Clase 5

- Objetivos
  - Más programación en listas
  - Más acumuladores
  - Otras estructuras en Prolog

Append

- Definiremos un predicado muy usado: **append/3**
- append(L1,L2,L3) ← la lista L3 es la concatenación de las listas L1 y L2

```
?- append([a,b,c,d],[3,4,5],[a,b,c,d,3,4,5]).
yes

?- append([a,b,c],[3,4,5],[a,b,c,d,3,4,5]).
no
```

Append

- El uso más habitual de append/3 es obtener la concatenación de 2 listas
- Esto se hace instanciando 1er y 2do argumento y con el 3er argumento variable

```
?- append([a,b,c,d],[1,2,3,4,5], X).
X=[a,b,c,d,1,2,3,4,5]
yes

?-
```

Definición de append/3

```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```

- Definición recursiva: se “deshace” el 1er argumento
  - Cláusula base: la concatenación de la lista vacía y cualquier lista L es L.
  - Cláusula recursiva: append([H|T],L1,L2) en función de append(T,L1,L3)

Cómo funciona append

– Arbol de búsqueda

?- append([a,b,c],[1,2,3], R).

Árbol de búsqueda

?- append([a,b,c],[1,2,3], R).

```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```

Árbol de búsqueda

?- append([a,b,c],[1,2,3], R).

/ \

```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```

Árbol de búsqueda

?- append([a,b,c],[1,2,3], R).

/ \

† R = [a|L0]

?- append([b,c],[1,2,3],L0)

```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```

Árbol de búsqueda

?- append([a,b,c],[1,2,3], R).

/ \

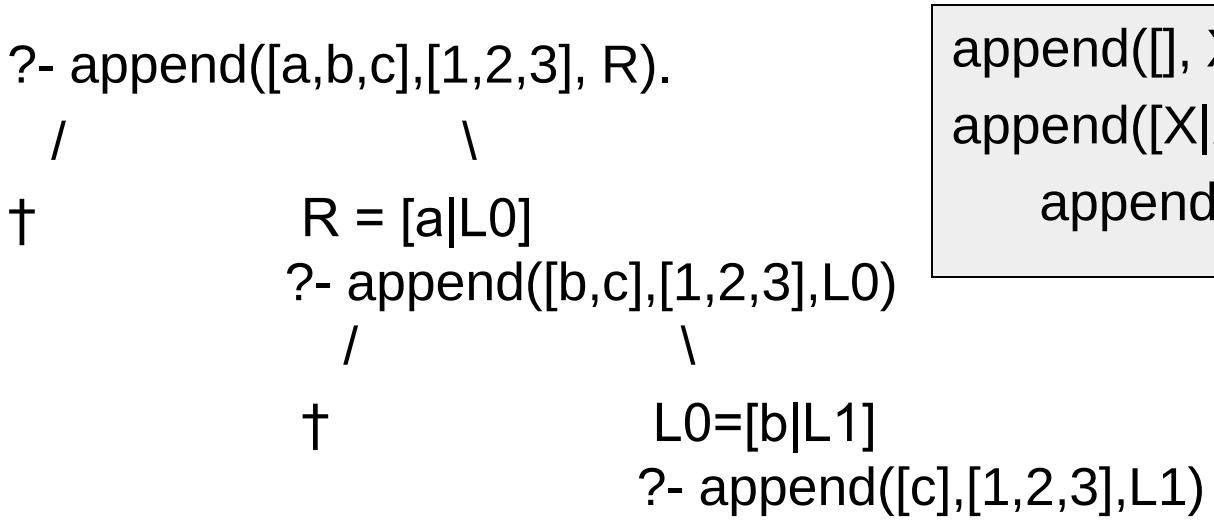
† R = [a|L0]

?- append([b,c],[1,2,3],L0)

