2]
inversa([3, 4, 8, 9, 8]) = [8, 9, 8, 4, 3]

c) **Probar por inducción** que para cualquier lista s, se cumple la siguiente propiedad:

inversa(inversa(s)) = s

La inducción debe realizarse sobre s considerando el caso básico s = [] y el caso inductivo s = x:r, donde la hipótesis de la inducción consistirá en suponer que r cumple la propiedad que se está probando. Además, hay que utilizar la siguiente propiedad sobre listas:

- Para cualquier par de listas v y w se cumple que

inversa(v ++ w) = inversa(w) ++ inversa(v).

21) *longitud(resto(s)) = longitud(sin_ultimo(s)) (abril 2009 #2)*

a) **Especificar ecuacionalmente** la función "*longitud*" que, dada una lista de tipo t devuelve el número de elementos de la lista. Si la lista es vacía devuelve 0.

longitud :: ([t]) → Int

Ejemplo: longitud[9, 7, 8, 8, 1] = 5

b) **Especificar ecuacionalmente** la función "*sin_ultimo*" que, dada una lista de tipo t devuelve la lista que se obtiene eliminando el último elemento de la lista (el que está en el extremo derecho). Si la lista es vacía devuelve un mensaje de error. Si la lista sólo contiene un elemento, devuelve la lista vacía:

sin_ultimo :: ([t]) → [t]

Ejemplo:
sin_ultimo([8, 6, 7, 3]) = [8, 6, 7]

c) **Especificar ecuacionalmente** la función "*resto*" que, dada una lista de tipo t devuelve la lista que se obtiene eliminando el primer elemento de la lista (el que está en el extremo izquierdo). Si la lista inicial es vacía devuelve un mensaje de error:

resto :: ([t]) → [t]

Ejemplo:
resto([8, 6, 7, 3]) = [6, 7, 3]

d) **Probar por inducción** que para cualquier lista no vacía s, se cumple la siguiente propiedad:

longitud(resto(s)) = longitud(sin_ultimo(s))

La inducción debe realizarse sobre s considerando el caso básico **s = z:[]** y el caso inductivo **s = z:r**, **donde r es una lista no vacía**. La hipótesis de la inducción consistirá en suponer que la lista no vacía **r cumple la propiedad** que se está probando. Además, hay que utilizar la siguiente propiedad sobre listas:

- Para cualquier lista no vacía w se cumple que

longitud(w) = 1 + longitud(resto(w)) **(Prop)**.

22) *nveces(x, s) ≥ nveces(x, sin_ultimo(s)) (junio 2009)*

a) **Especificar ecuacionalmente** la función "*nveces*" que, dados un elemento de tipo t y una lista de tipo t devuelve el número de veces que aparece dicho elemento en la lista:

nveces:: (t, [t]) → Int

Ejemplos: nveces(5, [8, 5, 6, 5]) = 2
nveces(8, []) = 0
nveces(8, [7, 1, 2]) = 0

b) **Especificar ecuacionalmente** la función "*sin_ultimo*" que, dada una lista de tipo t devuelve la lista que se obtiene eliminando el último elemento de la lista (el que está en el extremo derecho). Si la lista es vacía devuelve un mensaje de error. Si la lista solo contiene un elemento, devuelve la lista vacía:

sin_ultimo :: ([t]) → [t]

Ejemplo:
sin_ultimo([8, 6, 7, 3]) = [8, 6, 7]

c) **Probar por inducción** que para cualquier elemento x y cualquier lista no vacía s (siendo x y los elementos de s del mismo tipo), se cumple la siguiente propiedad:

$$nveces(x, s) \geq nveces(x, sin\_ultimo(s))$$

La inducción debe realizarse sobre s considerando el caso básico **s = z:[]** y el caso inductivo **s = z:r**, **donde r es una lista no vacía**. La hipótesis de la inducción consistirá en suponer que el elemento x y la lista no vacía **r cumplen la propiedad** que se está probando.

23) *resto(inversa(s)) = inversa(sin_ultimo(s)) (septiembre 2009)*

a) **Especificar ecuacionalmente** la función ++ que, dadas dos listas de tipo t devuelve la lista que se obtiene concatenándolas

++:: ([t], [t]) → [t]

Ejemplos: [1, 7] ++ [8, 5, 9] = [1, 7, 8, 5, 9]
[] ++ [8, 5, 9] = [8, 5, 9]

b) **Especificar ecuacionalmente** la función "inversa" que, dada una lista de tipo t devuelve la lista que se obtiene poniendo los elementos en orden inverso. Si la lista es vacía devuelve una lista vacia. Utilizar la operación de concatenación ++

inversa:: ([t]) → [t]

Ejemplos: inversa([2, 6]) = [6, 2]
inversa([3, 4, 8, 9, 8]) = [8, 9, 8, 4, 3]

c) **Especificar ecuacionalmente** la función "*resto*" que, dada una lista de tipo t devuelve la lista que se obtiene eliminando el primer elemento de la lista (el que está en el extremo izquierdo). Si la lista inicial es vacía devuelve un mensaje de error:

resto :: ([t]) → [t]

Ejemplo:
resto([8, 6, 7, 3]) = [6, 7, 3]

d) **Especificar ecuacionalmente** la función "*sin_ultimo*" que, dada una lista de tipo t devuelve la lista que se obtiene eliminando el último elemento de la lista (el que está en el extremo derecho). Si la lista es vacía devuelve un mensaje de error. Si la lista sólo contiene un elemento, devuelve la lista vacía:

sin_ultimo :: ([t]) → [t]

Ejemplo:
sin_ultimo([8, 6, 7, 3]) = [8, 6, 7]

e) **Probar por inducción** que para cualquier lista no vacía s, se cumple la siguiente propiedad:

resto(inversa(s)) = inversa(sin_ultimo(s))

La inducción debe realizarse sobre s considerando el caso básico **s = z:[]** y el caso inductivo **s = z:r**, **donde r es una lista no vacía**. La hipótesis de la

inducción consistirá en suponer que la lista no vacía **r cumple la propiedad** que se está probando. Además, hay que utilizar las siguientes propiedades sobre listas:

- Para cualquier lista no vacía w y cualquier lista v se cumple que

  resto(w ++ v) = resto(w) ++ v **(Prop1)**

- Para cualquier lista w se cumple que

  longitud(w) = longitud(inversa(w)) **(Prop2)**

24) *sumar(s) = sumar(inversa(s)) (Abril 2010 #1)*

a) **Especificar ecuacionalmente** la función ++ que, dadas dos listas de tipo t devuelve la lista que se obtiene al juntarlas

++:: ([t], [t]) → [t]

Ejemplos: [1, 7] ++ [8, 5, 9] = [1, 7, 8, 5, 9]
[] ++ [8, 5, 9] = [8, 5, 9]

b) **Especificar ecuacionalmente** la función "sumar" que dada una lista de enteros devuelve la suma de los elementos que conforman la lista:

sumar:: ([Int]) → Int

Ejemplos: sumar([4, 6, 5]) = 15

c) **Especificar ecuacionalmente** la función "inversa" que dada una lista de tipo t devuelve la lista que se obtiene colocando los elementos en orden inverso. Utilizar la operación de concatenación ++:

inversa:: ([t]) → [t]

Ejemplos: inversa([2, 6]) = [6, 2]
inversa([3, 4, 8, 9, 8]) = [8, 9, 8, 4, 3]

d) **Probar por inducción** que para cualquier lista s de tipo Int, se cumple la siguiente propiedad:

sumar(s) = sumar(inversa(s))

La inducción debe realizarse sobre s considerando el caso básico s = [] y el caso inductivo s = x:r. La hipótesis de la inducción consistirá en suponer que r cumple la propiedad. Además, habrá que utilizar la siguiente propiedad sobre listas:

- Para cualquier par de listas v y w que sean de tipo Int se cumple la siguiente propiedad

sumar(v ++ w) = sumar(v) + sumar(w) **(Prop)**

25) *incr(inversa(s)) = inversa(incr(s)) (Abril 2010 #2)*

a) **Especificar ecuacionalmente** la función ++ que, dadas dos listas de tipo t devuelve la lista que se obtiene al juntarlas

++:: ([t], [t]) → [t]

Ejemplos: [1, 7] ++ [8, 5, 9] = [1, 7, 8, 5, 9]
[] ++ [8, 5, 9] = [8, 5, 9]

b) **Especificar ecuacionalmente** la función "incr" que dada una lista de enteros devuelve la lista que se obtiene sumando un 1 a cada elemento de la lista. Si la lista es vacía, devuelve la lista vacía:

incr:: ([Int]) → [Int]

Ejemplos: incr([4, 6, 5]) = [5, 7, 6]

c) **Especificar ecuacionalmente** la función "inversa" que dada una lista de tipo t devuelve la lista que se obtiene colocando los elementos en orden inverso. Utilizar la operación de concatenación ++:

inversa:: ([t]) → [t]

Ejemplos: inversa([2, 6]) = [6, 2]
inversa([3, 4, 8, 9, 8]) = [8, 9, 8, 4, 3]

d) **Probar por inducción** que para cualquier lista s de tipo Int, se cumple la siguiente propiedad:

incr(inversa(s)) = inversa(incr(s))

La inducción debe realizarse sobre s considerando el caso básico s = [] y el caso inductivo s = x:r. La hipótesis de la inducción consistirá en suponer que r cumple la propiedad. Además, habrá que utilizar la siguiente propiedad sobre listas:

- Para cualquier par de listas v y w que sean de tipo Int se cumple la siguiente propiedad:

incr(v ++ w) = incr(v) ++ incr(w) **(Prop)**

26) *nveces(x, inversa(s)) = nveces(x, s) (Junio 2010)*

a) **Especificar ecuacionalmente** la función ++ que, dadas dos listas de tipo t devuelve la lista que se obtiene al juntarlas

$$++:: ([t], [t]) \rightarrow [t]$$

Ejemplos: [1, 7] ++ [8, 5, 9] = [1, 7, 8, 5, 9]
[] ++ [8, 5, 9] = [8, 5, 9]

b) **Especificar ecuacionalmente** la función "*nveces*" que, dados un elemento de tipo t y una lista de tipo t devuelve el número de veces que aparece dicho elemento en la lista:

nveces :: (t, [t]) → Int

Ejemplos:
nveces(d5, [d8, d5, d6, d5]) = 2
nveces(d8, []) = 0
nveces(d8, [d7, d1, d2]) = 0

c) **Especificar ecuacionalmente** la función "inversa" que dada una lista de tipo t devuelve la lista que se obtiene colocando los elementos en orden inverso. Al dar las ecuaciones hay que utilizar el operador ++ que sirve para concatenar dos listas:

$$inversa:: ([t]) \rightarrow [t]$$

Ejemplos: inversa([2, 6]) = [6, 2]
inversa([3, 4, 8, 9, 8]) = [8, 9, 8, 4, 3]

d) **Probar por inducción** que para cualquier elemento x y cualquier lista s (siendo x y los elementos de s del mismo tipo), se cumple la siguiente propiedad:

nveces(x, inversa(s)) = nveces(x, s)

La inducción debe realizarse sobre s considerando el caso básico **s = []** y el caso inductivo **s = z:r**. La hipótesis de la inducción consistirá en suponer que el elemento x y la lista r cumplen la propiedad que se está probando. Además, habrá que utilizar la siguiente propiedad Prop que indica que para cualquier elemento h y dos listas cualquiera v y w se cumple lo siguiente:

nveces(h, v ++ w) = nveces(h, v) + nveces(h, w) **(Prop)**

27) *incr(s ++ r) = incr(s) ++ incr(r) (Septiembre 2010)*

a) **Especificar ecuacionalmente** la función ++ que, dadas dos listas de tipo t devuelve la lista que se obtiene al juntarlas

++:: ([t], [t]) → [t]

Ejemplos: [1, 7] ++ [8, 5, 9] = [1, 7, 8, 5, 9]
[] ++ [8, 5, 9] = [8, 5, 9]

b) **Especificar ecuacionalmente** la función "incr" que dada una lista de enteros devuelve la lista que se obtiene sumando un 1 a cada elemento de la lista. Si la lista es vacía, devuelve la lista vacía:

incr:: ([Int]) → [Int]

Ejemplos: incr([4, 6, 5]) = [5, 7, 6]

c) **Probar por inducción** que para dos listas cualesquiera, s y r, se cumple la siguiente propiedad:

incr(s ++ r) = incr(s) ++ incr(r)

La inducción debe realizarse sobre s considerando el caso básico **s = []** y el caso inductivo **s = z:w**. La hipótesis de la inducción consistirá en suponer que las listas w y r cumplen la propiedad que se está probando.

