1. Completá la implementación de listas dada en el teórico usando punteros.

Especificación de Listas

```
spec List of T where

constructors
        fun empty() ret l : List of T
        {- crea una lista vacía. -}

        proc addl (in e : T, in/out l : List of T)
        {- agrega el elemento e al comienzo de la lista l. -}

destroy
      proc destroy (in/out l : List of T)
      {- Libera memoria en caso que sea necesario. -}
```

```
operations
      fun is_empty(l : List of T) ret b : bool
      {- Devuelve True si l es vacía. -}

      fun head(l : List of T) ret e : T
      {- Devuelve el primer elemento de la lista l -}
      {- PRE: not is_empty(l) -}

      proc tail(in/out l : List of T)
      {- Elimina el primer elemento de la lista l -}
      {- PRE: not is_empty(l) -}

      proc addr (in/out l : List of T,in e : T)
      {- agrega el elemento e al final de la lista l. -}

      fun length(l : List of T) ret n : nat
      {- Devuelve la cantidad de elementos de la lista l -}

      proc concat(in/out l : List of T,in l0 : List of T)
      {- Agrega al final de l todos los elementos de l0
        en el mismo orden.-}

      fun index(l : List of T,n : nat) ret e : T
      {- Devuelve el n-ésimo elemento de la lista l -}
      {- PRE: length(l) > n -}

      proc take(in/out l : List of T,in n : nat)
      {- Deja en l sólo los primeros n
        elementos, eliminando el resto -}

      proc drop(in/out l : List of T,in n : nat)
      {- Elimina los primeros n elementos de l -}

      fun copy_list(l1 : List of T) ret l2 : List of T
      {- Copia todos los elementos de l1 en la nueva lista l2 -}
```

Implementación de Listas mediante punteros

```
implement List of T where

type Node of T = tuple
                    elem : T
                    next : pointer to (Node of T)
                 end tuple

type List of T = pointer to (Node of T)

fun empty() ret l : List of T
    l := null
end fun
```

1) Ej1 - Constructors

1) Imp. addl

```
proc addl (in e : T, in/out l : List of T)
  var p : pointer to (Node of T)
  alloc(p)
  p->elem := e
  p->next := l
  l := p
end proc
```

1) Imp. empty

```
fun empty() ret l : List of T
        alloc(l)
        l := null
end fun
{- crea una lista vacía. -}
```

1) Imp. destroy

```
proc destroy (in/out l : List of T)
{- Libera memoria en caso que sea necesario. -}
    var p : pointer to (Node of T)
    if ( l := Null) →
        free(l)
    else →
        while ( l =! Null) do
            p := l→next
            free(l)
            l := p
         od
end proc
```

1) Ej1 - Operations

1) Imp. is_empty

```
fun is_empty(l : List of T) ret b : bool
   b := l = null
end fun

{- PRE:  not is_empty(l) -}
fun head(l : List of T) ret e : T
   e := l-> elem
end fun
```

1) Imp. tail

```
{- PRE:  not is_empty(l) -}
proc tail(in/out l : List of T)
   var p : pointer to (Node of T)
   p := l
   l := l->next
   free(p)
end proc
```

1) Imp. addr

```
proc addr (in/out l : List of T,in e : T)
   var p,q : pointer to (Node of T)
   alloc(q)
   q->elem := e
   q->next := null
   if (not is_empty(l))
      then p := l
           do (p->next != null)
                p := p->next
           od
           p->next := q
      else l := q
   fi
end proc
```

1) Imp. length

```
fun length(l : List of T) ret n : nat
  var p : pointer to (Node of T)
  n := 0
  p := l
  do (p != null)
      n := n+1
      p := p->next
  od
end fun
```

Ejercicio: Implementar el resto de las operaciones.

Preguntas importantes: ¿cuándo necesito hacer **alloc** a un puntero? ¿cuándo necesito hacer **free**?