/ \

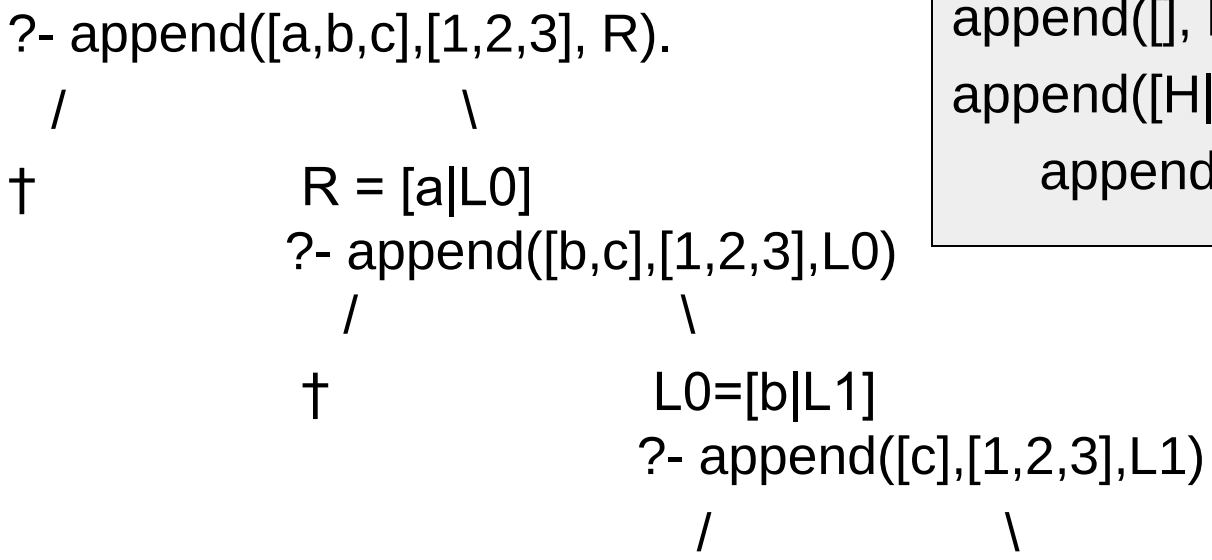
```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```

Árbol de búsqueda




```
append([], X, X).
append([X|Xs], Y, [X|Z]):-
    append(Xs, Y, Z).
```

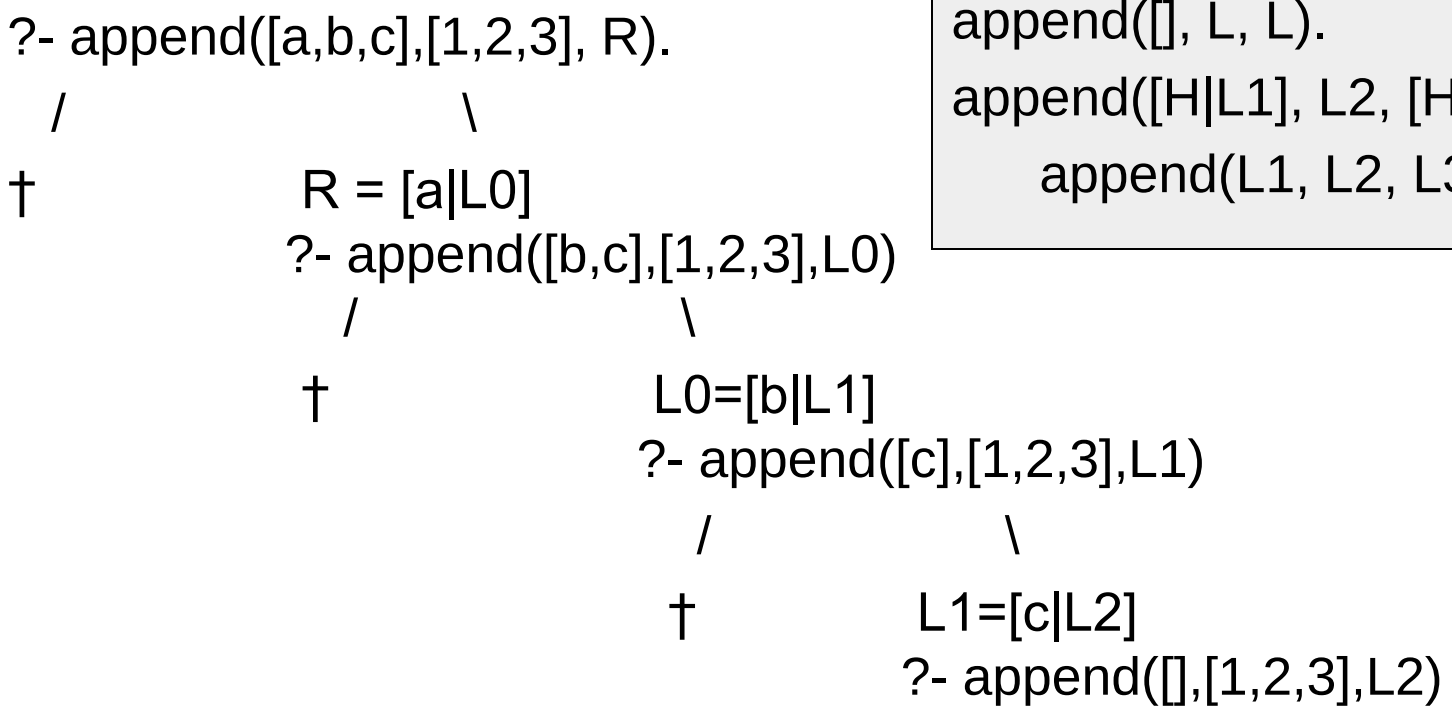
Árbol de búsqueda