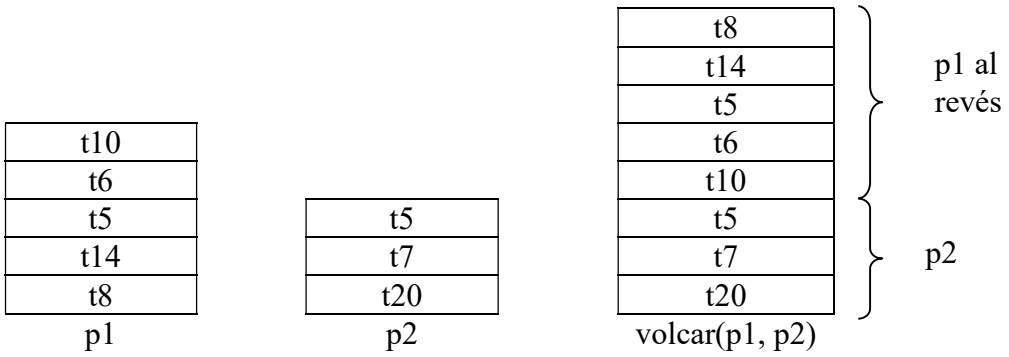
C) Operaciones sobre pilas

Dar ecuaciones que definan las siguientes operaciones sobre los tipos de datos Pila Int o Pila t según el caso:

1) volcar

Función que, dadas dos pilas, obtiene la que resulta de volcar una sobre la otra: *volcar*.

Ejemplo: (los *ti* son elementos de tipo t)

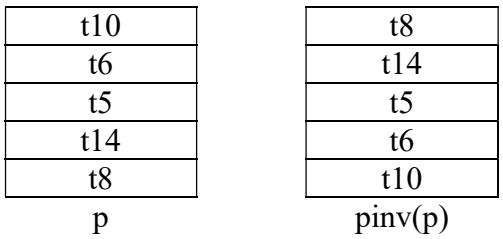


Hay que definir la funcíon *volcar* de manera recursiva y sin utilizar ninguna función auxiliar.

2) pinv

Función que, dada una pila, obtiene la pila inversa: *pinv*.

Ejemplo: (los *ti* son elementos de tipo t)

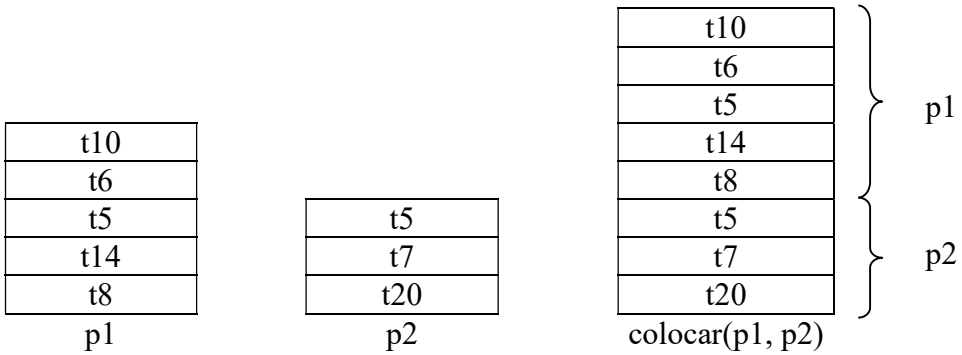


La funcíon *pinv* ha de ser definida utilizando la función *volcar* definida en el ejercicio anterior. Por tanto, *pinv* no es una función recursiva.

3) colocar

Función que, dadas dos pilas, obtiene la que resulta de colocar una sobre la otra: *colocar*.

Ejemplo: (los *ti* son elementos de tipo t)



D) <u>Pruebas por inducción para pilas</u>
Demostrar los siguientes **teoremas** inductivos:

1) altura(volcar(p, q)) = altura(p) + altura(q)

Probar por inducción que para dos pilas cualesquiera p y q s se cumple lo siguiente:

altura(volcar(p, q)) = altura(p) + altura(q)

2) altura(colocar(p, q)) = altura(p) + altura(q)

Probar por inducción que para dos pilas cualesquiera p y q s se cumple lo siguiente:

altura(colocar(p, q)) = altura(p) + altura(q)

E) Operaciones sobre colas

Dar ecuaciones que definan las siguientes operaciones sobre los tipos de datos Cola Int o Cola t según el caso:

1) juntar

Función que, dadas dos colas de tipo t, devuelve la cola que se obtiene juntando las dos colas (primera cola seguida de la segunda cola): *juntar*.

Ejemplos:

juntar (<< 3, 4 >>, << 5, 6 >>) = << 3, 4, 5, 6 >>
juntar (<< >>, << 5, 6 >>) = << 5, 6 >>

2) sumar

Función que, dada una cola de tipo Int, devuelve la suma de los elementos de la cola: *sumar*.

