1. Implementá el TAD Pila utilizando la siguiente representación:

   implement Stack **of** T **where**

   type Stack **of** T = List **of** T

Ej-1)

```
spec Stack of T where

constructors

      fun empty_stack() ret s : Stack of T
      {- crea una pila vacía. -}

      proc push (in e : T, in/out s : Stack of T)
      {- agrega el elemento e al tope de la pila s. -}

operations

      fun is_empty_stack(s : Stack of T) ret b : Bool
      {- Devuelve True si la pila es vacía -}

      fun top(s : Stack of T) ret e : T
      {- Devuelve el elemento que se encuentra en el tope de s. -}
      {- PRE: not is_empty_stack(s) -}

      proc pop (in/out s : Stack of T)
      {- Elimina el elemento que se encuentra en el tope de s. -}
      {- PRE: not is_empty_stack(s) -}

implement Stack of T where

      type Stack of T = List of T

fun empty_stack() ret s : Stack of T
      s := empty()
end fun



proc push (in e : T, in/out s : Stack of T)
      addr(s,e)
end proc
```

```
fun is_empty_stack(s : Stack of T) ret b : Bool
      b := is_empty(s)
end fun

fun top(s : Stack of T) ret e : T
      var l_aux : List of T
      l_aux := copy_list(s)
      var n : nat
      n := 0
      while ( not is_empty(l_aux) ) do
            tail(l_aux)
            n := n + 1
      od
      e := index(s,n-1)
end fun

proc pop (in/out s : Stack of T)
      var n : nat
      n := length(s)
      take(s, n - 1)
end proc

end implement
```

2. Implementá el TAD Pila utilizando la siguiente representación:

```
implement Stack of T where

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

type Stack of T = pointer to (Node of T)
```

Ej-2)

```
constructors

      fun empty_stack() ret s : Stack of T
            s := NULL
      end fun

      proc push (in e : T, in/out s : Stack of T)
            var p,q : pointer to (Node of T)
            alloc(q)
            q→elem := e
            q →next := null
            if ( not  s == Null )
                  then p := s
                        do ( p→next != null)
                              p := p→next
                        od
                        p →next := q
                  else s := q
            fi
      end proc

operations

      fun is_empty_stack(s : Stack of T) ret b : Bool
            b := s ==  null
      end fun









      fun top(s : Stack of T) ret e : T
            var q : pointer to (Node of T)
            q := s
            while( not is_empty_stack(q) )
                  q := q→ next
                  if ( q→next == null ) then
                        e := q→elem
                  fi
            od
      end fun
```

```
        proc pop (in/out s : Stack of T)
              var q : pointer to (Node of T)
              q := s
              while ( (q→next) → next != Null) do
                    q := q → next
              od
              free(q→next)
              q →next := Null
        end proc

end implement
```

3. (a) Implementá el TAD Cola 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
```

   (b) Implementá el TAD Cola utilizando un arreglo como en el inciso anterior, pero asegurando que todas las operaciones estén implementadas en orden constante.
   Ayuda1: Quizás convenga agregar algún campo más a la tupla. ¿Estamos obligados a que el primer elemento de la cola esté representado con el primer elemento del arreglo?
   Ayuda2: Buscar en Google *aritmética modular*.

Ej-3a)

```
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. -}
```

```
destroy (skip)


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) -}

end spec



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 e : T)
{- agrega el elemento e al final de la cola q. -}
      q.elems[size+1] = e
      q.size = q.size + 1
end proc

fun is_empty_queue(q : Queue of T) ret b : Bool
      b := q.size == 0
end fun

{- PRE: not is_empty_queue(q) -}
fun first(q : Queue of T) ret e : T
      e = q.elems[0]
end fun
```

```
{- PRE: not is_empty_queue(q) -}
proc dequeue (in/out q : Queue of T)
      for(i := 0 to (N - 1) - 1) do
            q.elems[i] := q.elems[i+1]
      od
      q.size := (N - 1) - 1
end proc

end implement
```

Ej-3b)

```
implement Queue of T where
      type Queue of T = tuple
                        elems : array[0..N-1] of T
                        size : nat
                        start : nat
                  end tuple

{- crea una cola vacía. -}
fun empty_queue() ret q : Queue of T
      var q : Queue of T
      q.size = 0
      q.sart = 0
end fun

{- agrega el elemento e al final de la cola q. -}
```

```
proc enqueue (in/out q : Queue of T, in e : T)
    var pos : nat
    pos := (q.start + q.size) % N
    q.elem[pos] := e
    q.size := q.size + 1
end proc