```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```

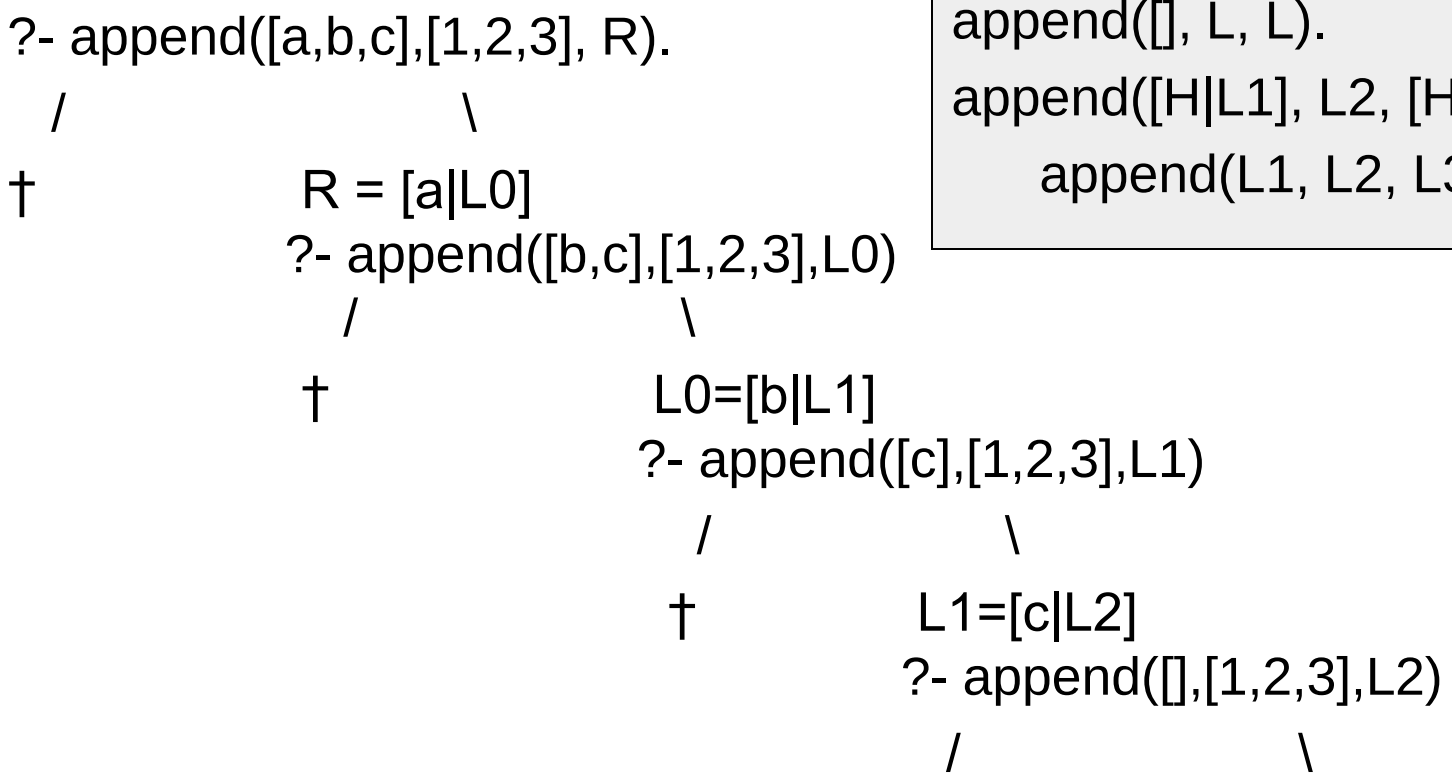
Árbol de búsqueda

```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```



