∀t :: AT a . ∀x :: a . (elem x (preorder t) = elem x (postorder t))

Tenemos que t es un arbol ternario, podemos usar el principio de inducción sobre esta estructura:

```
data AT a = Nil | Tern a (AT a) (AT a) (AT a) deriving Eq
```

Lo planteamos como un predicado unario:

```
P(t) = elem x (preorder t) = elem x (postorder t)
```

Sabiendo el tipo de dato, podemos ver que el caso base es el Nil, mientras que el Tern es el caso inductivo. Primero demostremos esta propiedad para el caso base.

Caso base: Nil

Primero recordemos la implementación de las funciones presentes:

```
foldAT :: (a -> b -> b -> b -> b) -> b -> AT a -> b
foldAT cTern cNil at = case at of
  Nil -> cNil
  (Tern r i c d) -> cTern r (recu i) (recu c) (recu d)
  where recu = foldAT cTern cNil

preorder :: Procesador (AT a) a
preorder = foldAT (\rr ri rc rd -> rr:(ri ++ rc ++ rd)) []

postorder :: Procesador (AT a) a
postorder = foldAT (\rr ri rc rd -> (ri ++ rc ++ rd) ++ [rr]) []

elem :: Eq a => a -> [a] -> Bool
elem e [] = False
elem e (x:xs) = (e == x) || elem e xs
```

En este caso t es Nil, lo que nos dejaría con:

```
elem x (preorder Nil) = elem x (postorder Nil)
```

Ahora desarrollemos las funciones de preorder y postorder:

```
elem x (foldAT (\rr ri rc rd -> rr:(ri ++ rc ++ rd)) [] Nil) = elem x (foldAT (\rr
ri rc rd -> (ri ++ rc ++ rd) ++ [rr]) [] Nil)
```

Vemos que en ambos casos se utiliza un foldAT, el cual según su definición se reduce al segundo argumento que se le es enviado cuando su tercer argumento es Nil (basicamente se reduce a lista vacía).

```
elem x [] = elem x []
```

Aplicamos la definición de elem, a ambos lados se reduce a false porque la lista es vacía.

```
False = False
```

Caso inductivo: Tern

En este caso t es Tern a (AT a) (AT a) (AT a), haremos el siguiente cambio para este paso de la demostración, diremos que la propiedad vale para todo "i", "c" y "d" de tipo "AT a" y también para un e cuyo tipo sea a. Es decir:

```
∀i :: AT a . ∀c :: AT a . ∀d :: AT a . ∀i :: AT a . ∀x :: a . ∀e :: a . (elem
x (preorder (Tern e i c d)) = elem x (postorder (Tern e i c d)))
```

Primero tengamos en cuenta que por la Hipotesis inductiva la propiedad que queremos demostrar vale para i, c y d.

HI: P(i) ∧ P(c) ∧ P(d)

Que sería lo mismo que decir que lo siguiente vale, siendo t en ese caso i, c o d.

```
∀t :: AT a . ∀x :: a . ( elem x (preorder t) = elem x (postorder t) )
```

Ahora podemos ver que es lo que hacen la función preorder y postorder con el arbol.

```
elem x (foldAT (\rr ri rc rd -> rr:(ri ++ rc ++ rd)) [] (Tern e i c d)) = elem x
(foldAT (\rr ri rc rd -> (ri ++ rc ++ rd) ++ [rr]) [] (Tern e i c d))
```

En este caso vemos que por defínción del foldAT, se le debe aplicar la función lambda descrita a los parametros.