1) Imp. head

```
{- PRE: not is_empty(l) -}
fun head (l : List of T) ret e : T
      e := l->elem
end fun
```

1) Imp. concat

```
proc concat(in/out l : List of T,in l0 : List of T)
{- Agrega al final de l todos los elementos de l0 en el mismo orden.-}
        var p : pointer to (Node of T)
        if ( l = null and l0 != null) then
                l := l0
        else →
                p := l
                while ( p → next != null ) do
                        p := p → next
                od
                p → next := l0
        fi
end proc
{- el último nodo de l, se establece como el inicio de la lista l0, efectivamente concatenado l0 al final de l -}
```

1) Imp. drop

{- Elimina los primeros n elementos de l -}

```
proc drop(in/out l : List of T,in n : nat)
      var p : pointer to (Node of T)
      var i : nat
      i := 0
      if (l =! null and 0 < n) then
            while( i < n and ( l != Null ) ) do
                  p := l
                  l := l→next
                  free(p)
                  i := i + 1
            od
      fi
end proc
```

1) Imp. index

{- PRE: length(l) > n -}

```
fun index(l : List of T,n : nat) ret e : T
```

{- Devuelve el n-ésimo elemento de la lista l -}

```
   var i : nat
     var p : pointer to (Node of T)
     p := l
    for i := 1 to n do
         p := p→next
    od
    e := p→elem
end fun
```

1) Imp. take

```
proc take(in/out l : List of T,in n : nat)
{- Deja en l sólo los primeros n elementos, eliminando el resto -}
      var p,q : pointer to (Node of T)
      var i : nat
      i := 0
      if ( l =! null )
            if( n = 0 ) then
                  while ( l =! null) do
                        p := l
                        l := l → next
                        free(p)
                  od
            else ( 0 < n ) →
                  p := l
                  while ( i < n and l =! null) do
                        p := p→next
                        i := i + 1
                  od
                  While ( p != Null ) do
                        q := p
                        p := p→next
                        free(q)
                  od
            fi
      fi
```

1) Imp. copy

```
fun copy_list(l1 : List of T) ret l2 : List of T
{- Copia todos los elementos de l1 en la nueva lista l2 -}
    var p: pointer to (Node of T)
    var n: nat
    n:= length(l1)
    if (l1 = null) then l2:= empty()
    else
        p:= l1
        alloc(l2)
        for i:= 1 to n do
            l2->elem = p->elem
            l2->next = p->next
            p:= p->next
        od
    fi
end fun
{- Copia todos los elementos de l1 en la nueva lista l2 -}
```

2) Ej2 - Imp TAD con arreglos

2. Dada una constante natural N, implementá el TAD Lista de elementos de tipo T, usando un arreglo de tamaño N y un natural que indica cuántos elementos del arreglo son ocupados por elementos de la lista. ¿Esta implementación impone nuevas restricciones? ¿En qué función o procedimiento tenemos una nueva precondición?

```
implement List of T where

type List of T = tuple
                   elems : array[1..N] of T
                   size : nat
                 end tuple
```

Ej2 - Constructors

Imp. addl

```
{- PRE: l.size < N -}
proc addl (in e : T, in/out l : List of T)

for i := n down to 1 do
      l.elem[i+1] := l.elem[i] {- obs: la 1er asignación no se pasa puesto que l.size < N -}
od
l.elems[1] := e
l.size := l.size + 1

end proc
```

Imp. empty

```
fun empty() ret l : List of T
      l.size := 0
end fun
```

Imp. destroy

```
proc destroy (in/out l : List of T)
    skip
end proc
```

{- esta implementación de listas no accede a memoria dinámica -}

Ej2 - Operations

Imp. is_empty

```
fun is_empty(l : List of T) ret b : bool
    b :=  l.size == 0
end fun
```

Imp. tail

{- PRE: not is_empty(l) -}

```
proc tail(in/out l : List of T)
    for i := 1 to n do
        l.elem[i] := l.elem[i+1]
    od
    l.size := l.size - 1
end proc
```