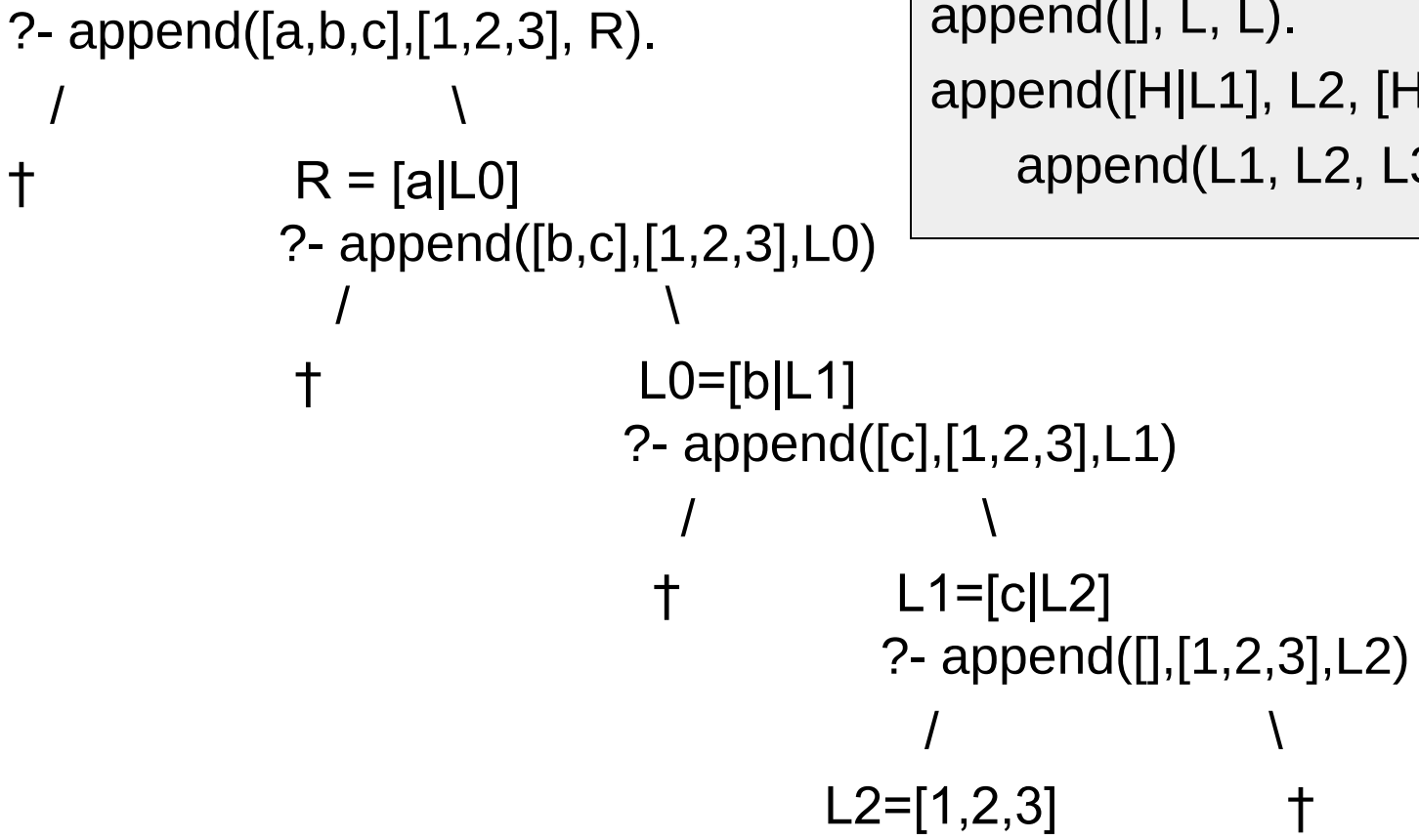
Árbol de búsqueda

```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```



Árbol de búsqueda

```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```



Árbol de búsqueda