{- Devuelve True si la cola es vacía -}
fun is_empty_queue(q : Queue of T) ret b : Bool
      b := s.size == 0
end fun

{- Devuelve el elemento que se encuentra al comienzo de q. -}
{- PRE: not is_empty_queue(q) -}
fun first(q : Queue of T) ret e : T
      e := q.elems[q.sart]
end fun

{- Elimina el elemento que se encuentra al comienzo de q. -}
{- PRE: not is_empty_queue(q) -}
proc dequeue (in/out q : Queue of T)
    if q.start == N - 1 then
      q.start := 0
    else →
      q.start := q.start + 1
      q.size := q.size - 1
end proc
```

4. Completá la implementación del tipo Árbol Binario dada en el teórico, donde utilizamos la siguiente representación:

```
implement Tree of T where

type Node of T = tuple
                    left: pointer to (Node of T)
                    value: T
                    right: pointer to (Node of T)
                end tuple

type Tree of T= pointer to (Node of T)
```

Ej-4)

```
type Direction =    enumerate
                      Left
                      Right
                    end enumerate
```

```
type Path = List of Direction

spec Tree of T where

constructors

      fun empty_tree() ret t : Tree of T
      {- crea una árbol vacío. -}

      fun node (tl : Tree of T, e : T, tr : Tree of T) ret t : Tree of T
      {- crea el nodo con elementos e y subárboles tl y tr. -}

      proc destroy_t (in/out t : Tree of T )
      {- Libera memoria en caso de ser necesario. -}

operations

      fun is_empty_tree(t : Tree of T) ret b : Bool
      {- Devuelve True si el árbol es vacío -}

      fun root(t : Tree of T) ret e : T
      {- Devuelve el elemento que se encuentra en la raíz de t. -}
      {- PRE: not is_empty_tree(t) -}

      fun left(t : Tree of T) ret tl : Tree of T
      {- Devuelve el subárbol izquierdo de t. -}
      {- PRE: not is_empty_tree(t) -}

      fun right(t : Tree of T) ret tl : Tree of T
      {- Devuelve el subárbol derecho de t. -}
      {- PRE: not is_empty_tree(t) -}



      fun height(t : Tree of T) ret n : Nat
      {- Devuelve la dist. que hay entre la raíz de t y la hoja más profunda. -}

      fun is_path(t : Tree of T, p : Path) ret b : Bool
      {- Devuelve True si p es un camino válido en t -}

      fun subtree_at(t : Tree of T, p : Path) ret t0 : Tree of T
      {- Devuelve el subárbol que se encuentra al recorrer el camino p en t. -}

      fun elem_at(t : Tree of T, p : Path) ret e : T
      {- Devuelve el elemento que se encuentra al recorrer el camino p en t. -}
      {- PRE: is_path(t,p) -}

implement Tree of T where

      type Node of T = tuple
```

```
                        left: pointer to (Node of T)
                        value: T
                        right: pointer to (Node of T)
                    end tuple

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

constructors

    fun empty_tree() ret t : Tree of T
        t := null
    end fun

    fun node (tl : Tree of T, e : T, tr : Tree of T) ret t : Tree of T
        alloc(t)
        t->value := e
        t->left := tl
        t->right := tr
    end fun

operations

    fun is_empty_tree(t : Tree of T) ret b : Bool
        b := t == null
    end fun

    {- PRE: not is_empty_tree(t) -}
    fun root(t : Tree of T) ret e : T
        e := t->value
    end fun



    {- PRE: not is_empty_tree(t) -}
    fun left(t : Tree of T) ret tl : Tree of T
        tl := t->left
    end fun
    {- PRE: not is_empty_tree(t) -}
    fun right(t : Tree of T) ret tl : Tree of T
        tl := t->right
    end fun

    fun height(t : Tree of T) ret n : Nat
        if (is_empty_tree(t)) then
            n := 0
        else
            n := (height(t->left) max height(t->right)) + 1
        fi
    end fun

    fun is_path(t : Tree of T, p : Path) ret b : Bool
```

```
	if (is_empty(p)) then
		b := true
	else if (is_empty_tree(t) and not is_empty(p))
		b := false
	else
		b := true
		var pc : List of Directions
		pc := copy_list(p)
		while (not is_empty(pc) and b) do
			if (head(pc) == Left) then
				if (t2->left == null) then
					b := false
				else
					t2 := t->left
				fi
			else if (head(pc) == Right) then
				if (t2->right == null) then
					b := false
				else
					t2 := t->right
				fi
			tail(pc)
		od
		destroy(pc)
	fi
end fun
```