Imp. addr

```
proc addr (in/out l : List of T,in e : T)
```

{- agrega el elemento e al final de la lista l. -}

```
   if ( l.size == 0 ) →
        l.elem[1] := e
    else →
      l.size := l.size + 1
       l.elem[size+1] := e
    fi
end proc
```

Imp. length

```
fun length(l : List of T) ret n : nat
```

{- Devuelve la cantidad de elementos de la lista l -}

```
n := l.size
end fun
```

Imp. head

```
{- PRE: not is_empty(l) -}
fun head (l : List of T) ret e : T
    e := l.elem[1]
end fun
```

Imp. concat

{- Agrega al final de l todos los elementos de l0 en el mismo orden.-}
{- PRE: l.size + l.size ≤ N -}

```
proc concat(in/out l : List of T,in l0 : List of T)
    for i := 1 to l0.size do
        l.elem[size+1] := l0.elem[i]
    od
    l.size := l.size + l0.size
end proc
```

Imp. drop

{- PRE: n ≤ l.size-}
{- Elimina los primeros n elementos de l -}

```
proc drop(in/out l : List of T,in n : nat)

    for i := n + 1 to l.size do
        l.elem[i - n] := l.elem[i]
    od
    l.size := l.size - n
end proc
```

Imp. index

{- Devuelve el n-ésimo elemento de la lista l -}
{- PRE: n ≤ l.size -}

```
fun index(l : List of T,n : nat) ret e : T
    e := l.elem[n]
end fun
```

Imp. take

{- Deja en l sólo los primeros n elementos, eliminando el resto -}
{- PRE n ≤ l.size -}

```
proc take(in/out l : List of T,in n : nat)
    l.size = n
end proc
```

Imp. copy

{- Copia todos los elementos de l1 en la nueva lista l2 -}

```
fun copy_list(l1 : List of T) ret l2 : List of T
    l2.size := l1.size
    for i := 1 to l1.size do
        l2.elem[i] := l1.elem[i]
    od
end fun
```

---

3. Implementá el procedimiento *add_at* que toma una lista de tipo T, un natural n, un elemento e de tipo T, y agrega el elemento e en la posición n, quedando todos los elementos siguientes a continuación. Esta operación tiene como precondición que n sea menor al largo de la lista.

   AYUDA: Puede ayudarte usar las operaciones copy, take y drop.

3) Ej3 - add_at

```
{- PRE: n < length(l) -}
proc add_at (in/out list of T, in n : nat, in e : T)
    var l2 : list of T
    l2 := copy(l1)
    take(l1, n - 1)
    drop(l2, n - 1)
    addr(l1,e)
    concat(l1,l2)
end proc
```

---

4. (a) Especificá un TAD *tablero* para mantener el tanteador en contiendas deportivas entre dos equipos (equipo A y equipo B). Deberá tener un constructor para el comienzo del partido (tanteador inicial), un constructor para registrar un nuevo tanto del equipo A y uno para registrar un nuevo tanto del equipo B. El tablero sólo registra el estado actual del tanteador, por lo tanto el orden en que se fueron anotando los tantos es irrelevante.

   Además se requiere operaciones para comprobar si el tanteador está en cero, si el equipo A ha anotado algún tanto, si el equipo B ha anotado algún tanto, una que devuelva verdadero si y sólo si el equipo A va ganando, otra que devuelva verdadero si y sólo si el equipo B va ganando, y una que devuelva verdadero si y sólo si se registra un empate.

   Finalmente habrá una operación que permita anotarle un número n de tantos a un equipo y otra que permita "castigarlo" restándole un número n de tantos. En este último caso, si se le restan más tantos de los acumulados equivaldrá a no haber anotado ninguno desde el comienzo del partido.

   (b) Implementá el TAD Tablero utilizando una tupla con dos contadores: uno que indique los tantos del equipo A, y otro que indique los tantos del equipo B.

