



Fecha de compilación: 16 de octubre de 2024

Esta sección contiene la definición de muchos de los TADs que vamos a ver en la materia:

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

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

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

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

  proc agregar(inout c: Conjunto<T>, in e: T)
    requiere {c = C0}
    asegura {c.elems = C0.elems ∪ ⟨e⟩}

  proc sacar(inout c: Conjunto<T>, in e: T)
    requiere {c = C0}
    asegura {c.elems = C0.elems - ⟨e⟩}

  proc unir(inout c: Conjunto<T>, in c': Conjunto<T>)
    requiere {c = C0}
    asegura {c.elems = C0.elems ∪ c'.elems}

  proc restar(inout c: Conjunto<T>, in c': Conjunto<T>)
    requiere {c = C0}
    asegura {c.elems = C0.elems - c'.elems}

  proc intersecar(inout c: Conjunto<T>, in c': Conjunto<T>)
    requiere {c = C0}
    asegura {c.elems = C0.elems ∩ c'.elems}

  proc agregarRápido(inout c: Conjunto<T>, in e: T)
    requiere {c = C0 ∧ e ∉ c.elems}
    asegura {c.elems = C0.elems ∪ ⟨e⟩}

  proc tamaño(in c: Conjunto<T>): ℤ
    asegura {res = |c.elems|}
}
```

```

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

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

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

  proc definir(inout d: Diccionario<K,V>, in k: K, in v: V)
    requiere {d = D0}
    asegura {d.data = setKey(D0.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 {d = D0  $\wedge$  k  $\in$  d.data}
    asegura {d.data = delKey(D0.data, k)}

  proc definirRápido(inout d: Diccionario<K,V>, in k: K, in v: V)
    requiere {d = D0  $\wedge$  k  $\notin$  d.data}
    asegura {d.data = setKey(D0.data, k, v)}

  proc tamaño(in d: Diccionario<K,V>):  $\mathbb{Z}$ 
    asegura {res = |d.data|}
}

```

```

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

  proc colaVacía(): Cola<T>
    asegura {res.s =  $\langle \rangle$ }

  proc vacía(in c: Cola<T>): bool
    asegura {res = true  $\leftrightarrow$  c.s =  $\langle \rangle$ }

  proc encolar(inout c: Cola<T>, in e: T)
    requiere {c = C0}
    asegura {c.s = concat(C0.s,  $\langle e \rangle$ )}

  proc desencolar(inout c: Cola<T>): T
    requiere {c = C0}
    requiere {c.s  $\neq$   $\langle \rangle$ }
    asegura {c.s = subseq(C0.s, 1, |C0.s|)}
    asegura {res = C0[0]}

  proc proximo(in c: Cola<T>): T
    requiere {c = C0}
    requiere {c.s  $\neq$   $\langle \rangle$ }
    asegura {res = C0.s[0]}
}

```

```

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

  proc pilaVacía(): Pila<T>
    asegura {res.s =  $\langle \rangle$ }
}

```

```

proc vacía(in p: Pila<T>): bool
  asegura {res = true  $\leftrightarrow$  p.s =  $\langle \rangle$ }

proc apilar(inout p: Pila<T>, in e: T)
  requiere {p = P0}
  asegura {p.s = concat(P0.s,  $\langle e \rangle$ )}

proc desapilar(inout p: Pila<T>): T
  requiere {p = P0}
  requiere {p.s  $\neq \langle \rangle$ }
  asegura {p.s = subseq(P0.s, 0, |P0.s| - 1)}
  asegura {res = P0.s[|P0.s| - 1]}

proc tope(in p: Pila<T>): T
  requiere {p = P0}
  requiere {p.s  $\neq \langle \rangle$ }
  asegura {res = P0.s[|P0.s| - 1]}

```

```

TAD ColaPrioridad<T> {
  obs d: dict<T,  $\mathbb{R}$ >

  proc ColaPrioridadVacía(): ColaPrioridad<T>
    asegura {res.d = {}}

  proc vacía(in c: ColaPrioridad<T>): bool
    asegura {res = true  $\leftrightarrow$  c.d = {}}

  proc encolar(inout c: ColaPrioridad<T>, e: T, in pri:  $\mathbb{R}$ )
    requiere {c = C0}
    requiere {e  $\notin$  c.d}
    asegura {c.d = setKey(C0.d, e, pri)}

  proc desencolarMax(inout c: ColaPrioridad<T>): T
    requiere {c = C0}
    requiere {c.d  $\neq$  {}}
    asegura {c.d = delKey(C0.d, res)}
    asegura {tienePriMax(C0.d, res)}

  proc cambiarPrioridad(inout c: ColaPrioridad<T>, e: T, in pri:  $\mathbb{R}$ )
    requiere {c = C0}
    requiere {e  $\in$  c.d}
    asegura {c.d = setKey(C0.d, e, pri)}

  pred tienePriMax(d: dict<T,  $\mathbb{R}$ >, e: T)
    {e  $\in$  d  $\wedge_L$  ( $\forall e' : T$ )(e'  $\in$  d  $\rightarrow_L$  d[e]  $\geq$  d[e'])}
}

```

