

Fundamentos de programación

Datos complejos I: Recursión numérica, listas arbitrariamente largas y estructuras recursivas

Facultad de Ingeniería. Universidad del Valle

Octubre de 2018

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

1 Recursión numérica

2 Listas arbitrariamente largas

3 Estructuras recursivas (Árboles)

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

1 Recursión numérica

2 Listas arbitrariamente largas

3 Estructuras recursivas (Árboles)

Definición

Los números naturales se pueden definir de la siguiente forma:

- 1 0 es un natural
- 2 Si n es un numero natural (`add1 n`) también lo es

Definición

Los números naturales se pueden definir de la siguiente forma:

- 1 0 es es un natural
- 2 Si $n = 0$ $(\text{add1 } 0) = 1$ es también natural
- 3 Si $n = 1$ $(\text{add1 } 1) = 2$ es también natural
- 4 Si $n = 2$ $(\text{add1 } 2) = 3$ es también natural

base

$$2 = (\text{add1 } 1)$$

$$2 = (\text{add1 } (\text{add1 } 0))$$

$$4 = (\text{add1 } (\text{add1 } (\text{add1 } (\text{add1 } 0))))$$

Definición

Los números naturales se pueden definir de la siguiente forma:

- 1 0 es el primer número
- 2 Si $n = 0$ $(\text{add1 } 0) = 1$
- 3 Si $n = 0$ $(\text{add1 } (\text{add1 } 0)) = 2$
- 4 Si $n = 0$ $(\text{add1 } (\text{add1 } (\text{add1 } 0))) = 3$

Definición

Si se observa un número natural se puede definir haciendo un llamado varias veces de la función **add1** a esto lo vamos a conocer como **definición recursiva**

```
(first 1)
(first (rest 1))
(first (rest (rest 1)))
(first (rest (rest (rest 1))))
```

Indicio
Recursión



$$n! = n * (n-1) * (n-2) * (n-3) * \dots * 1 =$$
$$5! = 5 \times 4 \times 3 \times 2 \times 1 \quad 4! = 4 \times 3 \times 2 \times 1$$

Definición

Esto también aplica para casos de funciones, por ejemplo el **factorial** que se define de la siguiente forma:

$$fact(n) = n * (n - 1) * (n - 2) * \dots * 1, fact(0) = 1 \quad (1)$$

Si observa es una secuencia de multiplicaciones.

|

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

$$4! = 4 \times 3 \times 2 \times 1$$

$$3! = 3 \times 2 \times 1$$

$$2! = 2 \times 1$$

$$1! = 1$$

$$0! = 1$$

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1$$

Def recursiva

Numero

$$\text{fact}(5) = (* 5 (\text{fact } 4))$$

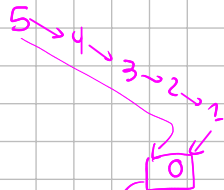
$$\text{fact}(4) = (* 4 (\text{fact } 3))$$

$$\text{fact}(3) = (* 3 (\text{fact } 2))$$

$$\text{fact}(2) = (* 2 (\text{fact } 1))$$

$$\text{fact}(1) = (* 1 (\text{fact } 0))$$

$$\text{fact}(0) = 1$$



Caso base

llamada recursiva

$$\text{fact } n = \begin{cases} (* n (\text{fact } (- n 1))) & n \neq 0 \\ 1 & n = 0 \end{cases}$$

(Caso base / trivial)

Definición

Si observamos la forma podemos definir una función para calcular el factorial ¿Como sería?

```
;;Contrato factorial: numero -> numero  
(define (factorial n)  
    ....  
)
```

Definición

Empezamos a analizar, si $n = 1$ el factorial es 1

```
;;Contrato factorial: numero -> numero
(define (factorial n)
  (cond
    [(= n 1) 1]
    ...
  )
```

