1. Tipos de datos.

   (a) **Especificar** el TAD Urna que permita registrar votos para una elección entre dos partidos (partido X y partido Y). El TAD debe permitir las siguientes operaciones: urna vacía, votar X, votar Y, votar en blanco, juntar dos urnas, averiguar si la urna está vacía, cuál es el número total de votos, si hay al menos un voto X, si hay al menos un voto Y, anular un voto X, anular un voto Y, averiguar si gana X, si gana Y, si empatan.

   (b) **Implementar** el TAD Urna utilizando una representación que le resulte conveniente, de manera de que todas las operaciones sean constantes.

Especificación: (solo tengo que poner el prototipo de la función)

spec Urna where

constructors:

```
fun urna_vacia() ret w: Urna
      w.votos_x := 0
      w.votos_y := 0
      w.votos_blanco := 0
end fun

proc votar_x(in/out u: Urna)
      u.votos_x := u.votos_x + 1
end proc

proc votar_y(in/out u: Urna)
      u.votos_y := u.votos_y + 1
end proc

proc votar_blanco(in/out u: Urna)
u.votos_blanco := u.votos_blanco + 1
end proc
```

operations:

```
proc juntar_urnas(in/out u: Urna, in v: Urna)
      u.votos_x := u.votos_x + v.votos_x
      u.votos_y := u.votos_y + v.votos_y
      u.votos_blanco := u.votos_blanco + v.votos_blanco
end proc

fun empty_urna(u: Urna) ret r: bool
      r := (u.votos_x= 0 && u.votos_y = 0 && u.votos_blanco = 0)
end fun

fun length_urna(u: Urna) ret r:nat
      r = u.votos_x + u.votos_y + u.votos_blancos
end fun
```

```
fun almenos_unvoto_x(u: Urna) ret r: bool
      r := length_urna(u.votos_x) > 0
end fun

fun almenos_unvoto_y(u: Urna) ret r: bool
      r := length_urna(u.votos_y) > 0
end fun

proc anular_voto_x(in\out u: Urna)
      u.votos_x := u.votos_x - 1
end proc

proc anular_voto_y(in\out u: Urna)
      u.votos_y := u.votos_y - 1
end proc

fun ganador(u: Urna) ret r: nat
      if u.votos_y = u.votos_x then
            r := -1
      if u.votos_y > u.votos_x then
            r := 1
      else
            r := 2
      fi
end fun

end spec
```

Implementación:

```
implement Urna where

type Urna = tuple
            votos_x: nat
            votos_y: nat
            votos_blanco: nat
      end tuple

end implement
```

TAD CONJUNTO

```
spec Conjunto of T where
```

Constructors:

```
      fun vacio() ret c: Conjunto of T
      {- Devuelve el conjunto vacio -}

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

Operations:

```
      fun pertenece(c: Conjunto of T, e: T) ret b: Bool
      {- Chequea si un elemento de tipo T e, pertenece a c -}

      fun esvacio(c: Conjunto) ret b: bool
      {- Chequea si un conjunto c es vacio -}

      proc union(in/out c: Conjunto of T, in c2: Conjunto of T)

      proc interseccion(in/out c: Conjunto of T, in c2: Conjunto of T)

      proc diferencia(in/out c: Conjunto of T, in c2: Conjunto of T)

      proc destroy(in/out c: Conjunto of T)

end spec

implement Conjunto of T where

      type Conjunto of T = List of T

fun vacio() ret c: Conjunto of T
      c := empty()
end fun

fun esvacio(c: Conjunto) ret b: bool
      b = (c == null)
end fun

proc agregar(in/out c: Conjunto of T, in e: T)
      var l_aux: List of T
      var n: nat
      var is_member: bool
      var d: T

      l_aux := copy_list(c)
      n := 0
      is_member := false

      while (not is_empty(l_aux) /\ not member) do
            d := head(l_aux)
            if d = e -> is_member := true
               d < e -> n := n+1
               d > e -> skip
            fi
            tail(l_aux)
      od
      if not is_member then
            add_at(c, n, e)
      fi
      destroy(l_aux)
end proc
```

