

Ejercicio 1

```
proc agregarAtras (inout l: listaEnlazada < T >, in t: T): {  
    nodo := new NodoLista < T >  
    nodo.valor := t  
    nodo.siguiente := null  
    if (l.longitud==0) then  
        l.primeros := nodo  
        l.ultimo := nodo  
    else  
        l.ultimo.siguiente := nodo  
        l.ultimo := nodo  
    endif  
    l.longitud++  
}  
  
proc obtener (in l: listaEnlazada < T >, in i:  $\mathbb{Z}$ ): T {  
    j := 0  
    while (j < i) do  
        j++  
    endwhile  
    return l[j]  
}  
  
proc concatenar (inout l1: listaEnlazada < T >, in l2: listaEnlazada < T >): {  
    if (l1.longitud! = 0 and l2 == 0) then  
        l1 := l2  
    ElseIf (l1.longitud == 0 and l2.longitud! = 0) then  
        l1.primeros := l2.primeros  
        l1.ultimo := l2.ultimo  
        puntero := l2.primeros.siguiente  
        while (puntero!=null) do  
            agregarAtras(l1,puntero)  
            puntero := puntero.siguiente  
        endwhile  
    else  
        l1.ultimo := l2.ultimo  
        puntero := l2.primeros  
        while (puntero!=null) do  
            agregarAtras(l1,puntero)  
            puntero := puntero.siguiente  
        endwhile  
    endif  
    l1.longitud := l1.longitud+l2.longitud  
}
```

agregarAtras $\in O(1)$

obtener $\in O(n)$

concatenar $\in O(n)$

El invariante de representacion debe indicar que el largo de la secuencia se corresponde con la cantidad de elementos que tiene la misma y que 'primero' y 'ultimo' pertenecen a la secuencia (esta mal)

Ejercicio 2

pred invRep (*c*:conjArr $< T >$) {*c.tamano* \geq *c.datos.length* \wedge noRepetidos(*c*)}

pred noRepetidos (*c*:conjArr $< T >$) { $(\forall i : \mathbb{Z})(0 \leq i < c.tamano \rightarrow \neg(\exists j : \mathbb{Z})(0 \leq j < c.tamano \wedge j \neq i \wedge c[j] = c[i]))$ }

pred abs (*c*:conjArr $< T >$, *c'*: ConjuntoAcotado $< T >$) {*c.tamano* = *c'.capacidad* $\wedge (\forall i : \mathbb{Z})(0 \leq i \leq c.tamano \rightarrow c[i] \in c'.elems)$ }

Ejercicio 3

pred invRep (*c*: conjuntoLista $< T >$) {*c.tamano* = *c.datos.longitud*}

pred abs (*c*: conjuntoLista $< T >$, *c'*: conjunto $< T >$) {*c.tamano* = $|c'.elems| \wedge (\forall e : T)(e \in c.datos \leftrightarrow e \in c'.elems)$ }

modulo conjuntoLista $< T >$ **implementa** conjunto $< T >$ {

```
  proc conjVacio (): ConjuntoLista  $< T >$  {  
    res.datos := new listavacia()  
    res.tamaño := 0  
    return res  
  }
```

```
  proc pertenece (in c:conjuntoLista  $< T >$ , in e:T): Bool {  
    res := False  
    puntero := c.datos.cabeza  
    while (puntero! = null  $\wedge$  res = False) do  
      if (puntero.val==e) then  
        res := True  
      else  
        puntero := puntero.siguiete  
      endif  
    endwhile  
    return res  
  }
```

```

proc agregar (inout c:conjuntoLista < T >, in e:T): {
    if (pertenece(c,e)) then
        newUlt := new nodoLista < T >
        newUlt.sig := null
        newUlt.ant := c.datos.ultimo
        c.datos.ultimo.siguiente := newUlt
        c.datos.ultimo := newUlt
    endif
}

```

(agregarRapido asume que el elemento se encuentra en la lista)

```

proc agregarRapido (inout c:conjuntoLista < T >, in e:T): {
    newUlt.sig := null
    newUlt := new nodoLista < T >
    newUlt.ant := c.datos.ultimo
    c.datos.ultimo.siguiente := newUlt
    c.datos.ultimo := newUlt
}