Definición

Empezamos a analizar, si $n = 1$ el factorial es $1*1$, y si $n = 2$ entonces $1*1*2$, **empezamos a ver un patrón**

```
;;Contrato factorial: numero -> numero
(define (factorial n)
  (cond
    [(= n 1) 1]
    [else
     (* n (factorial (- n 1)))]
  )
)
```

Definición

Estas funciones que se llaman así mismas son llamadas **funciones recursivas** debe tener en cuenta:

- 1 Una condición de parada, para que no se llame infinitamente. Es el caso inicial.
- 2 La función debe siempre retornar el mismo tipo de dato
- 3 Un llamado a la misma función, utilizando alguna función para unir las salidas (una operación matemática)

Ejemplo

- 1 Diseñe una función **multiplicación**, la cual recibe dos números (a y b) esta retorna el resultado de sumar b veces a
- 2 Diseñe una función **eleva**r, la cual recibe dos números (a y b), esta retorna el resultado de multiplicar b veces a

$$a, b > 0$$

$$b = 1 \quad R/a \quad a > b > 0$$

$$b = 0$$

$$a^b = 1 \quad 0^1 = a \times a^0 \quad a + 1$$

$$(\times a b) = \underbrace{a + \boxed{a} + \boxed{a} + \dots + \boxed{a}}_{\substack{b \text{ terms}}} \boxed{1}$$

$$(\text{mult } a b) \begin{cases} (+ a (\text{mult } a (- b 1))) & b \neq 1 \\ a & b = 1 \end{cases}$$

$$(\times 5 3)$$

$$(\times 5 3) = 5 \times 2 + 5$$

$$5 \times 2 = 5 \times 5 \times 1$$

$$5 \times 1 = 5$$

$$\begin{matrix} 5 \times 3 & & 5 \times 2 & & 5 \times 1 \\ 5 & \left[+ & 5 & + & 5 \right] \end{matrix}$$

$$a^b = \frac{a \times a \times a \times \dots \times a}{b \text{ veces}}$$

$$a^b = a \times a^{b-1} \quad \text{Recursive}$$

$$a^0 = 1$$

$$\text{elevar } a \text{ } b = \begin{cases} a \times \text{elevar } a \text{ } (-b - 1) & b \neq 0 \\ 1 & b = 0 \end{cases}$$

Ejemplo

Diseñe una función **multiplicación**, la cual recibe dos números (a y b) esta retorna el resultado de sumar b veces a

```
;;Contrato multiplicacion: numero, numero -> numero
(define (multiplicacion a b)
  (cond
    [(= b 1) a]
    [else
     (+ a (multiplicacion a (- b 1) ))]
  )
)
```

Ejemplo

Diseñe una función **eleva**r, la cual recibe dos números (a y b), esta retorna el resultado de multiplicar b veces a

```
;;Contrato elevar: numero, numero -> numero
(define (elevar a b)
  (cond
    [(= b 1) a]
    [else
     (* a (elevar a (- b 1)))]
  )
)
```

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

1 Recursión numérica

2 Listas arbitrariamente largas

3 Estructuras recursivas (Árboles)

Listas arbitrariamente largas

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

Definición

Hasta el momento hemos trabajado con listas de un tamaño dado, pero que sucede si trabajamos con listas de diferente tamaño. Por ejemplo una función que recibe listas de símbolos y se desea encontrar uno, podríamos diseñar una función así.

```
;;Contrato buscar-simbolo: lista-de-simbolos, simbolo ->
    booleano
(define (buscar-simbolo lista nombre)
  (cond
    [(eqv? (first lista) nombre) #t]
    [else ... ]
  )
)
```

*(first crest
first crest crest)*

Aquí miramos si el primer elemento de la lista es lo que buscamos, sin embargo, ¿Como verificamos el segundo, y los otros elementos?