```
proc inters(in/out s: Set of T, in s0: Set if T)
      var s_aux: List of T
      var d: T

      s_aux := list_copy(s)

      while not is_empty(s_aux) do
            d := head(s_aux)
            if not member(d, s0) then
                  elim(s, d)
            fi
            tail(s_aux)
      od
      list_destroy(s_aux)
end proc

fun pertenece(a: Set of T, n: nat) ret queck: bool
      var a_aux: List of T
      var w: nat
      queck := false
      a_aux := copy(a)
      while (a_aux != Null && !member) do
            w : get(a_aux)
            if w = n then
                  queck := true
            fi
            elim(a_aux, w)
      od
end fun

proc destroy(in/out c: Conjunto of T)
      var c_aux: List of T
      while c != null do
            c_aux := c
            s = s->next
            free(c_aux)
      od
end proc
```

TAD: Pila (con arreglos)

```
TAD pila[elem]

Constructores
      vacia: pila
      apilar: elem x pila -> pila
Operaciones
      primero: pila -> elem
      desapilar: pila-> pila
      es_vacia: pila-> bool
```

Entonces:

```
type stack = tuple
            elems: array[1..n] of elemen
            size: nat
      end

proc empty(out p: stack)
      p.size := 0
end

proc push(in e: elem, in/out p: stack)
      p.size = p.size + 1
      p.elem[p.size] := e
end proc

proc top(p: stack)
      p.elem[p.size]
end proc

proc pop(in/our p: stack)
      p.size := p.size-1
end proc

proc is_empty(p: stack) ret b: bool
      b := (p.size == 0)
end proc

fun is_full(p: stack) ret bool
      b := (p.size == N)
end fun
```

acá si hago push desde la izquierda tendría que correr todos los elementos del arreglo para introducir el nuevo Ej

```
proc addl(in e: T, in/out l: List of T)
      for i:=l.size downto 1 do
            l.elems[i+i] := l.elems[i]
      od
      l.elems[1] := e
      l.size := l.size + 1
end proc
```

TAD: Cola (con arreglos)

```
type queue = tuple
            elems: array[0..n-1] of elem
            size: nat
            fst: nat
      end

proc empty(out q: queue)
      q.size := 0
```

```
      q.fst := 0
end proc

proc enqueue(in/out q: queue, in elem: nat)
      q.elems[(q.fst+q.size) mod n] := e
      q.size := q.size + 1
end proc

fun fist(q: queue) ret elem
      e := q.elems[q.fst]
end fun

proc dequeue(in/out q: queue)
      q.size := q.size -1
      q.fst := (q.fst + 1) mod n
end proc

fun is_empty(q: queue) ret b: bool
      b := (q.size == 0)
end fun

fun is_full(q: queue) ret b: bool
      b := (q.size == n)
end fun
```

TAD: Pila (con Listas enlazadas -Punteros-)

```
type node = tuple
            value: elem
            next: pointer to node
      end

type list = pointer to node

Para la pila

type node = tuple
            value: elem
            next: pointer to node
      end

type stack = pointer to node

proc empty(out p: stack)
      p := null
end proc

proc push(in/out p: stack, in e: nat)
      var p_aux: pointer to node
      alloc(p_aux)

      q->value := e
```

```
      q->next := p
      p := q
end proc

fun top(p: stack) ret e: elem
      e := p->value
end fun

proc pop(in/out p: stack)
      var p_aux: pointer to node
      p_aux = p
      p := p->next
      free(p_aux)
end proc

fun is_empty(p: stack) ret b: bool
      b = p == null
end fun

proc destroy(in/out p: stack)
      while not is_empty(p) do
            pop(p)
      od
end proc
```

TAD: Cola (con Listas enlazadas -Punteros-)

```
type node = tuple
            value: elem
            next: pointer to node
      end

type list = pointer to node

Para la cola

type node = tuple
            value: elem
            next: pointer to node
      end

type queue = pointer to node

proc enqueue(in/out q: queue, in e: nat)
      var q_aux: pointer to node
      var r: pointer to node

      alloc(q_aux)

      q_aux->elem := e
      q_aux->next = null

      if q = null then
```

```
            q_aux := q
      fi
      if q != null then
            r := q
            while r->next != null do
                  r := r->next
            od
            r->next := q
      fi
end proc
```

_____ Queue con operaciones constantes.