```

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

  proc secuenciaVacía(): Secuencia<T>
    asegura {res.s =  $\langle \rangle$ }

  proc agregarAdelante(inout s: Secuencia<T>, in e: T)
    requiere {s = S0}
    asegura {s.s = concat( $\langle e \rangle$ , S0.s)}

```

```

proc agregarAtrás(inout s: Secuencia<T>, in e: T)
  requiere {s = S0}
  asegura {s.s = concat(S0.s, ⟨e⟩)}

proc vacía(in s: Secuencia<T>): bool
  asegura {res = true ↔ s.s = ⟨⟩}

proc fin(inout s: Secuencia<T>)
  requiere {s = S0}
  requiere {|s.s| > 0}
  asegura {s = tail(S0)}

proc comienzo(inout s: Secuencia<T>)
  requiere {s = S0}
  requiere {|s.s| > 0}
  asegura {s = head(S0)}

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>): ℤ
  asegura {res = |s.s|}

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

proc eliminar(inout s: Secuencia<T>, in i: ℤ)
  requiere {s = S0}
  requiere {0 ≤ i < |s.s|}
  asegura {s.s = concat(subseq(S0.s, 0, i - 1), subseq(S0.s, i + 1, |S0.s|))}

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

proc modificarPosición(inout s: Secuencia<T>, in i: ℤ, in e: T)
  requiere {s = S0}
  requiere {0 ≤ i < |s.s|}
  asegura {s.s = concat(subseq(S0.s, 0, i - 1), ⟨e⟩, subseq(S0.s, i + 1, |S0.s|))}

proc concatenar(inout s: Secuencia<T>, in s': Secuencia<T>)
  requiere {s = S0}
  asegura {s.s = concat(S0.s, s'.s)}

```

Nos va a resultar útil definir versiones acotadas de los TADs. Con *acotado* nos referimos a que tienen una cantidad máxima de elementos, definida al inicializar la estructura. A modo de ejemplo, se muestra a continuación el TAD Conjunto Acotado.

```

TAD ConjAcotado<T> {
  obs elems: conj<T>
  obs capacidad: ℤ

  proc conjVacío(in capacidad: ℤ): ConjAcotado<T>
    requiere {capacidad > 0}
    asegura {res.elems = ⟨⟩ ∧ res.capacidad = capacidad}

  proc pertenece(in c: ConjAcotado<T>, in e: T): bool
    asegura {res = true ↔ e ∈ c.elems}

```

```

proc agregar(inout c: ConjAcotado<T>, in e: T)
  requiere {c = C0 ∧ |c.elems| < c.capacidad}
  asegura {c.elems = C0.elems ∪ {e}}
  asegura {c.capacidad = C0.capacidad}

proc sacar(inout c: Conjunto<T>, in e: T)
  requiere {c = C0}
  asegura {c.elems = C0.elems - {e}}
  asegura {c.capacidad = C0.capacidad}

proc unir(inout c: Conjunto<T>, in c': Conjunto<T>)
  requiere {c = C0 ∧ |c.elems ∪ c'.elems| < c.capacidad}
  asegura {c.elems = C0.elems ∪ c'.elems}
  asegura {c.capacidad = C0.capacidad}

proc restar(inout c: Conjunto<T>, in c': Conjunto<T>)
  requiere {c = C0}
  asegura {c.elems = C0.elems - c'.elems}
  asegura {c.capacidad = C0.capacidad}

proc intersecar(inout c: Conjunto<T>, in c': Conjunto<T>)
  requiere {c = C0}
  asegura {c.elems = C0.elems ∩ c'.elems}
  asegura {c.capacidad = C0.capacidad}

proc agregarRápido(inout c: Conjunto<T>, in e: T)
  requiere {c = C0 ∧ e ∉ c.elems ∧ |c.elems| < c.capacidad}
  asegura {c.elems = C0.elems ∪ {e}}
  asegura {c.capacidad = C0.capacidad}

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

proc capacidad(in c: Conjunto<T>): ℤ
  asegura {res = c.capacidad}
}

```

Asumiremos versiones acotadas de los otros TADs que funcionan como contenedores (Cola, Pila, Secuencia, etc.).