   (c) Implementá el TAD Tablero utilizando una tupla con dos naturales: uno que indique los tantos del equipo A, y otro que indique los tantos del equipo B. ¿Hay alguna diferencia con la implementación del inciso anterior? ¿Alguna operación puede resolverse más eficientemente?

Ej4 TAD tablero

Especificación:

```
spec Tablero where

constructor
      {- aca definiremos funciones y procedimientos que
            devuelven/modifican tableros. -}

      fun tableroInicial() ret t : Tablero
      {- devuelve un tablero representando el estado en el que no hay
            tantos. -}

      proc nuevoTantoA (in/out t: Tablero)
      {- agrega el tablero t un punto para el equipo A. -}

      proc nuevoTantoB (in/out t: Tablero)
      {- agrega el tablero t un punto para el equipo B. -}

      destroy
            proc destroyTablero(in/out t : Tablero)
            {- Libera memoria en caso de ser necesario. -}
```

```
Operations
      fun tableroEnCero(t : Tablero) ret b : Bool
      {- Devuelve true si el tablero t representa cuando A y B no tienen
            puntos. -}

      fun anotaA (t : Tablero) ret b : Bool
      {- Devuelve true si A anotó algún punto. -}

      fun anotaB (t : Tablero) ret b : Bool
      {- Devuelve true si B anotó algún punto. -}

      fun ganaA (t : Tablero ) ret b : Bool
      {- Devuelve true si y sólo si A va ganando. -}

      fun ganaB (t : Tablero ) ret b : Bool
      {- Devuelve true si y sólo si B va ganando. -}

      fun empatan (t : Tablero) ret b : Bool
      {- Devuelve true si y sólo si A y B van empatando. -}

      proc nuevosTantosA (in/out t : Tablero, in n : nat)
      {- Agrega al tablero t n tantos para el equipo A. -}

      proc nuevosTantosB (in/out t : Tablero, in n : nat)
      {- Agrega al tablero t n tantos para el equipo B. -}

      proc restarTantosA (in/out t : Tablero, in n : nat)
      {- Agrega al tablero t n tantos para el equipo A. Si A no tiene
      tantos queda en 0. -}

      proc restarTantosB (in/out t : Tablero, in n : nat)
      {- Agrega al tablero t n tantos para el equipo B. Si B no tiene
      tantos queda en 0. -}
end spec
```

4)b)

```
spec Counter where

constructors
      fun init() ret c : Counter
      {- crea un contador inicial. -}

      proc incr (in/out c : Counter)
      {- incrementa el contador c. -}

destroy
    proc destroy (in/out c : Counter)
    {- Libera memoria en caso que sea necesario. -}
```

```
operations
    fun is_init(c : Counter) ret b : Bool
    {- Devuelve True si el contador es inicial -}

    proc decr (in/out c : Counter)
    {- Decrementa el contador c. -}
    {- PRE:  not is_init(c) -}
```