```
type queue = tuple
            fst: pointer to node
            lst: pointer to node
      end

proc empty(out p: queue)
      p.fst = null
      p.lst = null
end proc

fun first(p: queue) ret e: elem
      e := p->fst->value
end fun

proc enqueue(in/out p: queue, in e: elem)
      var q: pointer to node

      alloc(q)
      q->value := e
      q->next := null

      if p.lst = null then
            p.fst := q
            p.lst := q
      fi
      if p.lst != null then
            p.lst->next := q
            p.lst  := q
      fi
end proc

proc dequeue(in/out p: queue)
      var q: pointer to node
      q := p.fst
      if p.fst = p.lst then
            p.fst := null
            p.lst := null
      fi
      if p.fst != p.lst then
            p.fst := p.fst->next
```

```
      fi
      free(q)
end proc

fun is_empty(p: queue) ret b: bool
      b := p.fst = null
end fun
proc destroy(in/out p: queue)
      while not is_empty(p) do
            dequeue(p)
      od
end proc
```

TAD: Cola de prioridades (con Listas enlazadas -Punteros-)

---

5. El TAD VipQueue es una variante del tipo abstracto Queue que cuenta con un constructor adicional **enqueueVip**, el cual modifica la VipQueue agregándole un elemento de modo "preferencial" o "vip". También cuenta con una operación adicional **hayVip** que indica si en la VipQueue hay algún elemento que se haya agregado de modo vip.

   El resto de las operaciones tienen el mismo tipo que en la versión Queue original pero su comportamiento es modificado:

   La operación **first** devuelve el primer elemento que haya ingresado como vip, o el primer elemento que haya ingresado de modo normal, en caso que no haya ningún vip. La operación **dequeue** devuelve la VipQueue que resulta de eliminar el elemento que la operación **first** devolvería.

   Se pide:

   (a) Escribí la especificación completa del tipo VipQueue.

   (b) Implementá el tipo VipQueue utilizando una estructura que contenga dos arreglos de tamaño N y dos números naturales.

```
type VipQueue of T = tuple
                  elems1: array[1..n] of T
                  size: nat
                  Vipelems: array[1..n] of T
                  vipsize: nat
            end tuple

type VipQueue = pqueue

proc empty(out q: pqueue)
      q.size := 0
      q.vipsize := 0
end proc

fun hayVip(q: pqueue) ret b: bool
      if q.vipsize != 0 then
            b := true
      fi
end fun

proc enqueue(in/out q: pqueue, in e: elem)
      q.size := q.size + 1
      q.elems[q.size] := e
end proc
```

```
proc VipEnqueue(in/out q: pqueue, in e: elem)
      q.vipsize := q.vipsize + 1
      q.vipelems[q.vipsize] := e
end proc


fun firs(q: pqueue) ret e:elem
      if q.vipsize != 0 then
            e := q.vipelems[1]
end fun


proc dequeue(in/out q: pqueue)
      q.size := q.size - 1
end proc

fun is_empty(q: pqueue) ret b: bool
      b := (q.size = 0) & b := (q.vipsize = 0)
end fun
```

_____ TAD wallet.

5. (TADs) Te contratan para diseñar un software de billetera virtual, la cual puede almacenar saldo en tres monedas distintas: Peso, Real y Dólar. Cada usuario puede recibir pagos en cualquiera de las tres monedas, y puede realizar pagos en alguna de las tres monedas, siempre y cuando tenga el saldo suficiente.

   Se pide:

   (a) **Especificar** el TAD **Wallet** mediante un constructor que cree la wallet con saldo 0 en las tres monedas. Además debe proveer operaciones para averiguar cuál es el saldo en cada moneda, para recibir pagos en cada una de las monedas y para realizar pagos con cada una de ellas también. Las operaciones de recibir y realizar pagos deben especificarse como **procedimientos** que modifiquen una Wallet.

   La especificación debe realizarse utilizando el lenguaje visto en la materia, indicando el tipo de cada operación, la precondición en caso que tenga y un comentario describiendo qué hace.

   (b) **Implementar** el tad utilizando una tupla con 3 números. Se debe utilizar precisamente el lenguaje de la materia.

   (c) Utilizando el tipo **abstracto**, se debe implementar una operación que, dado un número racional indicando la relación entre dólar y peso, modifique una wallet convirtiendo todos los dólares que tenga en el saldo, a su correspondiente en pesos. Se debe utilizar precisamente el lenguaje de la materia.

Tres monedas distintas -> peso, real y dólar.

<u>Especificación:</u>

```
spec Wallet where

constructors
fun wallet_vacia() ret Wallet
fun saldo_pesos(w: Wallet) ret r: real
fun saldo_euros(w: Wallet) ret r: real
fun saldo_dolar(w: Wallet) ret r: real

operations
proc realizar_pago(in/out w: Wallet, in v: real)
proc recibir_pago(in/out w: wallet, in v: real)

end spec
```

Implementación:

```
implement Urna where

type Urna = tuple
            pesos: real
            dolares: real
            euros: real
      end tuple

end implement

fun init() ret w: Wallet
      w.pesos = 0
      w.reale = 0
      w.dolares = 0
end fun

fun saldo_pesos(w: Wallet) ret r: real
      res:= w.pesos
end fun

proc pago_pesos(in/out w: wallet, c: nat)
      w.pesos = w.pesos - c
end proc

proc recibir_pesos(in/out w: wallet, c: nat)
      w.pesos = w.pesos + c
end proc
```

(a) A partir de la siguiente implementación de listas mediante punteros, implemente las operaciones copy_list, tail y concat.

```
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


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
```

```
fun copy_list(l: List of T) ret p: List of T
      var q: pointer to (Node of T)
      p := empty()
      q := l

      while not is_empty(q) && q->next != NULL do
            p := addl(q->elem, p)
            q := q->next
      do
end fun

proc tail(in/out l: List of T)
      var p: pointer to (Node of T)
      p := l
      l := l->next
      free(p)
end proc

proc concat(in/out l: List of T, in l0: List of T)
      var p: pointer to (Node of T)
      var q: pointer to (Node of T)

      q := l0
      if not is_empty(l) then
            p := l
            while p->next != null do
                  p = p->next
            od
            alloc(p->next)
            p->next := q
      else
            l := q
      fi
end proc
```

---

5. A partir de la siguiente implementación de conjuntos utilizando listas ordenadas, implemente el constructor **add**, y las operaciones **member**, **inters** y **cardinal**. La implementación debe mantener el invariante de representación por el cual todo conjunto está representado por una lista ordenada crecientemente. Puede utilizar todas las operaciones especificadas para el tipo lista vistas en el teórico. Para cada operación que utilice, especifique su encabezado, es decir: si es función o procedimiento, cómo se llama, qué argumentos toma y devuelve.

```
implement Set of T where

type Set of T = List of T

fun empty_set() ret s : Set of T
    s := empty_list()
end fun
```

```
proc add(in r: T, in/out s: Set of T)
      var l_aux: List of T
      var n: nat

      l_aux := copy_list(c)
      n := 0
```

```
      while not is_empty(l_aux) do
            if head(l_aux) < e then
                  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

fun cardinal(s: Set of T) ret n: nat
      n := list_length(s)
end fun

proc inters(in/out s: Set of T, in s0: Set if T)
      var s_aux: List of T
      var d: T

      s_aux := list_copy(s)

      while not is_empty(s_aux) do
            d := head(s_aux)
            if not member(d, s0) then
                  elim(s, d)
            fi
            tail(s_aux)
      od
      list_destroy(s_aux)
end proc

fun member(a: Set of T, n: nat) ret queck: bool
      var a_aux: List of T

      queck := false
      a_aux := a
      while (a_aux != Null && !member) do
            queck := a_aux->elem == n
            a_aux := a_aux->next
      od
end fun
```

- Esto es en C -

```
set set_elim(set s, set_elem e) {
      if (s != NULL) {
            if (s->elem == e) {
                  set aux = s;
                  s = s->next
                  free(aux)
            } else {
                  set node = s->next, father = s;
                  while (node != NULL && node->elem != e) {
```

```
                        father = node;
                        node = node->next;
                    }
                    if (node != NULL) {
                        assert(node->elem == e);
                        set aux = father->next;
                        father->next = node->next;
                        free(aux);
                    }
                }
            }
            return s;
}
```

---

4. Dada la especificación del tad Cola:

spec Queue **of** T **where**

constructors

fun empty_queue() **ret** q : Queue **of** T
{- *crea una cola vacía.* -}

proc enqueue (**in**/**out** q : Queue **of** T, **in** e : T)
{- *agrega el elemento* e *al final de la cola* q. -}

operations

fun is_empty_queue(q : Queue **of** T) **ret** b : Bool
{- *Devuelve* True *si la cola es vacía* -}

fun first(q : Queue **of** T) **ret** e : T
{- *Devuelve el elemento que se encuentra al comienzo de* q. -}
{- **PRE:** not is_empty_queue(q) -}

proc dequeue (**in**/**out** q : Queue **of** T)
{- *Elimina el elemento que se encuentra al comienzo de* q. -}
{- **PRE:** not is_empty_queue(q) -}

Implementá los constructores y operaciones del TAD utilizando la siguiente representación, donde N es una constante de tipo nat:

implement Queue **of** T **where**

type Queue **of** T = **tuple**
elems : array[0..N-1] **of** T
size : nat
end tuple