$(\text{cons } 'a (\text{cons } 'b (\text{cons } 'c (\text{cons } 'd (\text{empty}))))$
 (cons 'a (cons 'b (cons 'c (cons 'd (empty)))))

Syn = d'

$(\text{eqv?} (\text{first } l) \text{ syn})$

$(\text{eqv?} (\text{first } (\text{rest } l)) \text{ syn})$

list

list

rest

rest

empty

$(\text{buscar-s } l \text{ syn})$

False

lst = empty

True

$(\text{eqv?} (\text{first } l) \text{ syn})$

$(\text{buscar-s } (\text{rest } l) \text{ syn})$

else

condicioner de parada

llamada
recursiva

Definición

Una idea sería analizar el resto de la lista (que es una lista que contiene los otros elementos)

```
;;Contrato buscar-simbolo: lista-de-simbolos, simbolo ->  
    booleano  
(define (buscar-simbolo lista nombre)  
  (cond  
    [(eqv? (first lista) nombre) #t]  
    [else ... (rest lista) ...]  
  )  
)
```

¿Observan algo en el contrato? ¿El resto de la lista que es?

Listas arbitrariamente largas

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

Definición

Podríamos enviar el resto de la lista a la misma función (para que siga buscando) y mirar si el símbolo está:

```
;;Contrato buscar-simbolo: lista-de-simbolos, simbolo ->  
    booleano  
(define (buscar-simbolo lista nombre)  
  (cond  
    → [(eqv? (first lista) nombre) #t]  
    → [else (buscar-simbolo (rest lista) nombre)]  
  )  
)
```

Pero, hay algo que está mal ¿Que pasa si el elemento no está en la lista?

Listas arbitrariamente largas

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

Definición

El problema es que si seguimos buscando, ¿Que hacemos cuando llegamos al final de la lista (empty)?

```
;;Contrato buscar-simbolo: lista-de-simbolos, simbolo ->  
    booleano  
(define (buscar-simbolo lista nombre)  
  (cond  
    1) [(empty? lista) #f]  
    2) [(eqv? (first lista) nombre) #t]  
    3) [else (buscar-simbolo (rest lista) nombre)]  
  )  
)
```

Debemos verificar que si llega al final de la lista

Definición

Para el diseño de funciones que trabajan sobre listas arbitrariamente grandes debe tener en cuenta:

- 1 Analizar el primer elemento de la lista: Verificación.
- 2 Tener en cuenta que la lista termina cuando esta es **empty**. Condición de parada
- 3 Analizar el resto de la lista, llamando la misma función.
Condición de llamado recursivo

Ejemplo

- 1 Diseñe una función **buscar-numero** que recibe un número y una lista de números. Esta función indica que el número está en la lista de números
- 2 Diseñe una función **buscar-persona-nombre** que recibe una lista de estructuras persona que tiene tres atributos: nombre, edad y cargo; y recibe un nombre. Esta función indica si hay alguna persona con el nombre indicado

Listas arbitrariamente largas

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

Definición

Diseñe una función **buscar-numero** que recibe un número y una lista de números. Esta función indica que el número está en la lista de números

```
;;Contrato buscar-numero: lista-de-numeros, numero ->  
    booleano  
(define (buscar-numero numero listan)  
  (cond  
    [(empty? listan) #f]  
    [(eqv? (first listan) numero) #t]  
    [else (buscar-numero numero (rest listan))])  
)
```

Definición

Diseñe una función **buscar-persona-nombre** que recibe una lista de estructuras persona que tiene tres atributos: nombre, edad y cargo; y recibe un nombre. Esta función indica si hay alguna persona con el nombre indicado

```
;;Contrato buscar-numero: lista-de-personas , simbolo ->  
    booleano  
(define-struct persona (nombre edad cargo))  
(define (buscar-persona-nombre nombre listaPersonas)  
  (cond  
    [(empty? listaPersonas) #f]  
    [(eqv? (persona-nombre (first listaPersonas))  
           nombre) #t]  
    [else (buscar-persona-nombre nombre (rest  
                                   listaPersonas))])  
  )  
)
```

Listas arbitrariamente largas

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

Definición

También podemos ir más allá, por ejemplo podemos realiza la suma de elementos en una lista de números observe:

```
;;Contrato sumar-lista: lista-de-numeros -> numero
(define (sumar-lista listan)
  (cond
    [(empty? listan) 0]
    [else (+ (first listan) (sumar-lista (rest listan)))]
  )
)
```