```
fun subtree_at(t : Tree of T, p : Path) ret t0 : Tree of T
	var p2 : List of Directions
	p2 = copy_list(p)
	var t2 : Tree of T
	t2 := t
	while(not is_empty(p2) and not_is_empty(t2)) do
		if (head(p2) == Right) then
			t2 := t2->right
		else
			t2 := t2->left
		fi
		tail(p2)
	od
	destroy(p2)
	t0 := t2
end fun

{- PRE: is_path(t,p) -}
```

```
fun elem_at(t : Tree of T, p : Path) ret e : T
	var p2 : List of Directions
	p2 = copy_list(p)
	while(not is_empty(p2)) do
		if (head(p2) == Right) then
			t2 := t2->right
		else
			t2 := t2->left
		fi
		tail(p2)
	od
	destroy(p2)
	e := t2->value
end fun
```

Ej-5)

Un Diccionario es una estructura de datos muy utilizada en programación. Consiste de una colección de pares (Clave,Valor), a la cual le puedo realizar las operaciones:

• Crear un diccionario vacío.
• Agregar el par consistente de la clave k y el valor v. En caso que la clave ya se encuentre en el diccionario, se reemplaza el valor asociado por v.
• Chequear si un diccionario está vacío.
• Chequear si una clave se encuentra en el diccionario.
• Buscar el valor asociado a una clave k. Solo se puede aplicar si la misma se encuentra.
• Una operación que dada una clave k, elimina el par consistente de k y el valor asociado. Solo se puede aplicar si la clave se encuentra en el diccionario.
• Una operación que devuelve un conjunto con todas las claves contenidas en un diccionario

Ej-5a)

Especifica el TAD diccionario indicando constructores y operaciones.

spec Dict of (K,V) where

donde K y V pueden ser cualquier tipo, asegurando que K tenga definida una función que chequea igualdad.

```
spec Dict of (K,V) where

constructors

	fun empty_dict( ) ret d : Dict of (K,V)
	{- Crea un diccionario vacío -}

	proc add_to_dict(in/out d : Dict of (K,V), in k : K, in v : V)
	{- Agrega la clave k, y el valor v al diccionario -}
```

```
        proc destroy_dict(in/out d : Dict of (K,V))
        {- Libera memoria en caso de ser necesario -}

operations

        fun is_empty_dict( d : Dict of (K,V)) ret b : Bool
        {- Verifica si el diccionario es vacío -}

        fun is_k( d : Dict of (K,V), k : K) ret b : Bool
        {- Verifica si la clave k esta en d -}

        {- PRE: is_k(d,k) -}
        fun index_k ( d : Dict of (K,V), k : K ) ret v : V
        {- Devuelve el elemento asociado a k -}

        {- PRE: is_k(d,k) -}
        proc elim_elem(in/out d : Dict of (K,V), k : K)

        fun k_set( d : Dict of (K,V)) ret c : Conjunto of K
        {- Devuelve un conjunto de tipo k con todas las claves de d -}
```

Ej-5b)

```
implement Dict of (K,V) where

type Node of (K,V) = tuple
                        left: pointer to (Node of (K,V))
                        key: K
                        value: V
                        right: pointer to (Node of (K,V))
                     end tuple

type Dict of (K,V)= pointer to (Node of (K,V))
Constructors

        fun empty_dict( ) ret d : Dict of (K,V)
                d := null
        end fun

        proc add_to_dict(in/out d : Dict of (K,V), in k : K, in v : V)
                if (d = null) then
                        var d2 : Dict of (K,V)
                        alloc(d2)
                        d2->key := k
                        d2->value := v
                        d2->right := null
                        d2->left := null
                        d := d2
                else
                        if ( d->key > k ) then
                                add_to_dict(d->right,k,v)
                        else if ( d->key < k )
```

```
                        add_to_dict(d->left,k,v)
                else ( d->key = k )
                        d->value := v
                fi
        end proc

        proc destroy_dict(in/out d : Dict of (K,V))
                if(not is_empty_dict(d)) then
                        destroy_dict(d->left)
                        destroy_dict(d->right)
                        free(d)
                        d := null
                fi

operations

        fun is_empty_dict( d : Dict of (K,V)) ret b : Bool
                b := d == null
        end fun

        fun is_k( d : Dict of (K,V), k : K) ret b : Bool
                if(is_empty_dict(d)) then
                        b := false
                else
                        if(d->key > k)
                                b := is_k(d->left,k)
                        else if(d->key < k)
                                b := is_k(d->right,k)
                        else
                                b := true
                        fi
                fi
        end fun
        {- PRE: is_k(d,k) -}
        fun index_k ( d : Dict of (K,V), k : K ) ret v : V
                if(d->key = k) then
                        v := d->value
                else if(d->key > k)
                        v := index_k(d->left,k)
                else
                        v := index_k(d->right,k)
        end fun