```
fun empty_queue() ret q: Queue of T
      q.size := 0
end fun

proc enqueue(in/out q: Queue of T, in r: T)
      q.size := q.size + 1
      q.elems[q.size] := e
end proc

fun is_empty_queue(q: Queue of T) ret b: bool
```

```
      b := (q.size = 0)
end fun

fun first(q: Queue of T) ret e: T
      e := q.elems[1]
end fun

proc dequeue(in/out q: Queue of T)
      q.size := q.size - 1
      q.elems[1] = (q.elems[1] + 1 mod N)
end poc
```

(a) Implementá los constructores del TAD Conjunto de elementos de tipo T, y las operaciones member, elim e inters, utilizando la siguiente representación:

implement Set **of** T **where**

type Set **of** T = **tuple**
elems : array[0..N-1] **of** T
size : nat
end tuple

¿Existe alguna limitación con esta representación de conjuntos? En caso afirmativo indicá si algunas de las operaciones o constructores tendrán alguna precondición adicional.

NOTA: Si necesitás alguna operación extra para implementar lo que se pide, debes implementarla también.

(b) Utilizando el tipo **abstracto** Conjunto de elementos de tipo T, implementá una función que reciba un conjunto de enteros s, un número entero i, y obtenga el entero perteneciente a s que está *más cerca* de i, es decir, un $j \in s$ tal que para todo $k \in s$, $|j-i| <= |k-i|$. Por ejemplo si el conjunto es $1, 5, 9$, y el entero 7, el resultado puede ser 5 o 9.

```
proc inters(in/out s: Set of T, in s0: Set if T)
      var s_aux: List of T
      var d: T

      s_aux := list_copy(s)

      for i:=0 to N-1 do
            d := head(s_aux)
            if not member(d, s0) then
                  elim(s, d)
            fi
            tail(s_aux)
      od
      list_destroy(s_aux)
end proc

fun member(a: Set of T, n: nat) ret queck: bool
      var a_aux: List of T
      var w: nat
      queck := false
      a_aux := copy(a)
      while (a_aux != Null && !member) do
            w : get(a_aux)
            if w = n then
```

```
                  queck := true
            fi
            elim(a_aux, w)
      od
end fun

proc elim(in/out s: Set of T, in e: T)
      var a_aux: List of T

      a_aux := copy_set(s)

      for i:=0 to N-1 do
            if member(a_aux, e) then
                  w := get(a_aux.elems[i])
                  elim(a_aux, e)
            fi
      od
      destroy(a_aux)
end proc
```

-version buena-

```
proc elim(in/out s: Set of T, in e: T)
      var aux_s: List of T
      var node: List of T
      var node_aux: List of T

      if (not is_empty(s)) then
            if s->elem = e then
                  aux_s = s
                  s = s->next
                  free(aux_s)
            fi
      else
            node = s->next
            node_aux = s
            while (node != null && node->elem != e) do
                  node_aux = node
                  node = node->next
            od
            if (node != null) then
                  var aux = node_aux->next
                  node_aux->next = node->next
                  free(aux)
            fi
end proc

fun a(s: Set of T, e: int) ret r: int
      var s_aux: Set of T

      s_aux := copy_set(s)

      for i:=0 to N-1 do
```

```
            if not member(s_aux, e) then
                  if abs(a[i]-e) <= abs(a[i+1]-e) then
                        r := i
                  fi
            fi
      od
      destroy(s_aux)
end fun


|1-7| <= |5-7| -> 6 <= 2 no
|5-7| <= |9-7| -> 2 <= 2 si
```

```
implement List of T where

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

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

{- agrega el elemento e al comienzo de la lista l. -}
{- PRE: l.size < N -}
proc addl (in e : T, in/out l : List of T)
  {- acá primero debemos correr una posición al a derecha los elementos en
el arreglo -}
  for i := l.size downto 1 do
       l.elems[i+1] := l.elems[i]  {- observación: la 1er asignación es
                                      l.elems[l.size+1] := l.elems[l.size]. ANDA-}
  od
  l.elems[1] := e
  l.size := l.size + 1
end proc
```