¿Que puede decir el comportamiento de esta función?
Analicemos el caso **(cons 1 (cons 2 (cons 3 empty)))**

Listas arbitrariamente largas

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

Definición

Cuando llamamos la función con la lista **(cons 1 (cons 2 (cons 3 empty)))**.

```
;;Contrato sumar-lista: lista-de-numeros -> numero
(define (sumar-lista listan)
  (cond
    [(empty? listan) 0]
    [else (+ (first listan) (sumar-lista (rest listan)))]
  )
)
```

La lista no está vacía por ende, se ejecuta la clausula **else** y queda lo siguiente:

(+ 1 (sumar-lista (cons 2 (cons 3 empty))))

Definición

En el siguiente llamado se tiene **(cons 2 (cons 3 empty))**.

```
;;Contrato sumar-lista: lista-de-numeros -> numero
(define (sumar-lista listan)
  (cond
    [(empty? listan) 0]
    [else (+ (first listan) (sumar-lista (rest listan)))]
  )
)
```

La lista no está vacía por ende, se ejecuta la clausula **else** y queda lo siguiente:

(+ 1 (+ 2 sumar-lista (cons 3 empty))))

Definición

En el siguiente llamado se tiene (**cons 3 empty**).

```
;;Contrato sumar-lista: lista-de-numeros -> numero
(define (sumar-lista listan)
  (cond
    [(empty? listan) 0]
    [else (+ (first listan) (sumar-lista (rest listan)))]
  )
)
```

La lista no está vacía por ende, se ejecuta la clausula **else** y queda lo siguiente:

(+ 1 (+ 2 (+ 3 (sumar-lista empty))))

Listas arbitrariamente largas

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

Definición

En el siguiente llamado se tiene **empty**.

```
;;Contrato sumar-lista: lista-de-numeros -> numero
(define (sumar-lista listan)
  (cond
    [(empty? listan) 0]
    [else (+ (first listan) (sumar-lista (rest listan)))]
  )
)
```

La lista está vacía por ende, retorna 0 y se tiene
(+ 1 (+ 2 (+ 3 0))) y se obtiene 6.

Definición

También podemos generar listas, observe:

```
;;Contrato doble-lista: lista-de-numeros ->  
  lista-de-numeros  
(define (doble-lista listan)  
  (cond  
    [(empty? listan) empty]  
    [else (cons (* (first listan) 2) (doble-lista (  
      rest listan)))]  
  )  
)
```

Ejemplo

- 1 Diseñe una función **multiplicar-lista** que recibe una lista de números. Esta función retorna los números de la lista multiplicados entre sí.
- 2 Diseñe una función **suma-lista-dobles** que recibe una lista de números y un número, esta retorna la suma de la multiplicación de cada uno de los elementos de la lista por el número.
- 3 Diseñe una función **elevar-cuadrado-lista** recibe una lista de números y esta retorna esa misma lista pero con los elementos elevados al cuadrado

Un supermercado requiere una aplicación para el calculo de una venta. El supermercado vende los siguientes tipos de productos, los cuales son:

1) Frutas: Las cuales tienen un nombre, peso(g) y un valor gramos

2) Carnes: las cuales itenen un nombre, un tipo (cerdo, pollo, res y pescado) y un peso (g)

- Si es cerdo, tiene un nombre (parte) y un precio(g)
- Si es pollo, tiene un nombre (parte), un tipo (gallina, pollo) y un precio por g
- Si es res , tiene un nombre(parte), proceso(molida ono) precio por g.
- Si es pesado, tiene un nombre (tipo de pez), origen (rio lago o mar) precio por g.

3) Electrodoméstico: Nombre, Marca, un precio por unidad

4) Higiene personal: Nombre, marca, precio por docena

5) Bebidas: nombre, marca, precio por unidad

Un venta es una colección de elementos que se compran, donde se tiene el elementos (producto supermercado) y el número de unidades/gramos/docenas que se compran.

