Algoritmos y Estructuras de Datos 2

Final 2021-04-21

1. Ejercicio 1

1.a.

- No es necesario el generador `pase_el_que_sigue` ya que esto sucede como parte del generador `atender_cliente`. Es decir, cuando se atiende al primer cliente de la cola, automáticamente pasa el siguiente si es que hay. Si planteamos este generador, estamos delegando al usuario lo que se pide como un comportamiento automático, permitiendo así instancias inválidas del TAD (hay clientes en la cola pero nadie está siendo atendido).
- Como el generador `llega_cliente` recibe un cliente, el observador `largo_de_la_cola` debería ser `cola: banco` $\rightarrow$ `cola(cliente)` para poder identificar los clientes en la cola y su posición. De la forma planteada, distintas colas del mismo largo son observacionalmente iguales aunque hayan distintos clientes en ella. Otra solución es simplificar el TAD y sacar el parámetro `cliente` del generador `llega_cliente` para modelar así únicamente el aspecto cuantitativo de una cola del banco (cuántos clientes hay) sin importar quiénes están en la cola. No está precisamente *mal* que reciba el parámetro `cliente` pero la realidad es que así como está planteado el TAD no se usa para nada.

1.b.

- El TAD no tiene problemas que impiden su uso.
- Está bien que `atender_cliente` sea otra operación ya que de esta forma los generadores son minimales. Cada vez que se atiende un cliente, podemos generar una nueva instancia desde cero aplicando el generador `llega_cliente` unas `largo_de_la_cola(b) - 1` veces.
- Aplica la misma aclaración planteada en el segundo item del caso a).

1.c.

- En este caso necesitamos que `clientes_atendidos` sea un observador para poder trackear correctamente el historial de clientes. Como queremos el historial, no es suficiente con solo observar el estado actual de la cola, necesitamos también incluir en la igualdad observacional lo que pasó antes. Pues sino, podríamos tener 2 instancias observacionalmente iguales a partir de sus colas actuales, pero con historiales totalmente distintos.
- Aplica la misma aclaración planteada en el segundo item del caso a).

2. Ejercicio 2

3. Ejercicio 3

4. Ejercicio 4

5. Ejercicio 5

5.a.

CalcularMínimos(**in** a: ABB(nat)) → **out** res: lista(tupla ⟨ nat, nat ⟩)

```
1: res ← CrearLista()                                          ▷ O(1)
2: if a ≠ nil then
3:     CalcularMínimosAux(a, res)                              ▷ O(n)
4: end if
```

Complejidad: $O(n)$

CalcularMínimosAux(**in** a: ABB(nat), **in/out** res: lista(tupla ⟨ nat, nat ⟩)) → **out** minNodo: nat

```
1: minNodo ← a.clave
2: if a.izq ≠ nil then
3:     minNodo ← CalcularMínimosAux(a.izq, res)
4: end if
5: AgregarAtras(res, ⟨ a.clave, minNodo ⟩)
6: if a.der ≠ nil then
7:     CalcularMínimosAux(a.der, res)
8: end if
```

Complejidad: $O(n)$ pues se visita exactamente 1 vez cada nodo del árbol.

5.b.

Si el árbol estuviese balanceado (por ejemplo es un AVL), la complejidad sería la misma ya que sigue siendo necesario visitar 1 vez cada nodo para calcular el mínimo de ese subárbol.