Implementación del tipo Tablero (utilizando dos contadores)

```
implement Tablero where

type Tablero = tuple
                  tantosA : Counter
                  tantosB : Counter
               end tuple
```

```
fun tableroInicial() ret t : Tablero
    var t : Tablero
    t.tantosA = init()
    t.tantosB = init()
end fun
```

```
proc nuevoTantoA (in/out t: Tablero)
    incr(t.tantosA)
end proc
```

```
proc nuevoTantoB (in/out t: Tablero)
    incr(t.tantosB)
end proc
```

```
proc destroyTablero(in/out t : Tablero)
      destroy(t.tantosA)
      destroy(t.tantosB)
end proc
```

```
fun tableroEnCero(t : Tablero) ret b : Bool
```

```
            b := false
            if (is_init(t.tantosA) and is_init(t.tantosB)) →
                  b := true
            fi
      end fun
```

```
      fun anotaA (t : Tablero) ret b : Bool
            b := (!is_init(t.tantosA))
      end fun
```

```
      fun anotaB (t : Tablero) ret b : Bool
            b := (!is_init(t.tantosB))
      end fun
```

```
      {- Asumimos la existencia de una función copy para Counters. -}
      fun ganaA (t : Tablero ) ret b : Bool
            b := true
            var c1,c2 : Counter;
            c1 := init()
            c2 := init()
            copyCounter(t.tantosA, c1)
            copyCounter(t.tantosB, c2)
            while(!is_init(c1) && !is_init(c2)) do
                  decr(c1)
                  decr(c2)
            od
            if(is_init(c1)) →
                  b := false
            fi
      end fun
```

```
      {- Asumimos la existencia de una función copy para Counters. -}
      fun ganaB (t : Tablero ) ret b : Bool
            b := true
            var c1,c2 : Counter;
            c1 := init()
            c2 := init()
            copyCounter(t.tantosA, c1)
            copyCounter(t.tantosB, c2)
            while(!is_init(c1) && !is_init(c2)) do
```

```
            decr(c1)
            decr(c2)
      od
      if(is_init(c2)) →
            b := false
      fi
end fun
```

```
{- Asumimos la existencia de una función copy para Counters. -}
fun empatan (t : Tablero) ret b : Bool
      b := false
      var c1,c2 : Counter;
      c1 := init()
      c2 := init()
      copyCounter(t.tantosA, c1)
      copyCounter(t.tantosB, c2)
      while(!is_init(c1) && !is_init(c2)) do
            decr(c1)
            decr(c2)
      od
      if(is_init(c2) && is_init(c1)) →
            b := true
      fi
end fun
```

```
proc nuevosTantosA (in/out t : Tablero, in n : nat)
      for i := 1 to n do
            incr(t.tantosA)
      od
end proc
```

```
proc nuevosTantosB (in/out t : Tablero, in n : nat)
      for i := 1 to n do
            incr(t.tantosB)
      od
end proc
```

```
proc restarTantosA (in/out t : Tablero, in n : nat)
```

```
            var i : nat
            var b : bool
            b := true
            i := 1
            if(is_init(t.tantosA)) →
                  skip
            else →
                  while(i ≤ n && b) do
                        if(is_init(t.tantosA)) →
                              b := false
                        else →
                              decr(t.tantosA)
                        fi
                  od
            fi
      end proc
```

```
      proc restarTantosB (in/out t : Tablero, in n : nat)
            var i : nat
            var b : bool
            b := true
            i := 1
            if(is_init(t.tantosB)) →
                  skip
            else →
                  while(i ≤ n && b) do
                        if(is_init(t.tantosB)) →
                              b := false
                        else →
                              decr(t.tantosB)
                        fi
                  od
            fi
      end proc
```

```
      end implement
```

4)b)

Implementación del tipo Tablero (utilizando dos contadores)

```
implement Tablero where

type Tablero = tuple
                  tantosA : nat
                  tantosB : nat
               end tuple
```

```
fun tableroInicial() ret t : Tablero
    var t : Tablero
    t.tantosA = 0
    t.tantosB = 0
end fun
```

```
proc nuevoTantoA (in/out t: Tablero)
    t.tantosA = t.tantosA + 1
end proc
```

```
proc nuevoTantoB (in/out t: Tablero)
    t.tantosB = t.tantosB + 1
end proc
```

```
proc destroyTablero(in/out t : Tablero)
      skip
end proc
```

```
fun tableroEnCero(t : Tablero) ret b : Bool
      b := false
      if (t.tantosA == 0 and t.tantosB == 0) →
            b := true
      fi
end fun
```

```
fun anotaA (t : Tablero) ret b : Bool
      b := (t.tantosA != 0)
end fun
```

```
fun anotaB (t : Tablero) ret b : Bool
      b := (t.tantosB != 0)
end fun
```

```
fun ganaA (t : Tablero ) ret b : Bool
      b := false
      if(t.tantosA > t.tantosB) →
            b := true
      fi
end fun
```

```
fun ganaB (t : Tablero ) ret b : Bool
      b := false
      if(t.tantosB > t.tantosA) →
            b := true
      fi

end fun
```

```
{- Asumimos la existencia de una función copy para Counters. -}
fun empatan (t : Tablero) ret b : Bool
      b := false
      if(t.tantosA == t.tantosB) →
            b := true
      fi
end fun
```

```
proc nuevosTantosA (in/out t : Tablero, in n : nat)
      for i := 1 to n do
            t.tantosA = t.tantosA + 1
      od
end proc
```

```
proc nuevosTantosB (in/out t : Tablero, in n : nat)
      for i := 1 to n do
            t.tantosB = t.tantosB + 1
      od
end proc
```

```
proc restarTantosA (in/out t : Tablero, in n : nat)
      var i : nat
      i := 1
      while( 1 ≤ n and t.tantosA != 0 ) do
            t.tantosA = t.tantosA - 1
      od
end proc
```

```
proc restarTantosB (in/out t : Tablero, in n : nat)
      var i : nat
      i := 1
      while( 1 ≤ n and t.tantosB != 0 ) do
            t.tantosB = t.tantosB - 1
      od

end proc
```

```
end implement
```

