Practica 4

Nahuel Rabey

April 30, 2024

Ejercicio 1

```
TAD Racional {
        obs num: ℤ
        obs den: ℕ
        proc nuevo(in n: ℤ, in d: ℕ){
                asegura{res.num = n}
                asegura{res.den = d}
        }
        proc sumar(in x: Racional, in y: Racional): Racional {
                asegura{res.num = x.num * y.den + y.num * x.den }
                asegura{res.den = x.den * y.den}
        }
        proc restar(in x: Racional, in y: Racional): Racional {
                asegura{res.num = x.num * y.den − y.num * x.den }
                asegura{res.den = x.den * y.den}
        }
        proc multiplicar(in x: Racional, in y: Racional): Racional {
                asegura{res.num = x.num * y.num }
                asegura{res.den = x.den * y.den}
        }
        proc dividir(in x: Racional, in y: Racional): Racional {
                asegura{res.num = x.num * y.den }
                asegura{res.den = x.den * y.num}
        }
}
```

Ejercicio 3

```
TAD DobleCola< T >{
        obs data: seq< T >
        proc nueva(){
                asegura{res.data =<>}
        }
        proc encolarAdelante(inout c: DC< T >, in elem: T){
                requiere {c = C_0}
                asegura {c.data =< elem > ⟺ |c.data| = 0}
                asegura {c.data =< elem > ++ C_0.data ⟺ |c.data| > 0}
        }
        proc encolarDetras(inout c: DC < T >, in elem: T){
                requiere {c = C_0}
                asegura { c.data =< elem > ⟺ |c.data| = 0}
                asegura {c.data = C_0.data ++ < elem > ⟺ |c.data| > 0}
        }
        proc desencolar(inout c: DC < T >, in elem: T){
                requiere {c = C_0}
                asegura {c.data =<> ⟺ |c.data| = 0}
                asegura {c.data =< C_0.data[0] > ⟺ |c.data| ∈ {1, 2}}
                asegura{c.data =< C_0.data[0] > ++ subseq(C_0.data, 2, |C_0.data| − 1)}

        }
}
```

Ejercicio 4

```
TAD DobleCola< T >{
    obs data: seq< T >
    proc nueva(){
        asegura{res.data =<>}
    }
    proc encolarAdelante(inout c: DC< T >, in elem: T){
        requiere {c = C_0}
        asegura {c.data =< elem > ⟺ |c.data| = 0}
        asegura {c.data =< elem > ++ C_0.data ⟺ |c.data| > 0}
    }
    proc encolarDetras(inout c: DC < T >, in elem: T){
        requiere {c = C_0}
        asegura { c.data =< elem > ⟺ |c.data| = 0}
        asegura {c.data = C_0.data ++ < elem > ⟺ |c.data| > 0}
    }
    proc desencolar(inout c: DC < T >, in elem: T){
        requiere {c = C_0}
        asegura {c.data =<> ⟺ |c.data| = 0}
        asegura {c.data =< C_0.data[0] > ⟺ |c.data| ∈ {1, 2}}
        asegura{c.data =< C_0.data[0] > ++ subseq(C_0.data, 2, |C_0.data| − 1)}

    }
}
```

Ejercicio 5

```
TAD ColaPrioridad < T >{
    //Asumamos que el resto de la implementacion esta aca
        proc desapilarMax(inout c: CP< T >){
                requiere{c = C_0, tmp}
                requiere{c.d ≠ {}}
                asegura{tienePriMax(C_0.d, res)}
                asegura{(∀e' ∈ d)(res == e' ∧_L delKey(C_0.d, res))}
                asegura{c.d = C_0.d}
        }
        pred tienePriMax(d: dict< T >, e: T){
                {e ∈ d ∧_L (∀e' : T)(e' ∈ D → d[e] ≥ d[e'])}
        }
}
```

Ejercicio 6

```
TAD Conjunto< T >{
        obs length(): ℤ≥0
        obs pertenece(elem: T): Bool
        proc conjuntoVacio(): Conjunto< T>{
            asegura{res.length = 0}
        }
        proc pertenece(in c: Conjunto < T >, in elem: T): Bool{
            asegura(res = c.pertenece(elem))
        }
        proc agregar(inout c: Conjunto< T>, in elem: T){
            asegura(c.pertenece(elem) = True)
        }
        proc sacar(inout c: Conjunto< T>, in elem: T){
            asegura(c.pertenece(elem) = False)
        }
        proc unir(inout A: Conjunto< T>, in B: Conjunto < T>){
            asegura {(∀x ∈ T)(B.pertenece(x) = True →L A.pertenece(x) = True)}
        }
        proc restar(inout A: Conjunto< T>, in B: Conjunto < T>){
            asegura {(∀x ∈ T)(B.pertenece(x) = True →L A.pertenece(x) = False)}
        }
        proc intersecar(inout A: Conjunto< T>, in B: Conjunto < T>){
            requiere{A0 = A}
            asegura {
                (∀x ∈ T)
                (B.pertenece(x) = True ∧ A0.pertenece(x) = True →L A.pertenece(x) = True)
            }
        }
        proc agregarRapido(inout A: Conjunto< T>, in elem:  T){
            requiere{A.pertenece(elem) = False}
            asegura{A.pertenece(elem) = True}
        }
        proc tamano(in c: Conjunto< T>): ℤ{
            asegura{res = c.length()}
        }
}
```

Ejercicio 7

a

```
TAD Conjunto< T >{
        obs length(): ℤ≥0
        obs pertenece(elem: T): Bool
        proc conjuntoVacio(): Conjunto< T>{
            asegura{res.length = 0}
        }
        proc pertenece(in c: Conjunto < T >, in elem: T): Bool{
            asegura(res = c.pertenece(elem))
        }
        proc agregar(inout c: Conjunto< T>, in elem: T){
            asegura(c.pertenece(elem) = True)
        }
        proc sacar(inout c: Conjunto< T>, in elem: T){
            asegura(c.pertenece(elem) = False)
        }
        proc unir(inout A: Conjunto< T>, in B: Conjunto < T>){
            asegura {(∀x ∈ T)(B.pertenece(x) = True →L A.pertenece(x) = True)}
        }
        proc restar(inout A: Conjunto< T>, in B: Conjunto < T>){
            asegura {(∀x ∈ T)(B.pertenece(x) = True →L A.pertenece(x) = False)}
        }
```

```
        proc intersecar(inout A: Conjunto< T>, in B: Conjunto < T>){
                requiere{A_0 = A}
                asegura {
                        (∀x ∈ T)
                        (B.pertenece(x) = True ∧ A_0.pertenece(x) = True →_L A.pertenece(x) = True)
                }
        }
        proc agregarRapido(inout A: Conjunto< T>, in elem:  T){
                requiere{A.pertenece(elem) = False}
                asegura{A.pertenece(elem) = True}
        }
        proc tamano(in c: Conjunto< T>): ℤ{
                asegura{res = c.length()}
        }
}
```