Si se compran Frutas o Carnes el IVA es el 5% por estos elementos, en otro caso se aplica 19%

Diseñe una función que reciba una lista de ventas, un número de bolsas que necesita la persona (impuesto 40 por bolsa) y retorne el detallado a pagar: Subtotal, IVA, impuesto a bolsas y total a pagar.

<https://pastebin.com/5t4bhSds>

1) ¿Que necesitan que solucione?

Un detallado a pagar a partir de una lista de elementos a comprar.

2) ¿Que necesito para especificar las entradas?

-> ¿Necesitamos estructuras?

* Si, hay unas estructuras los productos (Fruta, Higiene, Carnes, Bebidas, Electrodomestico, Cerdo, Pollo, Res, Pescado, Venta, detallado de la compra.

3) ¿Como vendería la función?

Entradas: Lista de ventas, número de bolsas

Salida: Detallado de la compra

Ejemplo

Voy a comprar 150 manzanas gramo cuesta 8, tres mil gramos de lomo de cerdo 15 el gramo y una lavadora 200000. 3 bolsas

Detallado: Subtotal: $150 \cdot 8 + 15 \cdot 3000 + 200000 = 246200$

IVA: $0.05(150 \cdot 8 + 15 \cdot 3000) + 0.19 \cdot 200000 = 40310$

Bolsas= $3 \cdot 40 = 120$

Total: 286630

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

1 Recursión numérica

2 Listas arbitrariamente largas

3 Estructuras recursivas (Árboles)

Estructuras recursivas (Árboles)

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

Definición

Existen estructuras cuyos campos pueden definirse con una estructura, un buen ejemplo de ello es una **muñeca rusa**



Estructuras recursivas (Árboles)

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

Definición

¿Como definiríamos una estructura que sea una muñeca rusa:

```
(define-struct rusa-doll  
  (doll-interna)  
)
```

Estructuras recursivas (Árboles)

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

Definición

Si queremos una muñeca que contenga otras dos adentro sería.

```
( make-rusa-doll  
  ( make-rusa-doll  
    ( make-rusa-doll empty ) ) )
```

Estructura vacía

Estructuras recursivas (Árboles)

Fundamentos
de
programación

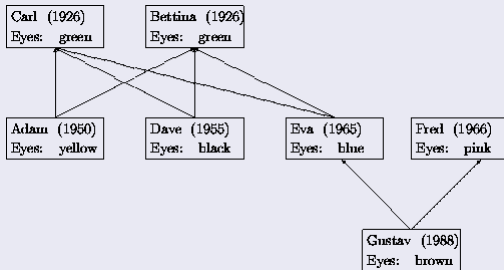
Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

Definición

Un caso más aplicado, son los árboles genealógico donde podemos relacionar los parentescos, por ejemplo:



Estructuras recursivas (Árboles)

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

Definición

Podemos definir un hijo (child) de la siguiente forma:

```
(define-struct child  
  (padre madre nombre fecha ojos)  
)
```

Estructuras recursivas (Árboles)

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

Definición

Por ejemplo podemos definir a alguien en la cima del árbol:

```
(make-child empty empty 'Carl 1926 'green)
```

Estructuras recursivas (Árboles)

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

Definición

Pero que pasa con un hijo:

```
(make-child  
  (make-child empty empty 'Carl 1926 'green)  
  (make-child empty empty 'Bettina 1926 'green)  
  'Adam  
  'yellow  
)
```

Estructuras recursivas (Árboles)

Fundamentos
de
programación

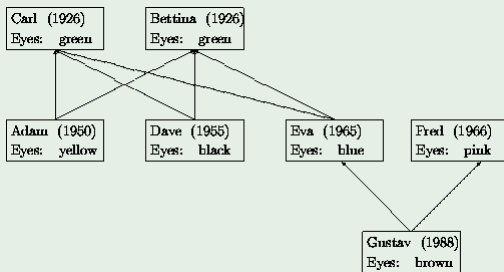
Recursión
numérica

Listas arbitra-
riamente
largas

Estructuras
recursivas
(Árboles)