Ejemplos:

sumar(<< 3, 4, 5 >>) = 12
sumar(<< >>) = 0

F) <u>Pruebas por inducción para colas</u>
Demostrar los siguientes **teoremas** inductivos:

1) sumar(juntar(c, d)) = sumar(c) + sumar(d)

Probar por inducción que para dos colas de enteros cualesquiera c y d se cumple lo siguiente:

sumar(juntar(c, d)) = sumar(c) + sumar(d)

2) sumar(juntar(c, d)) = sumar(juntar(d, c))

Probar por inducción que para dos colas de enteros cualesquiera c y d se cumple lo siguiente:

sumar(juntar(c, d)) = sumar(juntar(d, c))

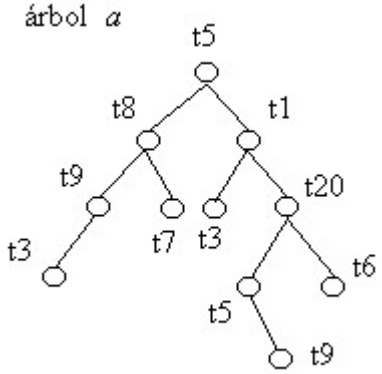
G) Operaciones sobre árboles binarios

Dar ecuaciones que definan las siguientes operaciones sobre los tipos de datos Arbin Int o Arbin t según el caso:

1) nveces

Función que calcula el número de veces que un elemento dado aparece en un árbol binario: *nveces*.

Ejemplos: (los *ti* son elementos de tipo t)



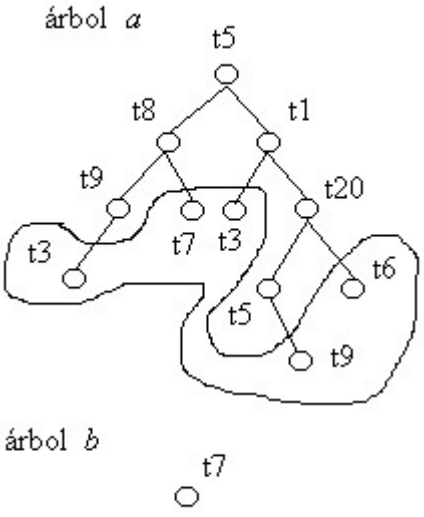
nveces(t5, *a*) = 2
nveces(t12, *a*) = 0

nveces(t8, Avacio) = 0

2) frontera

Función que calcula la frontera (lista de hojas) de un árbol binario: *frontera*.

Ejemplos: (los *ti* son elementos de tipo t)



frontera(*a*) = [t3, t7, t3, t9, t6]

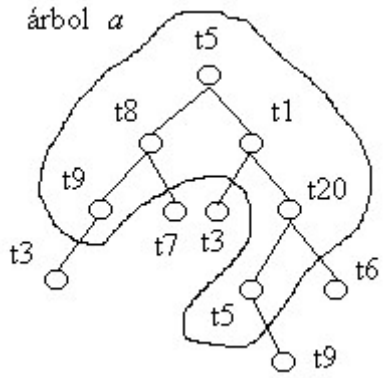
frontera(Avacio) = [ ]

frontera(*b*) = [t7]

3) ninternos

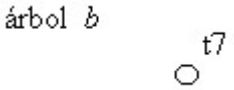
Funcion que calcula el número de nodos internos (nodos que no son hojas) de un árbol binario: *ninternos*.

Ejemplos: (los *ti* son elementos de tipo t)



ninternos(*a*) = 6

ninternos(Avacio) = 0

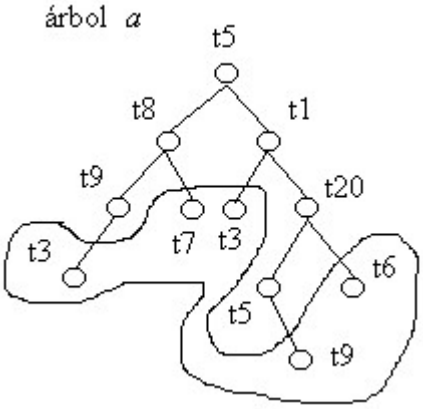


ninternos(*b*) = 0

4) nhojas

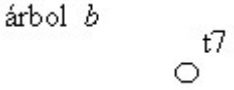
Función que calcula el número de hojas de un árbol binario: *nhojas*.

Ejemplos: (los *ti* son elementos de tipo t)



nhojas(*a*) = 5

nhojas(Avacio) = 0



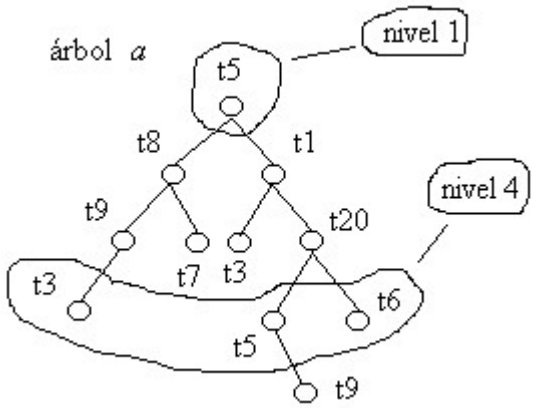
nhojas(*b*) = 1

5) nodos_nivel

Función que calcula el número de nodos que un árbol binario dado tiene a un determinado nivel: *nodos_nivel*.

Recordemos que el nivel de la raíz es 1, y que para cualquier otro nodo su nivel es uno más que el de su padre. Si se pide calcular el número de nodos de nivel 0 o inferior, se ha de devolver un mensaje de error.

Ejemplos: (los *ti* son elementos de tipo t)



nodos_nivel(*a*, 1) = 1

nodos_nivel(*a*, 4) = 3

nodos_nivel(*a*, 7) = 0

nodos_nivel(*a*, 0) = error

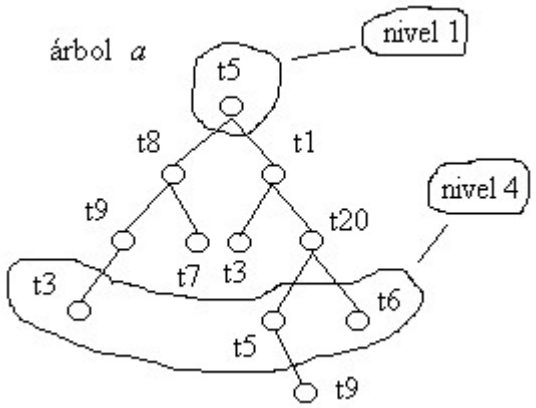
nodos_nivel(Avacio, 2) = 0

6) listanivel

Función que calcula la lista de nodos (de izquierda a derecha) que un árbol binario tiene a un determinado nivel: *listanivel*.