```
append([], L, L).
append([H|L1], L2, [H|L3]):-
    append(L1, L2, L3).
```

?- append([a,b,c],[1,2,3], R).

/ \

† R = [a|L0]

?- append([b,c],[1,2,3],L0)

/ \

† L0=[b|L1]

?- append([c],[1,2,3],L1)

/ \

† L1=[c|L2]

?- append([],[1,2,3],L2)

/ \

L2=[1,2,3] †

L2=[1,2,3]

L1=[c|L2]=[c,1,2,3]

L0=[b|L1]=[b,c,1,2,3]

R=[a|L0]=[a,b,c,1,2,3]

Usos de append/3

- Partir una lista en la concatenación de otras dos

```
?- append(X,Y, [a,b,c,d]).
X=[ ]          Y=[a,b,c,d];
X=[a]          Y=[b,c,d];
X=[a,b]        Y=[c,d];
X=[a,b,c]      Y=[d];
X=[a,b,c,d]    Y=[ ];
no
```

Prefijo y sufijo

- Sea L=[a1,a2,…,ai,…an]

- Prefijo de L :
  - lista de la forma [a1,…,aj] 1≤ j ≤ n
  - []
- Sufijo de L :
  - lista de la forma [aj,…,an] 1≤ j ≤ n
  - []

Definición of prefijo/2

```
prefijo(P,L):-
    append(P,_,L).
```

- Una lista P es un prefijo de una lista L si existe otra lista tal que P concatenada con esa lista da L
- Notar el uso de la variable anónima

Consultando prefijo/2

```
prefijo(P,L):-
    append(P,_,L).
```

```
?- prefijo(X, [a,b,c,d]).
X=[ ];
X=[a];
X=[a,b];
X=[a,b,c];
X=[a,b,c,d];
no
```

Definición of sufijo/2

```
sufijo(S,L):-
    append(_,S,L).
```

Consultando sufijo/2

```
sufijo(S,L):-
    append(_,S,L).
```

```
?- sufijo(X, [a,b,c,d]).
X=[a,b,c,d];
X=[b,c,d];
X=[c,d];
X=[d];
X=[];
no
```

Definición de sublista/2

- Predicado que encuentra sublistas de una lista
- Las sublistas de una lista L son simplemente los prefijos de los sufijos.

```
sublista(Sub,Lista):-
    sufijo(Sufijo,Lista),
    prefijo(Sub,Sufijo).
```

Definición de sublista/2

- Hay otros modos
  - Con sufijo y prefijo
  - Directamente

Patrones de instanciación

```
append(+L1,+L2,?L3)    -- ok
append(?L1,?L2,+L3)    -- ok
Qué pasa con
append(?L1,?L2,?L3)
```

Patrones de instanciación

append(+L1,+L2,?L3) -- ok
append(?L1,?L2,+L3) -- ok
Qué pasa con
append(?L1,?L2,?L3)

```
?- append(X,Y,Z).
X = [],
Y = Z ;
X = [_G362],
Z = [_G362|Y] ;
X = [_G362, _G368],
Z = [_G362, _G368|Y] ;
X = [_G362, _G368, _G374],
Z = [_G362, _G368, _G374|Y] ;
X = [_G362, _G368, _G374, _G380],

Z = [_G362, _G368, _G374, _G380|Y]
```

Patrones de instanciación

sufijo(Suf,L)
prefijo(Pref,L)
subLista(Sub,L)

append/3 y eficiencia

- El predicado **append/3** es útil, y es importante saber como usarlo
- Es también importante saber que **append/3** puede ser una fuente de ineficiencia
- Por qué?
  - La concatenación de listas no se hace en un paso
  - Es necesario recorrer, elemento a elemento, una de las listas (se verá más adelante un modo de concatenar listas en un solo paso).

Pregunta

- Usando **append/3** queremos concatenar dos listas, sin que nos preocupe el orden de los elementos
  - List 1: [a,b,c,d,e,f,g,h,i]
  - List 2: [j,k,l]
- Cuál de los siguientes objetivos es más eficiente

