Práctica 3

David Rivera Morales -- 320176876

Israel Rivera -- 320490747

La función longP

Dentro de esta función el caso base es una lista vacía que da 0 ya que una lista sin elementos , no tiene longitud por lo que es 0. En el caso de la recursión lo que hice fue que si la lista no es vacía entonces se le suma 2 a la longitud de la lista que se obtiene al quitar la primer sublista de la lista original. Esto se hace ya que la lista que se obtiene al quitar la primer sublista es una lista que tiene una longitud menor que la lista original y por lo tanto se le suma 2 a la longitud de la lista que se obtiene al quitar la primer sublista.

```
longP :: EList a -> Int
longP [] = 0
longP ((x,y):xs) = 2 + longP xs
```

La función elemP

Dentro de esta función el caso base es una lista vacía que da False ya que una lista sin elementos , no tiene elementos por lo que es False. En la recursión lo que hice fue comparar el elemento tanto con el primer valor de la sublista como con el segundo valor de la sublista y si alguno de los dos es igual al elemento que se busca entonces se regresa True, de lo contrario se llama a la función con la cola de la lista.

```
elemP :: Eq a => a -> EList a -> Bool
elemP e [] = False
elemP e ((x,y):xs) = e == x || e == y || elemP e xs
```

La función consP

En esta función el caso base es cuando se le quisieran agregar dos elementos a una lista vacía, por lo que se regresa una lista con una sublista que contiene los dos elementos que se quisieron agregar. En el caso de la recursión se agregan los dos elementos a la cabeza de la lista que se obtiene al quitar la primer sublista de la lista original.

```
consP :: a -> a -> EList a -> EList a
const e1 e2 [] = [(e1,e2)]
consP e1 e2 ((x,y):xs) = (e1,e2):(x,y):xs
```

La función appendP

El caso base de esta función está dado por una lista vacía a la que se le quiere agregar otra lista , por lo cual se regresa la lista que no es vacía. En el caso de la recursión se agrega la primer sublista de la lista que se quiere agregar a la cabeza de la lista que se obtiene al quitar la primer sublista de la lista original.

```
appendP :: EList a -> EList a -> EList a
appendP [] l = l
appendP ((x,y):xs) l = (x,y):appendP xs l
```

La función snocP

En esta función el caso base es cuando se le quieren agregar dos elementos a una lista vacía, por lo que se regresa una lista con una sublista que contiene los dos elementos que se quisieron agregar. En el caso de la recursión se agrega la primer sublista de la lista original a la cabeza de la lista que se obtiene al agregar los dos elementos al final de la lista que se obtiene al quitar la primer sublista de la lista original.

```
snocP :: (a,a) -> EList a -> EList a
snocP (e1,e2) [] = [(e1,e2)]
snocP (e1,e2) ((x,y):xs) = (x,y):snocP (e1,e2) xs
```

La función atP

En este caso debe de haber tres casos base , el primero es cuando la lista es vacía , por lo que se regresa un error. El segundo caso es cuando la posición es 0 , por lo que se regresa la primer sublista de la lista original. El tercer caso es cuando la posición es 1 , por lo que se regresa la segunda sublista de la lista original. En el caso de la recursión se llama a la función con la cola de la lista y la posición -2.

```
atP [] n = error "No existe el elemento"
atP ((x,y):xs) 1 = x
atP ((x,y):xs) 2 = y
atP ((x,y):xs) n = atP xs (n-2)
```

La función updateP

Aquí puede suceder que se quiera actualizar una posición que no existe , por lo que se debe de tener un caso base que sea cuando la lista es vacía, en este caso se regresa un error ya que no se puede actualizar una posición que no existe. En el caso de la recursión se agrega la primer sublista de la lista original a la cabeza de la lista que se obtiene al actualizar la posición -2 de la lista que se obtiene al quitar la primer sublista de la lista original.

```
updateP :: EList a -> Int -> a -> EList a
updateP [] n e = error "No existe el elemento"
updateP ((x,y):xs) 1 e = (e,y):xs
updateP ((x,y):xs) 2 e = (x,e):xs
updateP ((x,y):xs) n e = (x,y):updateP xs (n-2) e
```

La función aplanaP

En este caso el caso base es cuando la lista es vacía, por lo que se regresa una lista vacía. En el caso de la recursión se agrega la primer sublista de la lista original a la cabeza de la lista que se obtiene al aplicar aplanaP a la cola de la lista original.

```
aplanaP :: EList a -> [a]
aplanaP [] = []
aplanaP ((x,y):xs) = x:y:aplanaP xs
```

La función toEl

```
toEl :: [a] -> EList a
toEl [] = []
toEl [x] = []
toEl (x:y:xs) = (x,y):toEl xs
```

La función dropP

En este caso el caso base es cuando la lista es vacía o cuando la posición es 0, por lo que se regresa la lista original. En el caso de la recursión se llama a la función con la cola de la lista y la posición -2 hasta que la posición sea 0.

```
dropP :: Int -> EList a -> EList a
dropP 0 l = l
dropP n [] = []
dropP n ((x,y):xs) = if (mod n 2 == 0) then dropP (n-2) xs else error "No
se puede borrar un número impar de elementos"
```

La función dropN

Dentro de esta función , el caso base es cuando la lista es vacía o cuando la posición es 0, por lo que se regresa la lista original pero aplanada en caso de que tenga elementos y también hay un caso base en el que la posición es 1 donde se regresa la lista con el primer elemento eliminado. En el caso de la recursión se llama a la función con la cola de la lista y la posición -2 hasta que la posición sea 0.

```
dropN :: Int -> EList a -> [a]
drop  n [] = []
dropN 0 l = aplanaP l
dropN 1 ((x, y) : xs) = y : aplanaP xs
dropN n ((x, y) : xs) = dropN (n - 2) xs
```

La función takeP

En este caso el caso base es cuando la lista es vacía o cuando la posición es 0, por lo que se regresa una lista vacía. En el caso de la recursión se agrega la primer sublista de la lista original a la cabeza de la lista que se obtiene al llamar a la función con la cola de la lista y la posición -2 hasta que la posición sea 0 solo si la posición es par , ya que si es impar no se puede tomar un número impar de elementos.

```
takeP :: Int -> EList a -> EList a
takeP 0 l = []
takeP n [] = []
takeP n ((x, y) : xs) = if (mod n 2 == 0) then (x, y) : takeP (n - 2) xs
else error "No se puede tomar un número impar de elementos"
```

La función takeN

El caso base es cuando la lista es vacía o cuando la posición es 0, por lo que se regresa una lista vacía pero hay otro caso que ayuda donde la posición es 1 donde se regresa una lista con el primer elemento de la lista original. En el caso de la recursión se agrega el primer elemento de la primer sublista junto con el segundo elemento de la primer sublista a la cabeza de la lista que se obtiene al llamar a la función con la cola de la lista y la posición -2 hasta que la posición sea 0.

```
takeN :: Int -> EList a -> [a]
takeN 0 l = []
takeN 1 ((x, y) : xs) = [x]
takeN n ((x, y) : xs) = x : y : takeN (n - 2) xs
```

La función reversaP

Dentro de esta función el caso base es cuando la lista es vacía, por lo que se regresa una lista vacía. En el caso de la recursión , con la función snocP se agrega la primer sublista de la lista original a la cola de la lista que se obtiene al llamar a la función con la cola de la lista original.

```
reversaP :: EList a -> EList a
reversaP [] = []
reversaP ((x, y) : xs) = snocP (y, x) (reversaP xs)
```