

# Ejemplo diseño TAD

# TAD conjunto acotado

```
TAD ConjuntoAcotado {  
  obs elems: conj<Nat>  
  obs cota: Nat
```

Cota representa el máximo  
numero que podemos  
guardar

```
proc conjVacio(in n: Nat): ConjuntoAcotado  
  asegura res.elems = {} && res.cota= n
```

```
proc agregar(inout c: ConjuntoAcotado in e: Nat)  
  requiere e<=c.cota  
  asegura c.elems = old(c).elems U {e} && c.cota=old(c).cota
```

```
proc pertenece(in c: ConjuntoAcotado, in e: Nat): Bool  
  asegura res = True  $\Leftrightarrow$  e  $\in$  c.elems  
}
```

Podemos aprovechar el hecho de que los conjuntos pueden tener valores a lo sumo  $c.cota$ ?

$(\forall n: \text{Nat}) \ n \in c.elems \rightarrow n \leq c.cota$

# Representación

```
modulo ConjBitVector implementa ConjuntoAcotado{  
    var arr: Array<Bool>  
}
```

Aprovechamos que el conjunto esta acotado para usar la posición del array para indicar la pertenencia

```
pred InvRep(c': ConjBitVector) {  
    arr!=null  
}
```

Cualquier arrar sirve, lo que determina la cota será su tamaño

```
FuncAbs(c':ConjBitVector): ConjuntoAcotado = c |  
c.cota = |arr|-1 &&  
c.elems = { n:Nat | n<= c.cota && arr[n] = true) }
```

la posición n es **true** si y solo si n está en el conjunto abstracto

```
proc conjVacio(in cota: Nat): ConjuntoAcotado
  asegura res.elems = {} && res.cota= n
```

Podríamos decir  
FuncAbs(res).elems = {}  
FuncAbs(res).cota= n

```
proc conjVacio(in n: Int): ConjBitVector
  requiere Pre:n>=0
  asegura Pos: InvRep(res) && |res.arr|=n+1 && ( $\forall k:\text{Int}$ )  $0 \leq k \leq n \Rightarrow \text{res}[k] = \text{False}$ 
{
  var cbv: ConjBitVector;
  cbv := new ConjBitVector;
  cbv.arr:= new Array<Bool>(n+1); //setea todo en False
  return cbv;
}
```

Para garantizar InvRep solo  
necesitamos  $|\text{arr}| = n+1$   
Pero para FuncAbs necesitamos  
todo en False

True  $\Rightarrow$  WpPos, código) por semántica axiomática asumida del New de Array (crea un arreglo de capacidad establecida y lo setea todo en False).

- (1) asumimos que siempre hay lugar en la memoria
- (2) esto puede ser caro en términos de tiempo: “O(bound)”. Más adelante vamos a entender formalmente esto

```
proc agregar(inout c: ConjuntoAcotado in e: Nat)
  requiere e<=c.cota
  asegura c.elems = old(c).elems U {e} && c.cota=old(c).cota
```

```
proc agregar(inout cbv: ConjBitVector, in e: Int)
  requiere Pre:InvRep(cbv) && 0<=e<=|res.arr|-1
  asegura Pos: InvRep(cbv) && cbv.arr[e] = True &&
    (∀n:int) 0<=n<|arr| && n≠e → cbv.arr[n]=old(cbv).arr[n]
{
  cbv.arr[e] := True;
  return;
}
```

Notar que:

- 1) La Pre sale de  $e \leq \text{FuncAbs}(\text{cbv}).\text{cota}$  (y  $|\text{arr}| = \text{cota} + 1$ )
- 2) La Pre podría debilitarse siempre y cuando se pueda asegurar la pos

1) Por TAD  $\text{FuncAbs}(\text{cbv}).\text{elems} = \text{FuncAbs}(\text{old}(\text{cbv}).\text{elems}) \cup \{e\}$  &&  $\text{FuncAbs}(\text{cbv}).\text{cota} = \text{FuncAbs}(\text{old}(\text{cbv})).\text{cota}$

2) La pos tiene que ser equiv o más fuerte que la pos del TAD

Nota: Pre → Wp(Pos, c código) x semántica de la asignación de array (i.e setAt).

Se podría poner una precondition más debil pero habria que chequear que e sea más chico que la cota.

Nota: Estamos asegurando no solo corrección funcional sino que InvRep se preserva

```
proc pertenece(in c: ConjuntoAcotado, in e: Nat): Bool
```

```
  asegura res = True  $\Leftrightarrow$  e  $\in$  c.elems
```

```
proc pertenece(in cbv: ConjBitVector, in e: Int): Bool
```

```
  requiere Pre:InvRep(cbv)
```

```
  asegura Pos: res = True  $\Leftrightarrow$  0 $\leq$ e $\leq$ |arr|-1 && arr[e] = True {
```

```
    if e $\leq$  |cbv.arr|-1
```

```
      res := cbv.arr[e];
```

```
    else
```

```
      res:= False
```

```
    return res;
```

```
  }
```

La pos es equiv o más fuerte que  $\text{res} = \text{True} \Leftrightarrow e \in \text{FuncAbs}(\text{cbv}).\text{elems}$



