Especificación TADs

Algoritmos y Estructuras de Datos

Esta sección contiene la mayoría de los TADs que vamos a ver en la materia

- Conjunto
- Diccionario
- Cola
- Pila
- Cola de prioridad
- Secuencia
- Iterador
- IteradorBidireccional

```
TAD Conjunto<T> {
    obs elems: conj<T>

    proc conjVacío(): Conjunto<T>
        asegura res.elems == {}

    proc pertenece(in c: Conjunto<T>, in T e): bool
        asegura res == true <==> e in c.elems

    proc agregar(input c: Conjunto<T>, in e: T)
        asegura c.elems == old(c).elems + {e}

    proc sacar(inout c: Conjunto<T>, in e: T)
        asegura c.elems == old(c).elems - {e}

    proc unir(inout c: Conjunto<T>, in c': Conjunto<T>)
        asegura c.elems == old(c).elems + c'.elems

    proc restar(inout c: Conjunto<T>, in c': Conjunto<T>)
        asegura c.elems == old(c).elems - c'.elems

    proc intersecar(inout c: Conjunto<T>, in c': Conjunto<T>)
        asegura c.elems == old(c).elems * c'.elems


    proc agregarRápido(input c: Conjunto<T>, in e: T)
        requiere ! e in c
        asegura c.elems == old(c).elems + {e}

    proc tamaño(in c: Conjunto<T>): int
        asegura res == |c.elems|


}
```

```
TAD Diccionario<K, V> {
    obs data: dict<K, V>

    proc diccionarioVací    o(): Diccionario<K, V>
        asegura res.data == {}

    proc está(in d: Diccionario<K, V>, in k: K): bool
        asegura res == true <==> k in d.data

    proc definir(inout d: Diccionario<K, V>, in k: K k, in v: V)
        asegura d.data == setKey(old(d).data, k, v)

    proc obtener(in d: Diccionario<K, V>, in k: K): V
        requiere k in d.data
        asegura res == d.data[k]

    proc borrar(inout d: Diccionario<K, V>, in k: K)
        requiere k in d.data
        asegura d.data == delKey(old(d).data, k)



    proc definirRápido(inout d: Diccionario<K, V>, in k: K k, in v: V)
        requiere ! k in d.data
        asegura d.data == setKey(old(d).data, k, v)

    proc tamaño(in d: Diccionario<K, V>): int
        asegura res == |d.data|


}
```

```
TAD Cola<T> {
    obs s: seq<T>

    proc colaVacía(): Cola<T>
        asegura res.s == []

    proc vacía(in c: Cola<T>): bool
        asegura res == true <==> c.s == []

    proc encolar(inout c: Cola<T>, in e: T)
        asegura c.s == old(c).s + [e]

    proc desencolar(inout c: Cola<T>): T
        requiere c.s != []
        asegura c.s == old(c).s[1..|old(c).s|]
        asegura res == old(c)[0]


    proc proximo(in c: Cola<T>): T
        requiere c.s != []
        asegura res == old(c)[0]


        }
```

```
TAD Pila<T> {
    obs s: seq<T>

    proc pilaVacía(): Pila<T>
        asegura res.s == []

    proc vacía(in p: Pila<T>): bool
        asegura res == true <==> p.s == []

    proc apilar(inout p: Pila<T>, in e: T)
        asegura p.s == old(p).s + [e]

    proc desapilar(inout p: Pila<T>): T
        requiere p.s != []
        asegura p.s == old(p).s[0..|old(p).s|-1]
        asegura res == old(p).s[|old(p).s|-1]


    proc tope(in p: Pila<T>): T
        requiere p.s != []
        asegura res == old(p).s[|old(p).s|-1]


}
```

```
TAD ColaPrioridad<T> {
    obs s: seq<T>

    proc ColaPrioridadVacía(): ColaPrioridad<T>
        asegura res.s == []

    proc vacía(in c: ColaPrioridad<T>): bool
        asegura res == true <==> c.s == []

    proc apilar(inout c: ColaPrioridad<T>, e: T)
        asegura c.s == old(c).s + [e]

    proc desapilarMax(inout c: ColaPrioridad<T>): T
        requiere c.s != []
        asegura esMax(old(c).s, res)
        asegura exists i: int :: 0 <= i < |c.s| && c.s[i] == res &&
            c.s == old(c).s[0..i] + old(c).s[i+1..|old(c).s|]

    pred esMax(s: seq<T>, res: T) {
        res in s && forall e: T :: e in s ==> e <= res
    }

}
```

```
TAD Secuencia<T> {
    obs s: seq<T>
```

```
    proc secuenciaVacía(): Secuencia<T>
        asegura res.s == []

    proc agregarAdelante(inout s: Secuencia<T>, in e: T)
        asegura s.s == [e] + old(s).s

    proc agregarAtrás(inout s: Secuencia<T>, in e: T)
        asegura s.s == old(s).s + [e]

    proc vacía(in s: Secuencia<T>): bool
        asegura res == true <==> s.s == []

    proc fin(inout s: Secuencia<T>)
        requiere |s.s| > 0
        asegura s == tail(old(s))

    proc comienzo(inout s: Secuencia<T>)
        requiere |s.s| > 0
        asegura s == head(old(s))

    proc primero(in s: Secuencia<T>): T
        requiere |s.s| > 0
        asegura res == s[0]

    proc último(in s: Secuencia<T>): T
        requiere |s.s| > 0
        asegura res == s[|s|-1]

    proc longitud(in s: Secuencia<T>): int
        asegura res == |s.s|

    proc obtener(in s: Secuencia<T>, in i: int): T
        requiere 0 <= i < |s.s|
        asegura res == s[i]

    proc eliminar(inout s: Secuencia<T>, in i: int)
        requiere 0 <= i < |s.s|
        asegura s.s == old(s).s[0..i-1] + old(s).s[i+1..|old(s).s|]

    proc copiar(in s: Secuencia<T>): Secuencia<T>
        asegura res.s == s.s

    proc modificarPosición(inout s: Secuencia<T>, in i: int, in valor: T)
        requiere 0 <= i < |s.s|
        asegura s.s == old(s).s[0..i-1] + [valor] + old(s).s[i+1..|old(s).s|]

    proc concatenar(inout s: Secuencia<T>, in s': Secuencia<T>)
        asegura s.s == old(s).s + s'.s
}
```

```
TAD Iterador<T> {
    obs s: seq<T>
```

```
    obs curr: int

    proc haySiguiente(in it: iterador<T>): bool
        asegura res == true <==> it.curr < |it.s|

    proc siguiente(inout it: iterador<T>): T
        requiere it.curr < |it.s|
        asegura res == it.s[old(it).curr]
        asegura it.s == old(it).s
        asegura it.curr == old(it).curr + 1
}
```

```
TAD IteradorBidireccional<T> {
    obs s: seq<T>
    obs curr: int

    proc haySiguiente(in it: IteradorBidireccional<T>): bool
        asegura res == true <==> it.curr < |it.s|

    proc hayAnterior(in it: IteradorBidireccional<T>): bool
        asegura res == true <==> it.curr > 0

    proc siguiente(inout it: IteradorBidireccional<T>): T
        requiere it.curr < |it.s|
        asegura res == it.s[old(it).curr]
        asegura it.s == old(it).s
        asegura it.curr == old(it).curr + 1

    proc anterior(in it: IteradorBidireccional<T>): T
        requiere it.curr >= 0
        asegura res == it.s[old(it).curr]
        asegura it.s == old(it).s
        asegura it.curr == old(it).curr - 1
}
```