Módulo Padrón

Se explica con: Padrón

Estructura de representación

```
padron se representa con estr
donde estr es tupla(
    porCódigo    : diccDigital(código, persona),
    porDNI       : diccLog(dni,persona),
    cantJubilados: nat,
    fechaActual: tupla(día:nat, año:nat)
)

persona es tupla(
    dni: nat,
    código:string,
    nombre: string
    díaNac: nat,
    añoNac: nat
)

dni es nat
código es string
```

Interfaz

pre:{día entre 1 y 365, año mayor a 2022, ...}
pos:{padron es el resultado del generador, ...}
Costo:
Aliasing:
Descripción: Instancia un sistema nuevo vacío
*** NuevoPadron(in fechaInicio:tupla(día,año)) -> padron**

pre:{la persona NO existe en el sistema, ...}
pos:{...}
Costo: O(l + log n)
Aliasing: ...
Descripción: Agrega una nueva persona al padrón
*** AgregarPersona(inout p: padron, in per: persona)**

pre:{la persona existe en el sistema, ...}
pos:{...}
Costo: O(l + log n)
Aliasing: ...
Descripción: Elimina una persona existente del padrón
*** BorrarPersona(inout p: padron, in código: string)**

pre:{Hay una persona en el sistema con ese código}
pos:{...}
Costo: O(l)
Aliasing: Se devuelve una referencia no modificable a la persona.
Descripción: Obtiene una persona del padrón dado el código.
*** BuscarPorCódigo(in p:padron, in código:string) -> persona**

pre:{Hay una persona en el sistema con ese dni}
pos:{...}
Costo: O(log n)
Aliasing: Se devuelve una referencia no modificable a la persona.
Descripción: Obtiene una persona del padrón dado el dni.
*** BuscarPorDNI(in p:padron, in dni:nat) -> persona**

pre:{...}
pos:{...}
Costo: O(n)
Aliasing: No
Descripción: Avanza a un día nuevo en el sistema.
*** AvanzarDía(inout p:padron)**

pre:{...}
pos:{...}
Costo: O(1)
Aliasing: No
Descripción: Devuelve la cantidad de personas que están en edad jubilatoria en este momento.
*** CuántosEnEdadJubilatoria(in p:padron) -> nat**

pre:{...}
pos:{...}
Costo: O(l + log n)
Aliasing: No
Descripción: Indica si la persona de código y dni indicados pertenece al sistema.
*** ExistePersona(in p:padron, in código:string, in dni:nat) -> bool**

Algoritmos

iNuevoPadron(in fechaInicio:tupla(día,año)) -> padron

...

```
iAgregarPersona(inout p: estr, in per: persona){
    Definir(p.porCódigo,per.código, per) // O(1)
    Definir(p.porDNI,per.dni, per) // O(log n + l) (acá le sumamos 'l', el costo de copiar la tupla)

    if(EdadActual(p.fechaActual, per.díaNac, per.añoNac) > 64){
        p.cantJubilados = p.cantJubilados + 1 // O(1)
    }
}
```

Complejidad: O(l + log n + 1) = O(l + log n)

```
iBorrarPersona(inout p: estr, in código: string){
    per = Significado(p.porCódigo, código) // O(1)
    if(EdadActual(p.fechaActual, per.díaNac, per.añoNac) > 64){
        p.cantJubilados = p.cantJubilados - 1 // O(1)
    }
    Borrar(p.porDNI, per.dni)   // O(log n)
    Borrar(p.porCódigo, código) // O(1)
}
```

Complejidad: O(l + 1 + log n + l) = O(l + log n)

```
iBuscarPorCódigo(in p:estr, in código:string) -> persona{
    return Significado(p.porCódigo, código)
```

```
}

iBuscarPorDNI(in p:estr, in dni:nat) -> persona{
    return Significado(p.porDNI, dni)
}

iAvanzarDía(inout p:estr){
    SumarUnDía(p.fechaActual) // O(1)
    it = CrearIt(p.porDNI)  // O(1), porque no me interesa el orden de los elementos

    while (HaySiguiente(it)){ // O(n)
        per = SiguienteSignificado(it) // O(1)
        edadActual = EdadActual(p.fechaActual, per.díaNac, per.añoNac)
        cumpleAños = CumpleAños(per.díaNac,p.fechaActual)

        if(cumpleAños AND edadActual==65){ // O(1) (esto quiere decir que acaba de cumplir 65)
            p.cantJubilados = p.cantJubilados + 1 // O(1)
        }
        Avanzar(it) // O(1)
    }
}
```