```

```

proc unir (inout c1:conjuntoLista < T >, in c2:conjuntoLista < T >): {
    if (c2.tamaño!=0 and c1.tamaño!=0) then
        puntero := c2.datos.cabeza
        c1.ultimo.siguiente := puntero
        while (puntero!=null) do
            if (pertenece(c1, puntero) == False) then
                agregar(c1, puntero)
                if (puntero.siguiente == null) then
                    c1.datos.ultimo := puntero
                endif
            endif
            puntero := puntero.siguiente
        endwhile
        ElseIf (c1.tamaño=0) then
            c1 := c2
        endif
    }
}

```

- $conjVacio \in O(1)$
- $pertenece \in O(n)$
- $agregar \in O(n)$
- $agregarRapido \in O(1)$
- $unir \in O(n^2)$

Ejercicio 4

```

modulo indice implementa Conjunto⟨tupla⟨ℤ, ℤ, ℤ⟩⟩ {
    var data : array⟨tupla⟨ℤ, ℤ, ℤ⟩⟩
    var indices : array⟨array⟨ℤ⟩⟩
}

pred invRep (i:indice){(∀a : array⟨ℤ⟩)(a ∈ i.indices → a.length = i.data.longitud) ∧
(∀j : ℤ)(0 ≤ j ≤ 2 →L (∀k : ℤ)(1 ≤ k < i[j].length →L i.data[k]j ≥ i.data[k - 1]j)) ∧ i.indices.length = 3}

pred abs (i: indice,c': Conjunto⟨tupla⟨ℤ, ℤ, ℤ⟩⟩){(∀e : tupla⟨ℤ, ℤ, ℤ⟩)(e ∈ i.data ↔ e ∈ c'.elems)}

proc buscarPor (i : indice, comp : ℤ, t : tupla⟨ℤ, ℤ, ℤ⟩): Bool {
    res := false
    int j := 0
    pos := i.indices[comp][j]
    while (j < i.data.length) do
        if (t == i.data[pos]) then
            res := true
        endif
    endwhile
    return res
}

proc agregar (inout i : indice, in t : tupla⟨ℤ, ℤ, ℤ⟩, in comp : ℤ): {
    if (buscarPor(i,t,comp)) then
        i.data.length++
        i.data[i.data.length-1] := t
        int j := 0
        while (j < 3) do
            agregarAIndice(i,j,t)
        endwhile
    }

proc agregarAIndice (inout i : indice, in j : ℤ, in t : tupla⟨ℤ, ℤ, ℤ⟩, in comp : ℤ): {
    j := 0
    newIndex := new array⟨ℤ⟩(i.data.length)
    noSeAgrego := True
    while (j < i.data.length - 1) do
        pos := i.indices[comp][j]
        if (i.data[pos]comp ≥ tcomp and noSeAgrego) then
            newArray[j] := i.data.length-1
            noSeAgrego := False
            newArray[j+1] := pos
        else
            newArray[j] := pos
        endif
    endwhile
}

```

```

    if (noSeAgrego) then
        newArray[i.data.length-1] := i.data.length-1
    endif
    i.indices[comp] := newIndex
}

proc sacar (inout i : indice, in j :  $\mathbb{Z}$ , in t : tupla( $\mathbb{Z}$ ,  $\mathbb{Z}$ ,  $\mathbb{Z}$ ), in comp :  $\mathbb{Z}$ ): {
    if (buscarPor(i,t,comp)) then
        newData := newarray<tupla( $\mathbb{Z}$ ,  $\mathbb{Z}$ ,  $\mathbb{Z}$ )>(i.data.length - 1)
        j := 0
        while (j < i.data.length) do
            if (i.data[j] != t) then
                newArray[j] := i.data[j]
            else
                posTupla := j
            endif
        endwhile
        k := 0
        newIndex0 := new array< $\mathbb{Z}$ >(i.data.length - 1)
        newIndex1 := new array< $\mathbb{Z}$ >(i.data.length - 1)
        newIndex2 := new array< $\mathbb{Z}$ >(i.data.length - 1)
        while (k < i.data.length) do
            if (i.indices[k] != posTupla) then
                newIndex0[k] := i.indices[0][k]
                newIndex1[k] := i.indices[1][k]
                newIndex2[k] := i.indices[2][k]
            endif
        endwhile
        i.indices[0] := newIndex0
        i.indices[1] := newIndex1
        i.indices[2] := newIndex2
        i.data := newData
    }
}

```