        {- PRE: is_k(d,k) -}
        proc elim_elem(in/out d : Dict of (K,V), k : K)
                var max_key : K
                var max_val : V
                if (d->key = k) then
                        {- eliminar -}
                        if (d->left = null) then
                                var d2 : Dict of (V,K)
```

```
                        d2 := d
                        d := d->right
                        free(d2)
                  else
                  {- buscamos el máximo k de la izquierda (también podría ser
      buscando el mínimo de la derecha) -}
                        borra_max_key_val(d->left, max_key, max_val)
                        d->key := max_key
                        d->value := max_val
                  fi
            else if (d->key > k) then
                  elim_elem(d->left,k)
            else
                  elim_elem(d->right,k)
            fi
      end proc

      proc borra_max_key_val(in/out d : Dict of (V,K), out k : K, out v : V)
            var d2 : Dict of (V,K)
            if (d->right = null)
                  k := d->key
                  v := d->value
                  d2 := d
                  d := d->left
                  free(d2)
            else
                  borra_max_key_val(d->right,k,v)
            fi
      end proc
```

```
fun k_set( d : Dict of (K,V)) ret c : Conjunto of K
{- Devuelve un conjunto de tipo k con todas las claves de d -}
      if(not is_empty_dict(d)) then
            agregar(c,d->key)
            if(not is_empty_dict(d->left)) then
                  k_set(d->left,c)
            fi
            if(not is_empty_dict(d->right)) then
                  k_set(d->right,c)
            fi
      else
            c := conjuntoVacio()
      fi
end fun
```

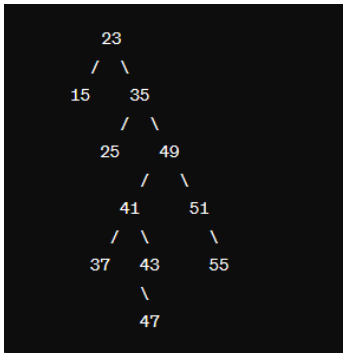
Ej-6)

En un ABB cuyos nodos poseen valores entre 1 y 1000, interesa encontrar el número 363.

¿Cuáles de las siguientes secuencias no puede ser una secuencia de nodos examinados según el algoritmo de búsqueda? ¿Por qué?
(a) 2, 252, 401, 398, 330, 344, 397, 363. SI.
(b) 924, 220, 911, 244, 898, 258, 362, 363. SI.
(c) 925, 202, 911, 240, 912, 245, 363. NO, pues 911 < 912.
(d) 2, 399, 387, 219, 266, 382, 381, 278, 363. SI.
(e) 935, 278, 347, 621, 299, 392, 358, 363. NO, pues 294 < 347

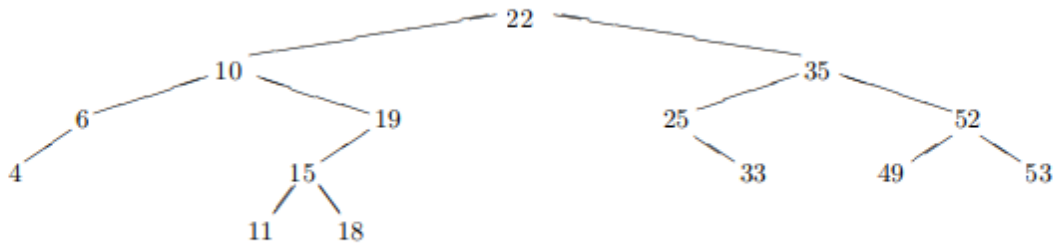
Ej-7)

Dada la secuencia de números 23, 35, 49, 51, 41, 25, 50, 43, 55, 15, 47 y 37, determinar el ABB que resulta al insertarlos exactamente en ese orden a partir del ABB vacío.



Ej-8)

8. Determinar al menos dos secuencias de inserciones que den lugar al siguiente ABB:



22-35-52-53-49-25-33-10-6-19-4-15-11-18

4-6-10-11-15-18-19-22-25-33-35-49-52-53

22-10-19-15-18-11-6-4-35-52-53-49-25-33

Preguntas:
-)En qué momento debería de hacer alloc dentro de una fun/proc.
-)Cuando copio listas debo eliminarlas con destroy dentro de un proc o fun
-)Puedo generar precondiciones nuevas en la implementación de subtree_at

22-4-6-10-11-18-15-19-25-33-35-49-52-53