# TAD Conjunto

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

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

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

```
proc pertenece(in c: Conjunto<T>, in e: T): Bool  
    asegura res = True  $\Leftrightarrow$   $e \in c.elems$   
}
```

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



# Implementación de conjuntos

Muchas posibles implementaciones

- Arrays (redimensionando)
- Secuencias
- Listas encadenadas
- Árboles (binarios, AVL, etc)
- Tablas de Hash, etc

# TAD Secuencia

Asumo que existe una implementación que respeta esta especificación

```
TAD Secuencia<T> {  
  obs elems: Seq<T>  
  
  proc secVacía(): Secuencia<T>  
    asegura res.elems = []  
  
  proc agregar(inout s: Secuencia<T>, in pos: Int, in e: T)  
    requiere  $0 \leq \text{pos} \leq |s.\text{elems}|$   
    asegura  $s.\text{elems} = \text{subsec}(\text{old}(s).\text{elems}, 0, \text{pos}) ++ [e] ++$   
       $\text{subsec}(\text{old}(s).\text{elems}, \text{pos}, |\text{old}(s).\text{elems}|)$   
  
  proc at(in s: Secuencia<T>, in pos: Int) T  
    requiere  $0 \leq \text{pos} < |\text{elems}|$   
    asegura res = s.elems[pos]  
  
  proc esta(in c: Secuencia<T>, in e: T): Bool  
    asegura res =  $\text{True} \Leftrightarrow e \in s.\text{elems}$   
  }  
  proc long(in c: Secuencia<T>): Nat  
    asegura res = |s.elems|  
  }  
}
```

Si  $\text{pos} = |s.\text{elems}|$   
se va a agregar un  
elemento al final

# Implementación sobre Secuencia

```
modulo ConjSecuencia<T> implementa Conjunto<T>
{
  var sec: Secuencia<T>
}
```

```
pred InvRep(cs: ConjSecuencia<T>) {
  sinRepetidos(cs.sec.elems)
}
```

Para simplificar la implementación

```
aux FuncAbs(cs: ConjSecuencia<T>): Conjunto<T> = c |
  mismos(c.elems, cs.sec.elems)
```

Hay una correspondencia uno a uno entre el conjunto y la secuencia que lo representa

```
pred sinRepetidos(s: Seq<T>) {  $(\forall i: \text{int}) 0 \leq i < |s| \rightarrow \text{apariciones}(s, s[i]) = 1$  }
```

```
pred mismos(c: Conj<T>, s: Seq<T>) {  $(\forall e: T) \text{apariciones}(e, c) = \text{apariciones}(e, s)$  }
```

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

```
proc conjVacio(): ConjSecuencia<T>
  asegura res.sec.elems = []
{
  res.sec = Secuencia.secVacía();
  return res;
}
```

```
proc agregar(inout cs: Conjunto<T>, in e: T)
  asegura cs.elems = old(cs).elems U {e}
```

```
proc agregar(inout cs: ConjSecuencia<T>, in e: T)
  asegura FuncAbs(cs).elems = FuncAbs(old(cs)).elems U {e}

{
  if !cs.sec.esta(e)
    cs.sec.agregar(cs.sec.long(), e)
  else
    skip;
}
```

Podríamos decir que  $e$  está en  $cs.sec.elems$  para todo  $x \neq e$  pasa que  $cs.sec.elems$  está en  $old(cs).sec.elems$  pero así es más fácil porque usamos conjuntos en vez de secuencias

```
proc pertenece(in c: Conjunto<T>, in e: T): Bool
  asegura res = True  $\Leftrightarrow$   $e \in c.\text{elems}$ 
}
```

$e \in \text{FuncAbs}(cs).\text{elems}$

```
proc pertenece(in cs: ConjSecuencia<T>, in e: T): Bool
  asegura res = True  $\Leftrightarrow$   $e \in cs.\text{sec}.\text{elems}$ 
  var i:int
  i:= 0;
  while i<cs.sec.longitud() && cs.sec.at(i)≠e {
    i:=i+1;
  }
  res = i<cs.sec.elems.longitud();
  return res;
}
```

Invariante?

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

Podríamos decir que para todo  $x \neq e$  pasa que  $cs.sec.elems$  esta en  $old(cs).sec.elems$ , pero así es más fácil porque usamos conjuntos en vez de secuencias

```
proc sacar(inout cs: ConjSecuencia <T>, in e: T)
  asegura FuncAbs(cs).elems = FuncAbs(old(cs)).elems - {e}
{
  // Tenemos que generar una nueva secuencia sin e
  var i:int; var s:Secuencia<T>;
  i:= 0; s:= secVacía(); i

  while i<cs.sec.longitud() {
    if cs.sec.at(i)≠e then
      s.agregar(s.long(), cs.sec.elems[i])
      i:=i+1;
    }
  cs.sec = s; // reemplazo la secuencia
  return;
}
```

Inv: apariciones(e,s) = 0 &&  
 $0 \leq i \leq |cs.sec.elems|$  &&  
 $(\forall x: T) x \neq e \rightarrow apariciones(x,s) =$   
 $apariciones(x, subseq(cs.sec.elems, 0, i))$