Si se pide calcular la lista de nodos de nivel 0 o inferior, se ha de devolver un mensaje de error.

Ejemplos: (los *ti* son elementos de tipo t)



listanivel(*a*, 1) = [t5]

listanivel(*a*, 4) = [t3, t5, t6]

listanivel(*a*, 7) = [ ]

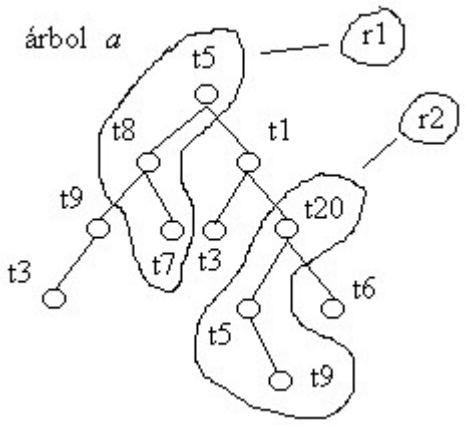
listanivel(*a*, 0) = error

listanivel(Avacio, 2) = [ ]
listanivel(Avacio, 0) = error

7) es_rama

Función que dados una lista y un árbol binario decide si la lista coincide con una rama del árbol: *es_rama*.

Ejemplos: (los *ti* son elementos de tipo t)



r1 = [t5, t8, t7]

r2 = [t20, t5, t9]

es_rama(r1, *a*) = True

es_rama(r2, *a*) = False

es_rama ([ ], *a*) = False

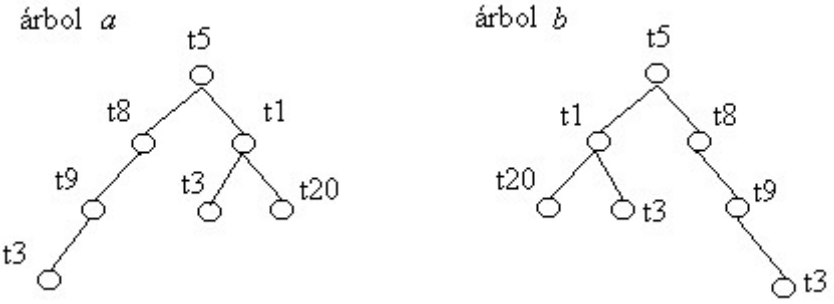
es_rama([ ], Avacio) = True

es_rama([t4, t30, t1, t2], *a*) = False

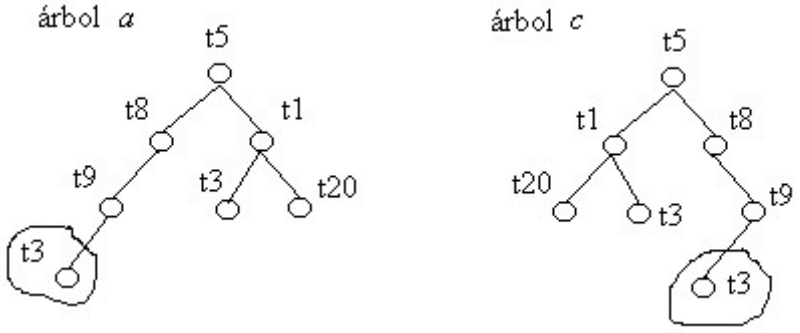
8) es_espejo

Función que decide si un árbol binario es la imagen en espejo de otro árbol binario: *es_espejo*.

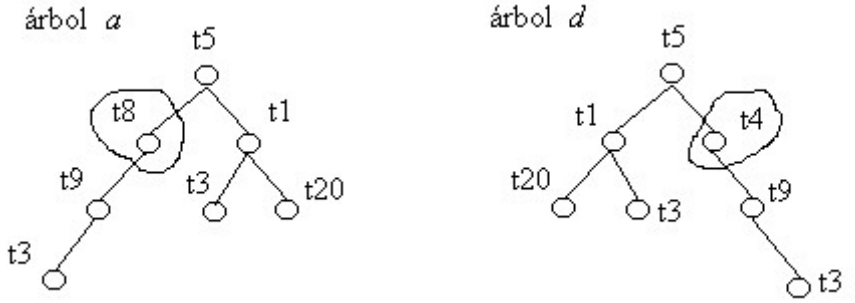
Ejemplos: (los *ti* son elementos de tipo t)



es_espejo(*a*, *b*) = True



es_espejo(*a*, *c*) = False



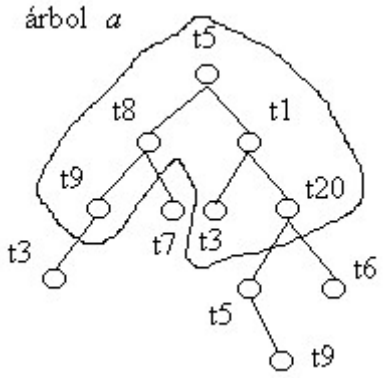
es_espejo(*a*, *d*) = False
es_espejo(*a*, Avacio) = False
es_espejo(Avacio, Avacio) = True

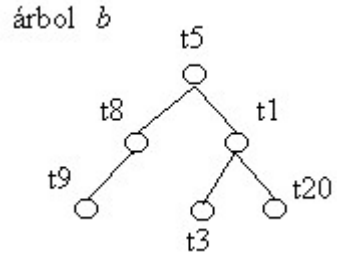
9) es_prefijo

Función que decide si un árbol binario es prefijo de otro: *es_prefijo*.

Para que un árbol *a* sea prefijo de otro *b*, el árbol *b* tiene que ser un árbol obtenido añadiendo hacia abajo todos los nodos que se quiera al árbol *a*. Un árbol vacío es prefijo de cualquier árbol.