Ejercicio

Defina las estructuras para el resto de la familia (en hoja de papel)



Estructuras recursivas (Árboles)

Fundamentos
de
programación

Recursión
numérica

Listas arbitra-
riamente
largas

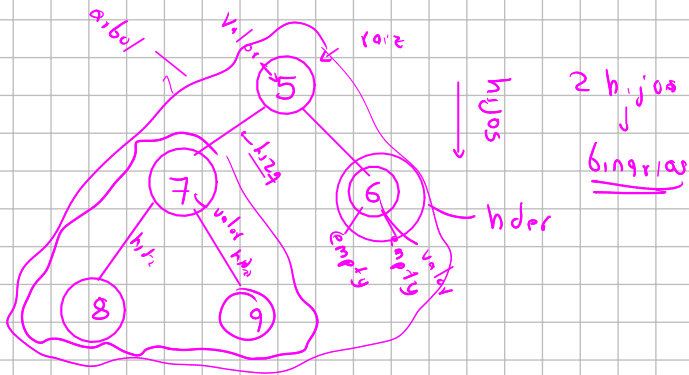
Estructuras
recursivas
(Árboles)

Ejercicio

```
;; Primera Generación:
(define Carl (make-child empty empty 'Carl 1926 'green))
(define Bettina (make-child empty empty 'Bettina 1926 '
    green))

;; Segunda Generación:
(define Adam (make-child Carl Bettina 'Adam 1950 'yellow)
)
(define Dave (make-child Carl Bettina 'Dave 1955 'black))
(define Eva (make-child Carl Bettina 'Eva 1965 'blue))
(define Fred (make-child empty empty 'Fred 1966 'pink))

;; Tercera Generación:
(define Gustav (make-child Fred Eva 'Gustav 1988 'brown))
```

(define-struct arbol (valor hizq hder))

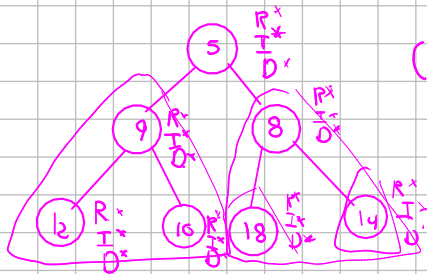
Recorridos en arboles - Preorden

← rackrt

- Inorden

← Normal recorrer ap

- Posorden

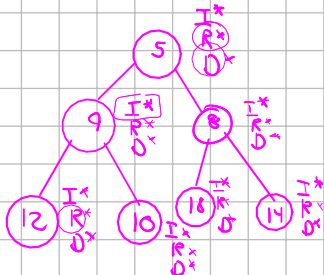


(5 9 12 10 8 18 14)

- 1) (arbol->valor arb)
- 2) (f (arbol->hizq arb))
- 3) (f (arbol-Q>hder arb))

Preorden

- 1) Visite la raiz
- 2) Visite el hijo izquierdo
- 3) Visite el hijo derecho

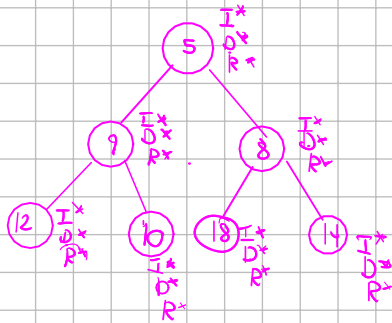


Recorrido inorden

- 1) Explore hijo izquierdo
- 2) Saque la raiz
- 3) Explore hijo derecho

(12 9 10 5 18 8 14)

- 1) (f (arbol->hizq arb))
- 2) (arbol->valor arb)
- 3) (f (arbol->hder arb))