```
?- append([a,b,c,d,e,f,g,h,i],[j,k,l],R).
?- append([j,k,l],[a,b,c,d,e,f,g,h,i],R).
```

Respuesta

- Recursión en el 1er argumento, nunca se toca el 2do.
- Lo mejor es entonces usar la lista más corta como 1er argumento
- No siempre sabremos cuál es la más corta, y muchas veces es importante el orden

Reverso de una lista

- Ilustramos el problema con la eficiencia de append/3 usándolo para obtener el reverso de una lista.
- Lo usaremos para definir un predicado que transforma [a,b,c,d,e] en [e,d,c,b,a]
- Esta herramienta puede ser útil, ya que Prolog solo brinda acceso fácil al 1er elemento de una lista.

Naïve reverse

- Definición recursiva
  1. El reverse de la lista vacía es la lista vacía
  2. El reverse de la lista [H|T], es la concatenación del reverse de la lista T con la lista cuyo único elemento es H

- Ilustración: considere la lista [H|T]=[a,b,c,d].
  - El reverse de T es [d,c,b].
  - [d,c,b] concatenado con [a] da [d,c,b,a]

Naïve reverse en Prolog

```
naiveReverse([],[]).
naiveReverse([H|T],R):-
    naiveReverse(T,RT),
    append(RT,[H],R).
```

- La definición es correcta, pero:
- La ejecución usa mucho tiempo en realizar appends
- Existe un modo mejor…

Reverse con acumulador

- El modo mejor es con acumulador
- El acumulador es una lista donde se construye el resultado
- Cada elemento de la lista se coloca en la cabeza del acumulador, y este se pasa como argumento en la recursión
- Al llegar a la lista vacía el acumulador contiene el reverse de la lista original!

Reverse con acumulador

```
accReverse([ ],L,L).
accReverse([H|T],Acc,Rev):-
    accReverse(T,[H|Acc],Rev).
```

Reverse con wrapper

```
accReverse([ ],L,L).
accReverse([H|T],Acc,Rev):-
    accReverse(T,[H|Acc],Rev).
```

```
reverse(L1,L2):-
    accReverse(L1,[ ],L2).
```

Ilustrando el acumulador

- Lista: [a,b,c,d] Acumulador: []

Ilustrando el acumulador

- Lista: [a,b,c,d] Acumulador: []
- Lista: [b,c,d] Acumulador: [a]

Ilustrando el acumulador

- Lista: [a,b,c,d] Acumulador: []
- Lista: [b,c,d] Acumulador: [a]
- Lista: [c,d] Acumulador: [b,a]

Ilustrando el acumulador

- Lista: [a,b,c,d]
- Lista: [b,c,d]
- Lista: [c,d]
- Lista: [d]

Acumulador: []

Acumulador: [a]

Acumulador: [b,a]

Acumulador: [c,b,a]

Ilustrando el acumulador

- Lista: [a,b,c,d]
- Lista: [b,c,d]
- Lista: [c,d]
- Lista: [d]
- Lista: []

Acumulador: []

Acumulador: [a]

Acumulador: [b,a]

Acumulador: [c,b,a]

Acumulador: [d,c,b,a]

Arboles en Prolog

Definiremos una estructura de datos para árboles binarios.

El árbol vacío es un árbol binario

Si X es un elemento y A1 y A2 son árboles binarios
arbol(X,A1,A2) es un árbol binario

Un árbol se representa por un único término complejo !!!

Arboles en Prolog

Predicados

arbol_binario(A) ← A es un arbol binario

pertenece_arbol(X,A) ← X es un elemento del árbol binario A

inorder(A,L) ← la lista L es el resultado de recorrer en inorden (raíz, izq.,der.) el árbol A

Resumen

- Predicado muy usado: **append/3**
- Más aplicaciones de acumuladores
- Otros predicados sobre listas
- Otras estructuras de datos (árbol binario)

Próximas clases

- Fundamentos teóricos
  - Repaso lógica proposicional y de 1er orden
  - Semántica, modelos
  - Forma Clausal
  - Inferencia, regla de resolución
  - Conceptos fundamentales
    - Consistencia
    - Completitud
    - Satisfactibilidad

Ejercicios

1. LPN Ejercicio 6.1
   - Una lista es una duplicación si está formada por dos bloques iguales de elementos,ej. [a,b,b,a,b,b] es una duplicación.
   - Escriba un predicado duplicación(L) que es verdadero cuando la condición dé duplicación se cumple.

2. LPN Ejercicio 6.2
   - Escriba un predicado palindrome(X), verdadero si la lista X se lee igual de izquierda a derecha que de derecha a izquierda.

3. Defina el predicado member(X,L) usando append