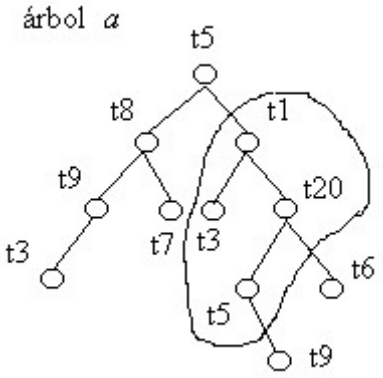
Ejemplos: (los *ti* son elementos de tipo t)

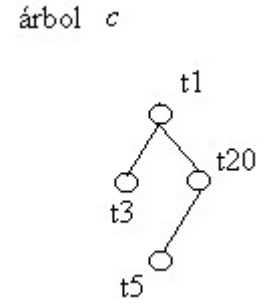




es_prefijo(*b*, *a*) = True

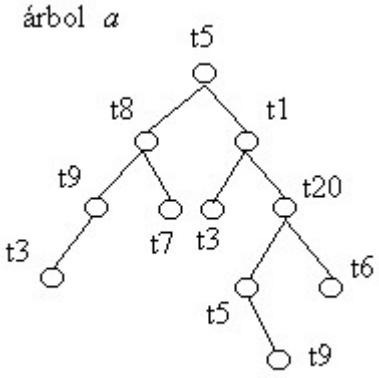
es_prefijo(*a*, *b*) = False

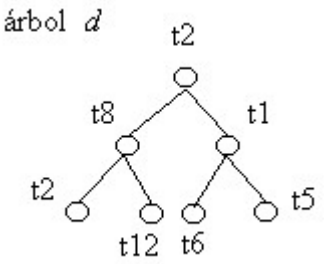




es_prefijo(*c*, *a*) = False

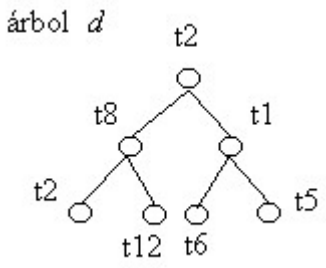
es_prefijo(*a*, *c*) = False

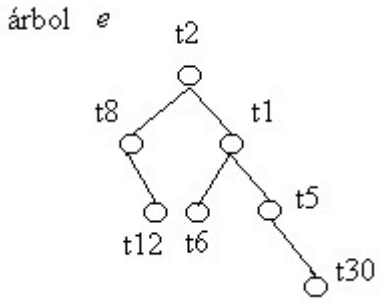




es_prefijo(*d*, *a*) = False

es_prefijo(*a*, *d*) = False





es_prefijo(*e*, *d*) = False

es_prefijo(*d*, *e*) = False

es_prefijo(Avacio, *d*) = True

es_prefijo(*d*, Avacio) = False

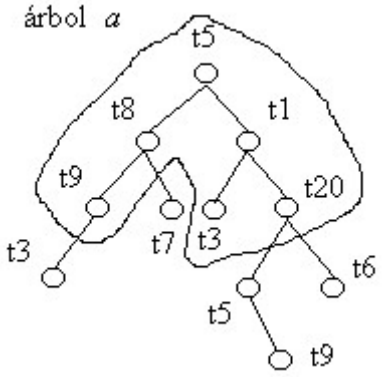
es_espejo(Avacio, Avacio) = True

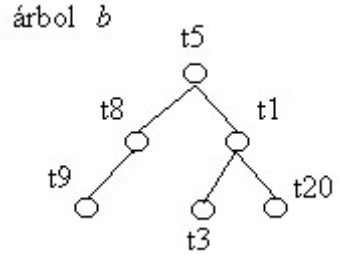
10) es_subarbol

Función que decide si un árbol binario es subárbol de otro: *es_subarbol*.

Para que un árbol *a* sea subárbol de otro *b*, el árbol *a* (conservando su estructura) ha de aparecer en *b*. El árbol vacío es subárbol de cualquier árbol.

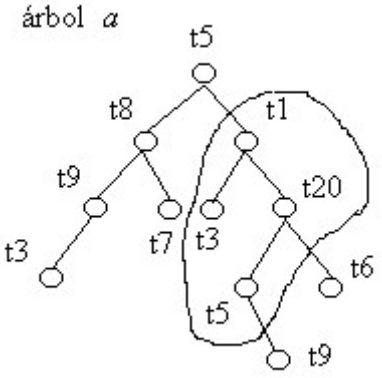
Ejemplos: (los *ti* son elementos de tipo t)