5. Especificá el TAD Conjunto finito de elementos de tipo T. Como constructores considerá el conjunto vacío y el que agrega un elemento a un conjunto. Como operaciones: una que chequee si un elemento e *pertenece* a un conjunto c, una que chequee si un conjunto *es vacío*, la operación de *unir* un conjunto a otro , *insersectar* un conjunto con otro y obtener la *diferencia*. Estas últimas tres operaciones deberían especificarse como procedimientos que toman dos conjuntos y modifican el primero de ellos.

Ej5)

spec Conjunto **where**

constructors

```
fun vacío ( ) ret c : Conjunto
```
{- Devuelve el conjunto vacío -}

```
proc agregar ( in/out c : Conjunto of T, in e : T )
```
{- añade un elemento al conjunto c -}

destroy

```
proc destruirConjunto ( in/out c : Conjunto of T )
```
{- Procedimiento que libera memoria en caso de ser necesario -}

operations

```
fun pertenece ( c : Conjunto of T, e : T ) ret b : bool
```
{- Función que devuelve true si un elemento e está en el conjunto s -}

```
fun esVacio ( c : Conjunto of T) ret b : bool
```
{- Función que devuelve true si el conjunto s es vacío -}

```
proc unir (in/out c : Conjunto of T, in c2 : Conjunto of T )
```
{- Procedimiento que une el conjunto c al conjunto c2 -}

```
proc intersección (in/out c : Conjunto of T, in c2 : Conjunto of T)
```
{- Procedimiento que modifica s dejando solo los elementos comunes con c-}

```
proc diferencia (in/out c : Conjunto of T, in c2 : Conjunto of T)
```
{- Proc que modifica c quitando aquellos elementos comunes que tenga con c2 -}

end spec

6. Implementá el TAD Conjunto finito de elementos de tipo T utilizando:

   (a) una lista de elementos de tipo T, donde el constructor para agregar elementos al conjunto se implementa directamente con el constructor **addl** de las listas.

   (b) una lista de elementos de tipo T, donde se asegure siempre que la lista está ordenada crecientemente y no tiene elementos repetidos. Debes tener cuidado especialmente con el constructor de agregar elemento y las operaciones de unión, intersección y diferencia. A la propiedad de mantener siempre la lista ordenada y sin repeticiones le llamamos *invariante de representación*.
   Ayuda: Para implementar el constructor de agregar elemento puede serte muy útil la operación *add_at* implementada en el punto 3.