Para simplificar los siguientes pasos solo vamos a desarrollar la parte izquierda de la igualdad.

```
elem x (e:((recu i) ++ (recu c) ++ (recu d)))
    where recu = foldAT (\rr ri rc rd -> rr:(ri ++ rc ++ rd)) []
```

Pero ahora podriamos hacer una simplificación viendo que recu es exactamente la definición de preorder, por lo cual lo vuelvo a reemplazar.

```
elem x (e:((preorder i) ++ (preorder c) ++ (preorder d)))
```

Antes de avanzar aplicando la definición de elem, veamos como desarrollar el lado derecho del igual.

```
elem x (foldAT (\rr ri rc rd -> (ri ++ rc ++ rd) ++ [rr]) [] (Tern e i c d))
```

Aplicamos la definición de foldAT con lo que dice la función lambda:

```
elem x ((recu i) ++ (recu c) ++ (recu d)) ++ [e]
    where recu = foldAT (\rr ri rc rd -> (ri ++ rc ++ rd) ++ [rr]) []
```

Similar al caso anterior, vemos que recu es la definción de postorder, por lo que podemos simplificarlo reemplazando nuevamente.

```
elem x ((postorder i) ++ (postorder c) ++ (postorder d)) ++ [e]
```

Ahora volvemos a la igualdad:

```
elem x (e:((preorder i) ++ (preorder c) ++ (preorder d))) = elem x ((postorder i)
++ (postorder c) ++ (postorder d)) ++ [e]
```

Demostración de lemas

Para avanzar veamos de demostrar un par de lemas:

```
1: ∀e :: a . ∀x :: a . ∀xs :: [a] . (elem x e:xs = elem x (xs ++ [e]))

2: ∀e :: a . ∀x :: a . ∀xs :: [a] . ∀ys :: [a]  (elem x xs || elem x ys = elem
x (xs ++ ys))
```

El primero nos serviría para colocar el elemento "e" al principio de todo para el caso de postorder. El segundo nos ayudaría a separar las listas concatenadas para poder aplicarles la Hipotesis Inductiva. Empecemos por el primer lema.

Primer lema

```
elem x e:xs = elem x (xs ++ [e])
```

Caso base, xs = []

Entonces reemplazamos xs por [].

```
elem x e:[] = elem x ([] ++ [e])
```

Aplicamos la definición del constructor de listas y de la concatenación.

```
elem x [e] = elem x [e]
```

Vemos que a ambos lados del igual tenemos la misma expresión, por lo que queda demostrado el caso base por el principio de extensionalidad funcional. (Ya que a ambos lados del igual las expresiones son iguales punto a punto.)

Paso inductivo.

Ahora demostramos para el caso general.

```
elem x e:(z:xs) = elem x ((z:xs) ++ [e])
```

Sabiendo que por Hipotesis inductiva vale que:

```
elem x e:xs = elem x (xs ++ [e])
```

Aplicamos la definición de elem a la derecha de la igualdad.

```
elem x e:(z:xs) = (z == e) || elem x (xs ++ [e])
```

Ahora aplicamos extensionalidad sobre booleanos en "z".

Caso "z != e"

En este caso la comparación se reducirá a False.

```
elem x e:(z:xs) = False || elem x (xs ++ [e])
```

Si tenemos un False en una disyunción sabemos que el valor de verdad queda determinado por el resto de la expresión. Por lo que podemos ignorarlo.

```
elem x e:(z:xs) = elem x (xs ++ [e])
```

Entonces vemos ahora que por hipotesis inductiva, lo que tenemos a la derecha de la igualdad es igual a elem x e:xs. Entonces reemplazamos.

```
elem x e:(z:xs) = elem x e:xs
```

Ahora debemos aplicar elem haciendo extensionalidad sobre booleanos para la comparación con e.

Caso "e = x"

```
-- Aplicamos def. de elem.
(e == x) || elem x z:xs = (e == x) || elem x xs
-- Esto se valua a True
True || elem x z:xs = True || elem x xs
-- Sabemos que por propiedad de disyunción, al tener un True el valor de la
expresión será True, por lo que se reduce todo a True.
True = True
```

Caso "e != x"

```
-- Aplicamos def. de elem.
(e == x) || elem x z:xs = (e == x) || elem x xs
-- Esto se valua a False
False || elem x z:xs = False || elem x xs
-- Si tenemos un False en una disyunción sabemos que el valor de verdad queda
determinado por el resto de la expresión. Por lo que podemos ignorarlo.
elem x z:xs = elem x xs
-- Estamos en el caso donde "z != x", por lo que el caso de la izquierda se reduce
de la siguiente forma.
(z == x) || elem x xs = elem x xs
-- Se reduce a False, que por propiedad de disyunción ignoramos.
elem x xs = elem x xs
```

Vemos que a ambos lados del igual tenemos la misma expresión, entonces para el caso en el que "z != x" la propiedad se cumple.