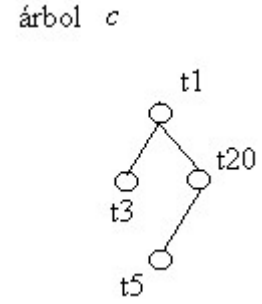




es_subarbol(*b*, *a*) = True

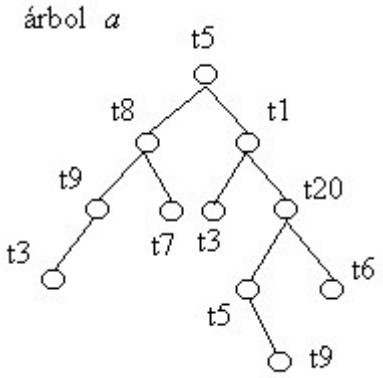
es_subarbol(*a*, *b*) = False

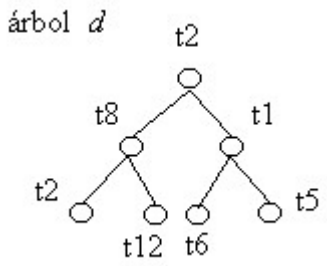




es_subarbol(*c*, *a*) = True

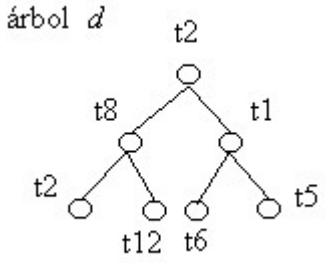
es_subarbol(*a*, *c*) = False

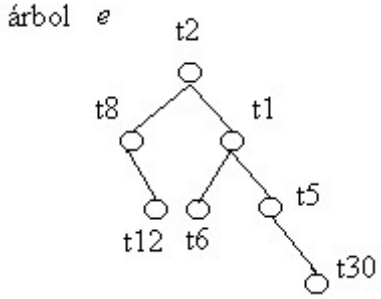




es_subarbol(*d*, *a*) = False

es_subarbol(*a*, *d*) = False





es_subarbol(*e*, *d*) = False

es_subarbol(*d*, *e*) = False

es_subarbol(Avacio, *d*) = True

es_subarbol(*d*, Avacio) = False

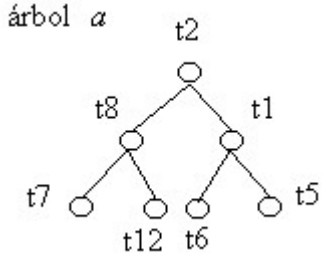
es_subarbol(Avacio, Avacio) = True

11) preorden

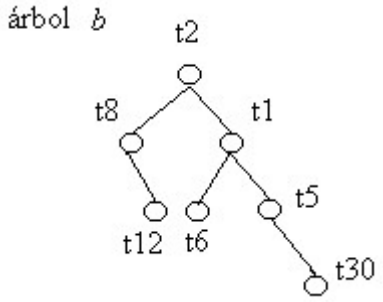
Función que obtiene la secuencia que resulta de recorrer un árbol binario en preorden: *preorden*.

Recorrer un árbol en preorden consiste en visitar primero la raíz, recorrer a continuación el subárbol izquierdo en preorden y recorrer por último el subárbol derecho en preorden. A este recorrido se le llama preorden porque la raíz es la primera en ser visitada.

Ejemplos: (los *ti* son elementos de tipo t)



preorden(*a*) = [t2, t8, t7, t12, t1, t6, t5]



preorden(*b*) = [t2, t8, t12, t1, t6, t5, t30]

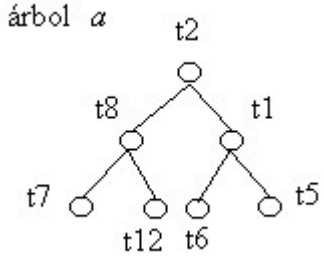
preorden(Avacio) = [ ]

12) inorden

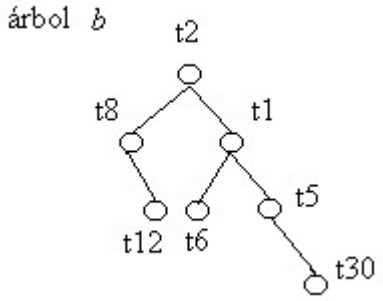
Función que obtiene la secuencia que resulta de recorrer un árbol binario en inorden: *inorden*.

Recorrer un árbol en inorden consiste en recorrer primero el subárbol izquierdo en inorden, visitar a continuación la raíz y recorrer por último el subárbol derecho en inorden. A este recorrido se le llama inorden porque la raíz es visitada después de recorrer el subárbol izquierdo y antes de recorrer el subárbol derecho.

Ejemplos: (los *ti* son elementos de tipo t)



inorden(*a*) = [t7, t8, t12, t2, t6, t1, t5]



inorden(*b*) = [t8, t12, t2, t6, t1, t5, t30]

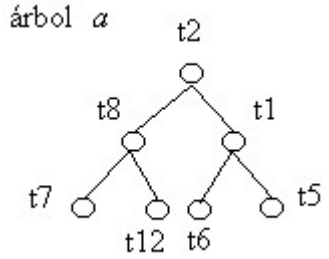
inorden(vacio) = [ ]

13) postorden

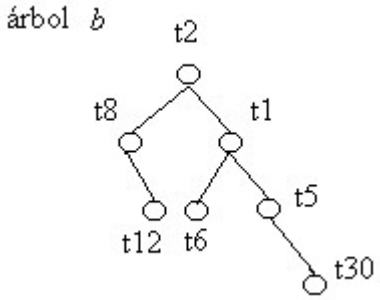
Función que obtiene la lista que resulta de recorrer un árbol binario en postorden: *postorden*.

Recorrer un árbol en postorden consiste en recorrer primero el subárbol izquierdo en postorden, recorrer a continuación el subárbol derecho en postorden y visitar por último la raíz. A este recorrido se le llama postorden porque la raíz es la última en ser visitada.

Ejemplos: (los *ti* son elementos de tipo t)



postorden(*a*) = [t7, t12, t8, t6, t5, t1, t2]



postorden(*b*) = [t12, t8, t6, t30, t5, t1, t2]

postorden(Avacio) = [ ]

H) Pruebas por inducción para árboles binarios

Demostrar los siguientes **teoremas** inductivos:

1) nnodos(a) ≤ $2^{prof(a)}$ – 1

Probar por inducción que para todo árbol binario *a* se cumple lo siguiente:

$$\text{nnodos}(a) \leq 2^{\text{prof}(a)} - 1$$

Es decir, todo árbol binario *a* tiene a lo sumo $2^{\text{prof}(a)} - 1$ nodos.

2) ninternos(a) ≤ $2^{(prof(a) - 1)}$ – 1

Probar por inducción que para todo árbol binario **no vacío** *a* se cumple lo siguiente:

$$\text{ninternos}(a) \leq 2^{(\text{prof}(a) - 1)} - 1$$

Es decir, el número de nodos internos de todo árbol binario no vacío *a* es a lo sumo $2^{(\text{prof}(a) - 1)} - 1$.

3) nhojas(a) ≤ $2^{(prof(a) - 1)}$

Probar por inducción que para todo árbol binario **no vacío** *a* se cumple lo siguiente:

$\text{nhojas}(a) \leq 2^{(\text{prof}(a) - 1)}$

Es decir, el número de hojas de todo árbol binario no vacío *a* es a lo sumo $2^{(\text{prof}(a) - 1)}$.

4) ninternos(a) ≤ nhojas(a) * (prof(a) – 1)

Probar por inducción que para todo árbol binario **no vacío** *a* se cumple lo siguiente:

$$\text{ninternos}(a) \leq \text{nhojas}(a) * (\text{prof}(a) - 1)$$

Es decir, el número de nodos internos de todo árbol binario *a* es a lo sumo: $\text{nhojas}(a) * (\text{prof}(a) - 1)$

5) longitud(inorden(a)) = nnodos(a)

Probar por inducción que para todo árbol binario *a* se cumple lo siguiente:

longitud(inorden(a)) = nnodos(a)

Es decir, la longitud de la lista que resulta de recorrer en inorden cualquier árbol binario *a* es igual al número de nodos de *a*.