Complejidad: O(1 + n) = O(n)

```
iCuántosEnEdadJubilatoria(in p:estr) -> nat{
    return p.cantJubilados
}
```

Complejidad: O(1)

```
iExistePersona(in p:est, in código:string, in dni:nat) -> bool{
    if (Definido?(p.porDNI, dni) AND Definido?(p.porCódigo, código){ // O(l + log n)
        return (BuscarPorDNI(p,dni) == BuscarPorCódigo(p,código)) // O(l + log n)
    }
    return false
}
```

Complejidad: O(log n + l)

Variantes

1) Se quiere saber e O(1) la cantidad de de personas que tienen una edad determinada
2) Se quiere cambiar de día en O(m), donde m es la cantidad de personas que cumplen en el nuevo día.

A continuación sólo se indica lo que cambia con respecto a lo anterior.

```
padron se representa con estr
donde estr tupla(
    porCódigo    : diccDigital(código, persona),
    porDNI       : diccLog(dni,persona),
    personaEnConjCumples: diccLog(dni, itConjLineal(persona)),
    cantJubilados: nat,
    fechaActual: tupla(día:nat, año:nat),
    cantPorEdad: arreglo_dimensionable(nat),
    cumplenEnDía: arreglo_dimensionable(conjLineal(persona))
)
```

Algoritmos

```
iNuevoPadron(in fechaInicio:tupla(día,año)) -> padron
      ...
      (acá falta instanciar todos las componentes nuevos de la tupla)
      ...
```

```
iAgregarPersona(inout p: estr, in per: persona){
    Definir(p.porCódigo,per.código, per) // O(1)
    Definir(p.porDNI,per.dni, per) // O(log n + 1) (el 1 es contemplando la copia de la tupla)

    edadActual = EdadActual(p.fechaActual, per.díaNac, per.añoNac) // O(1)
    cantPorEdad[edadActual] = cantPorEdad[edadActual] + 1 // O(1)
    it = AgregarRápido(cumplenEnDía[per.díaNac], per) // O(1)
    Definir(p.personaEnConjCumples, per.dni, it) // O(log n + 1 ) (el '1' es por la copia del iterador)

    if(edadActual > 64){
        p.cantJubilados = p.cantJubilados + 1 // O(1)
    }
}
```

Complejidad: O(l + log n + 1 + log n + 1) = O(l + log n)

```
iBorrarPersona(inout p: estr, in código: string){
    per = Significado(p.porCódigo, código) // O(1)

    edadActual = EdadActual(p.fechaActual, per.díaNac, per.añoNac)
    cantPorEdad[edadActual] = cantPorEdad[edadActual] - 1
    // Borrar(cumplenEnDía[per.díaNac], per) // O(n) (esta era la versión anterior, sin iteradores)
```

```
    it = Significado(p.personaEnConjCumples, per.dni) // O(log n) (Recuperamos el iterador que apunta
al elemento en el conjunto
    EliminarSiguiente(it) // O(1) (esto vale por la definición del iterador del diccionario lineal)

    edadActual = EdadActual(pe.dia,pe.año, p.fechaActual)
    if(edadActual > 64){ //Esto quiere decir que ya estaba contado para edad jubilatoria
        p.cantJubilados = p.cantJubilados - 1 // O(1)
    }
    Borrar(p.porDNI, per.dni)   // O(log n)
    Borrar(p.porCódigo, código) // O(l)
}
```

Complejidad: O(l + 1 + log n + l + log n + 1) = O(l + log n)

```
iAvanzarDía(inout p:estr) {
    SumarUnDía(p.fechaActual) // O(1)

    it = CrearIt(p.cumplenElDía[p.fechaActual]) // O(1)

    while (HaySiguiente(it)){ // O(m)
        per = Siguiente(it) // O(1)

        edadActual = EdadActual(pe.dia,pe.año, p.fechaActual) // O(1)
        cantPorEdad[edadActual] = cantPorEdad[edadActual] + 1 // O(1) - Se le suma 1 a la edad nueva
        cantPorEdad[edadActual-1] = cantPorEdad[edadActual] - 1 // O(1) - Se le resta 1 a la anterior

        if(edadActual == 65){ // O(1)
            p.cantJubilados = p.cantJubilados + 1  //O(1)
        }
        it.Avanzar() // O(1)
```

```
        }
}
```

Complejidad: O(m)