Recorrido posorden

1) Recorra izquierda

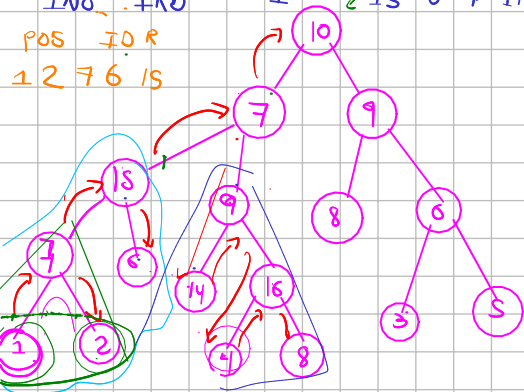
2) Recorra derecha

3) Dé el valor de la raíz

(12 10 9 18 14 8 5)

PRE R I D 10 7 15 7 1 2
 I N O I R O 1 7 2 15 6 7 14

POS I O R
 1 2 7 6 15



Recorridos

- Preorden: Raiz, hizq, hder
- Inorden: Hizq, Raiz, hder
- Posorden: Hizq, hder, raiz

Solo es hacer el RECORRIDO
 NO CODIFICAR

PREORDEN

(10 7 15 7 1 2 6 9
 14 16 4 8 9 8 6 3 5)

INORDEN

(1 7 2 15 6 7 14 9 4 16
 8 10 8 9 3 6 5)

POSORDEN

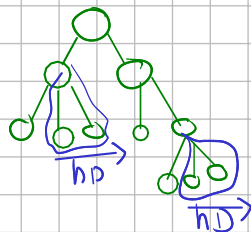
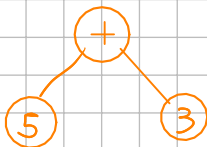
(1 2 7 6 15 14 4 8 16 9 7 8 3 5 6 9 10)

PRE (list 10 7 15 7 1 2 6 9 14 16 4 8 9 8 6 3 5)

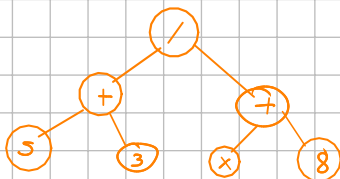
INOR (list 1 7 2 15 6 7 14 9 4 16 8 10 8 9 3 6 5)

POS (list 1 2 7 6 15 14 4 8 16 9 7 8 3 5 6 9 10)

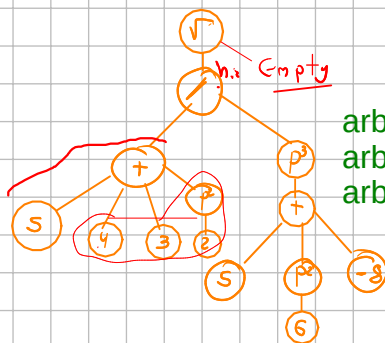
$$(5 + 3)$$



$$\frac{(5 + 3)}{(x + 8)}$$



$$\sqrt{\frac{5 + 4 + 3 + 2^2}{(5 + 6^2 - 8)^3}}$$



arbol n-ario
 arbol binario
 arbol ternario

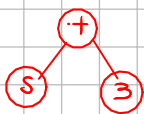
PRE ($\sqrt{ / (+ 5 \ 4 \ 3 \ (\text{expt } 2 \ 2))$
 $(\text{expt } (+ 5 \ (\text{expt } 6 \ 2) - 8) \ 3))$)

INO $(5 + 4 + 3 + 2^2) / (+ 5 + 6^2 - 8)^3 \sqrt{ 5 + 3 }$
 $\sqrt{(5 + 4 + 3 + 2^2) / (+ 5 + 6^2 - 8)^3}$

$8 + 5$
 $8 \sqrt{5}$
 $\sqrt{8}$

POSR (((5 \ 4 \ 3 \ (2 \ p^2) +)
 $((5 \ (6 \ p^2) - 8 +) \ p^3) /) \sqrt{ }$

5 + 3 Infix



Prefijo + 5 3
 Posfijo 3 5 +

VAMO A