Caso "z = e"

En este caso la comparación se reducirá a True.

```
elem x e:(z:xs) = True || elem x (xs ++ [e])
```

Sabemos que por propiedad de disyunción, al tener un True el valor de la expresión será True, por lo que se reduce todo a True.

```
elem x e:(z:xs) = True
```

Para el lado izquierdo de la igualdad, debemos aplicar nuevamente extensionalidad sobre booleanos en e.

Caso "x = e"

```
-- Aplicamos def. de elem.
(e == x) || elem x z:xs = True
-- Se reduce a True
True || elem x z:xs = True
-- Por propiedad de la disyunción, todo se reduce a True
True = True
```

Caso "x != e"

```
-- Aplicamos def. de elem.
(e == x) || elem x z:xs = True
-- Se reduce a False
False || elem x z:xs = True
-- Por propiedad de la disyunción, podemos ignorar y eliminar ese False
elem x z:xs = True
-- Aplicamos def. de elem.
(z == x) || elem x xs = True
-- Estamos en el caso de z = x, por lo que se reduce a True
True || elem x xs = True
-- Por propiedad de disyunción reduce a True
True = True
```

Asi queda demostrado el primer lema.

Segundo lema

```
2: ∀e :: a . ∀x :: a . ∀xs :: [a] . ∀ys :: [a]  (elem x xs || elem x ys = elem
x (xs ++ ys))
```

Para este lema debemos hacer inducción estructural sobre la lista xs. Empiezo con el caso base.

Caso base: "xs = []"

```
elem x [] || elem x ys = elem x ([] ++ ys)
```

Aplico definición de elem a la izquierda del igual mientras que a la derecha aplico la concatenación.

```
False || elem x ys = elem x ys
-- Por propiedad de la disyunción, puedo eliminar el False.
elem x ys = elem x ys
```

A ambos lados del igual tengo la misma expresión, el caso base queda demostrado.

Caso inductivo.

El caso inductivo es el siguiente.

```
elem x (z:xs) || elem x ys = elem x ((z:xs) ++ ys)
```

Sabiendo que por hipotesis inductiva se cumple esto:

```
elem x xs || elem x ys = elem x (xs ++ ys)
```

Entonces puedo aplicar definición de elem a ambos lados del igual. (En el lado de la izquierda solo se lo aplico al elem cuya lista es z:xs)

```
(z == x) || elem x xs || elem x ys = (z == x) || elem x (xs ++ ys)
```

Ahora realizamos extensionalidad sobre booleanos para analizar si z es igual a x.

Caso "z = x"

En este caso, como son iguales su comparación se reduce a True

```

```

```
True || elem x xs || elem x ys = True || elem x (xs ++ ys)
-- Por propiedad de la disyunción, a ambos lados de la igualdad esto reduce a True
True = True
```

Es True a ambos lados, se cumple la igualdad.

Caso "z != x"

En este caso la comparación se reduce a False

```
False || elem x xs || elem x ys = False || elem x (xs ++ ys)
-- Por propiedad de la disyunción, podemos eliminar los False
elem x xs || elem x ys = elem x (xs ++ ys)
```

Vemos que esto es exactamente lo que sabemos que vale por hipotesis inductiva, por lo que el lema queda demostrado para todos los casos.