6) ninternos(a) ≥ nhojas(a) – 1

Probar por inducción que para todo árbol binario **no vacío** *a* se cumple lo siguiente:

ninternos(*a*) ≥ nhojas(*a*) – 1

Es decir, el número de nodos internos de todo árbol binario *a* es mayor o igual al número de hojas menos 1

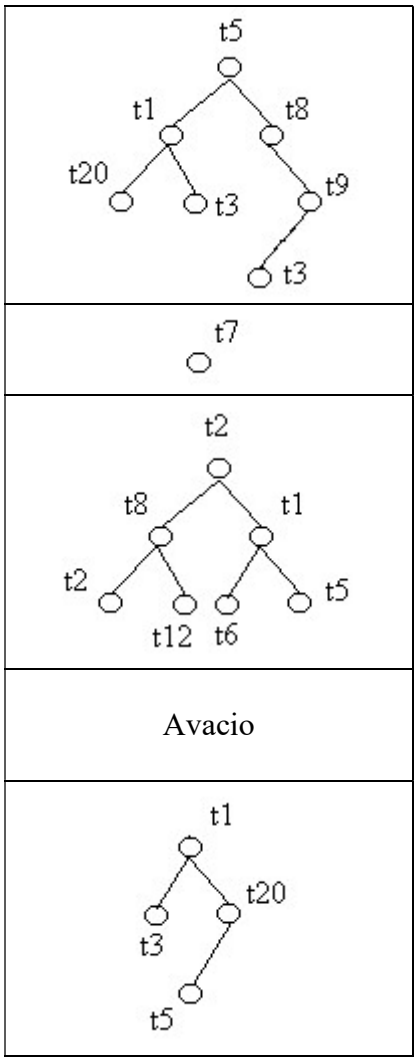
I) Operaciones con mezcla de tipos abstractos de datos

Dar ecuaciones que definan las siguientes operaciones:

1) nvecespab

Función que calcula el número de veces que un elemento dado aparece en una pila de árboles binarios de tipo t: *nvecespab*.

Ejemplo 1: (los *ti* son elementos de tipo t)



p

nvecespab(t3, p) = 3

nvecespab(t14, p) = 0

nvecespab(t7, p) = 1

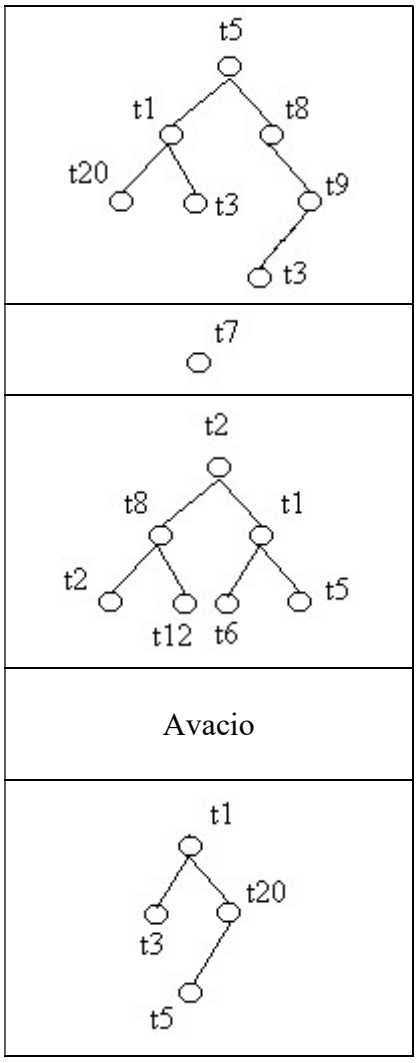
Ejemplo 2:

nvecespab(t8, Pvacia) = 0

2) lispostpab

Función que obtiene la secuencia que resulta de recorrer en postorden todos los árboles de una pila de árboles binarios de tipo t: *lispostpab*.

Ejemplo 1: (los *ti* son elementos de tipo t)



p

lispostpab(p) = [t20, t3, t1, t3, t9, t8, t5, t7, t2, t12, t8, t6, t5, t1, t2, t3, t5, t20, t1]

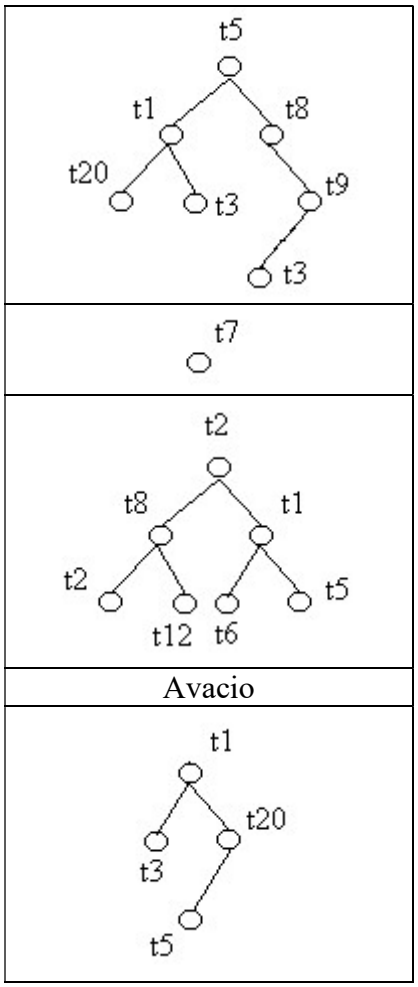
Ejemplo 2:

lispostpab(Pvacia) = [ ]

3) lisfrontpab

Función que obtiene la secuencia formada por las fronteras consecutivas de todos los árboles de una pila de árboles binarios de tipo t: *lisfrontpab*.

Ejemplo 1: (los *ti* son elementos de tipo t)



p

lisfrontpab(p) = [t20, t3, t3, t7, t2, t12, t6, t5, t3, t5]

Ejemplo 2:

lisfrontpab(Pvacia) = [ ]