Ej6)

a)

```
implement Conjunto of T where
      type Conjunto of T = List of T

fun conjuntoVacío ( ) ret c : Conjunto
      c := empty()
end fun

proc agregar ( in/out c : Conjunto of T, in e : T )
      addl(c,e)
end proc

proc destruirConjunto ( in/out c : Conjunto of T )
      skip
end proc

fun pertenece ( c : Conjunto of T, e : T ) ret b : bool
      var elem : T
      var i : Nat
      i := 0;
      b := false
      do ( i < length(c) and not b) do
            elem := index(c,i)
            b := elem == e;
            i := i + 1;
      od
end fun

fun esVacio ( c : Conjunto of T ) ret b : bool
      b := length(c) == 0
end fun

proc unir (in/out c  : Conjunto of T, in c2 : Conjunto of T )
      var c_aux : Conjunto of T
      c_aux = copy_list(c2)
      if (esVacio(c)) then
            c = copy_list(c2)
      else
            while ( not is_empty(c_aux) ) do
                  if ( not pertenece(c,head(c_aux)) ) then
                        agregar(c,head(c_aux))
                  fi
                  tail(c_aux)
            od
end proc

proc intersección (in/out c : Conjunto of T, in c2 : Conjunto of T)
    var s_tmp: Conjunto of T
    s_tmp:= copy_list(s)
```

```
    s:= conjuntoVacio()
    for i:= 1 to length(s_tmp) do
        for j:= 1 to length(s1) do
            if(index(s_tmp,i) = index(s1,j)) then
                addr(s,index(s_tmp,i))
            fi
        od
    od
end proc

proc diferencia (in/out c : Conjunto of T, in c2 : Conjunto of T)
      var l_aux : Conjunto of T
      l_aux = copy_list(c)
      while (not is_empty(l_aux)) do
            if( is_elem(c2,head(l_aux))) then
                  tail(c)
                  tail(l_aux)
            else then
                  addr(c,head(l_aux)
                  tail(c)
                  tail(l_aux)
            fi
      od
end proc
```

b)

```
implement Conjunto of T where
```

```
      type Conjunto of T = List of T

fun vacío ( ) ret c : Conjunto
      c := empty()
end fun

proc agregar ( in/out c : Conjunto of T, in e : T )
      var l_aux : List of T
      var n : nat
      l_aux := copy_list(c)
      n := 0
      while ( not is_empty(l_aux) and head(l_aux) < e ) do
            n := n + 1
            tail(l_aux)
      od
      if is_empty(l_aux) or head(l_aux) > e then
            add_at(c, n, e)
      fi
      destroy(l_aux)
end proc

proc destruirConjunto ( in/out c : Conjunto of T )
      skip
end proc

fun is_elem ( c : Conjunto of T, e : T ) ret b : bool
      var elem : T
      b := false
      do ( i < length(c) and not b) do
            elem := index(c,i)
            b := elem == e;
            i := i + 1;
      od
end fun

fun is_empty ( c : Conjunto of T ) ret b : bool
      b := is_empty(c)
end fun




proc unir (in/out c : Conjunto of T, in c2 : Conjunto of T )
      var l_aux : List of T
      l_aux = copy_list(c2)
```

```
      while (not is_empty(l_aux)) do
            if( !is_elem(c,head(l_aux))) then
                  agregar(c,head(l_aux))
            fi
            tail(l_aux)
      od
end proc

proc intersección (in/out c : Conjunto of T, in c2 : Conjunto of T)
      var l_aux : List of T
      l_aux = copy_list(c)
      while (not is_empty(l_aux)) do
            if( is_elem(c2,head(l_aux))) then
                  addr(c,head(l_aux))
                  tail(c)
                  tail(l_aux)
            else then
                  tail(c)
                  tail(l_aux)
            fi
      od
end proc

proc diferencia (in/out c : Conjunto of T, in c2 : Conjunto of T)
      var l_aux : List of T
      l_aux = copy_list(c)
      while (not is_empty(l_aux)) do
            if( is_elem(c2,head(l_aux))) then
                  tail(c)
                  tail(l_aux)
            else then
                  addr(c,head(l_aux)
                  tail(c)
                  tail(l_aux)
            fi
      od
end proc
```