Conclusión de la demostración

Volvemos a la expresión original la cual queriamos demostrar.

```
elem x (e:((preorder i) ++ (preorder c) ++ (preorder d))) = elem x ((postorder i)
++ (postorder c) ++ (postorder d)) ++ [e]
```

Para solucionar este problema contamos ahora con estos lemas.

```
1: ∀e :: a . ∀x :: a . ∀xs :: [a] . (elem x e:xs = elem x (xs ++ [e]))

2: ∀e :: a . ∀x :: a . ∀xs :: [a] . ∀ys :: [a]  (elem x xs || elem x ys = elem
x (xs ++ ys))
```

Primero aplicamos el primer lema en el lado derecho de la igualdad (ya que la concatenación de los postorder da como resultado otra lista, la cual en el lema se contempla como xs) y lo reemplazamos teniendo el elemento "e" al principio de todo.

```
elem x (e:((preorder i) ++ (preorder c) ++ (preorder d))) = elem x (e:((postorder
i) ++ (postorder c) ++ (postorder d)))
```

Ahora podemos aplicar definición de elem teniendo en ambas listas a "e" como primer elemento.

```

```

```
(e == x) || elem x ((preorder i) ++ (preorder c) ++ (preorder d)) = (e == x) elem
x ((postorder i) ++ (postorder c) ++ (postorder d))
```

Ahora hacemos extensionalidad sobre booleanos para chequear los casos en el que "e = x" y "e != x"

Caso: "e = x"

En este caso la comparación se reduce a True.

```
True || elem x ((preorder i) ++ (preorder c) ++ (preorder d)) = True || elem x
((postorder i) ++ (postorder c) ++ (postorder d))
-- Por propiedad de la disyunción, podemos desestimar el resto de la lista, el
resultado a ambos lados es True
True = True
```

Caso: "e != x"

En este caso la comparación se reduce a False.

```
False || elem x ((preorder i) ++ (preorder c) ++ (preorder d)) = False || elem x
((postorder i) ++ (postorder c) ++ (postorder d))
```

Por propiedad de la disyunción, el valor de verdad depende del resto de la expresión, entonces eliminamos el False.

```
elem x ((preorder i) ++ (preorder c) ++ (preorder d)) = elem x ((postorder i) ++
(postorder c) ++ (postorder d))
```

Y acá es donde aplicamos el segundo lema que demostramos. Podemos agarrar una de las listas de preorder y las otras dos restantes de preorder como si fueran nuestras xs e ys respectivamente del segundo lema. Entonces, al aplicar el reemplazo queda así:

```
elem x (preorder i) || elem x ((preorder c) ++ (preorder d)) = elem x ((postorder
i) ++ (postorder c) ++ (postorder d))
```

Y ahora bien podemos volver a aplicar el lema en los preorder que aun siguen concatenados.

```
elem x (preorder i) || elem x (preorder c) || elem x (preorder d) = elem x
((postorder i) ++ (postorder c) ++ (postorder d))
```

Ahora por la hipotesis inductiva (la vuelvo a copiar):

```
∀t :: AT a . ∀x :: a . ( elem x (preorder t) = elem x (postorder t) )
-- Siento t en este caso el arbol i, c o d.
```

Vemos que podemos reemplazar (elem x (preorder i)) por (elem x (postorder i)), y lo mismo hacemos para el arbol c y d.

```
elem x (postorder i) || elem x (postorder c) || elem x (postorder d) = elem x
((postorder i) ++ (postorder c) ++ (postorder d))
```

Ahora puedo aplicar el segundo lema devuelta para desandar el camino que nos llevó a la disyunción de los elem. Haciendo eso volvemos en primer lugar a unir las listas de postorder c con postorder d.

```
elem x (postorder i) || elem x ((postorder c) ++ (postorder d)) = elem x
((postorder i) ++ (postorder c) ++ (postorder d))
```

Y lo hacemos una vez mas para unir a la lista postorder i con el resto

```
elem x ((postorder i) ++ (postorder c) ++ (postorder d)) = elem x ((postorder i)
++ (postorder c) ++ (postorder d))
```

Y ahora finalmente vemos que a ambos lados de la igualdad tenemos exactamente la misma expresión, por lo que por el principio de extensionalidad funcional, al probar que las funciones son iguales punto a punto, esta igualdad queda demostrada.