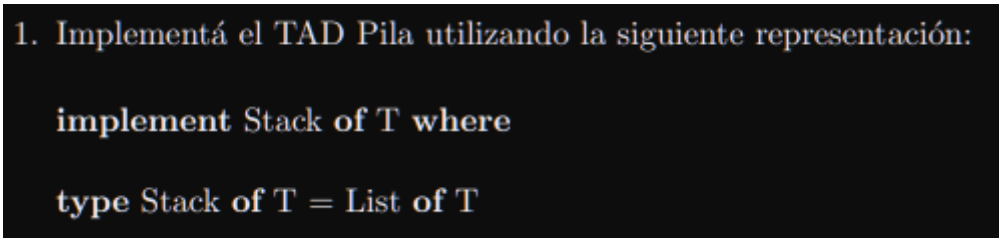
Práctico 2.3

1)




```
type Stack of T = List of T
```

Constructores:

```
fun empty_stack() ret s : Stack of T
      s ≔ empty_list

proc push (in e : T, in/out s : Stack of T)
      addl(r, s)
```

Destructor:

```
proc destroy_stack(in/out s : Stack of T)
      destroy_list(s)
```

Operaciones:

```
fun is_empty_stack(s : Stack of T) ret b : Bool
      b ≔ is_empty_list(s)

fun top(s : Stack of T) ret e : T
      e ≔ head(s)

proc pop(in/out s : Stack of T)
      tail(s)

fun copy_stack(s : Stack of T) ret t : Stack of T
      t ≔ copy_list(s)
```

3)

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*.

3a)

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

Constructores:

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

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

Operadores:

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

fun first(q : Queue of T) ret e : T
    e ≔ (q.elems)[0]

proc dequeue(in/out q : Queue of T)
    for j ≔ 0 to q.size - 1 do
        (q.elems)[j] ≔ (q.elems)[j+1]
    od
    q.size ≔ q.size - 1
```

3b)

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

Constructores:

```
fun empty_queue() ret q : Queue of T
      q.start ≔ 0
      q.end ≔ 0

proc enqueue(in/out q : Queue of T, in e : T)
      q.end ≔ (q.end + 1) `mod` N
      (q.elems)[q.end] ≔ e
```

Operadores:

```
fun is_empty_queue(q : Queue of T) ret b : Bool
      b ≔ q.start = q.end

fun first(q : Queue of T) ret e : T
      e ≔ (q.start)[0]

proc dequeue(in/out q : Queue of T)
      q.start ≔ (q.start + 1) `mod` N

fun copy(q : Queue of T) ret res : Queue of T
      res.start ≔ q.start
      res.end ≔ q.end
      res.elems ≔ array_copy(q.elems)
end fun
```

4)

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)
```

```
type Node of T = tuple
                    left : Tree of T
                    value : T
                    right : Tree of T
                 end tuple

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

type Direction = enumerate
                    Left
                    Right
                 end enumerate

type Path = List of Direction
```

Constructores:

```
fun empty_tree() ret t : Tree of T
    t ≔ null

fun node(tl : Tree of T, e : T, tr : Tree of T) ret t : Tree of T
    alloc(t)
    t→left ≔ copy_tree(tl)
    t→value ≔ e
    t→right ≔ copy_tree(tr)
end fun
```

Destructor:

```
proc destroy_tree(in/out t : Tree of T)
    if ¬is_empty_tree(t) →
        destroy_tree(t→left)
        destroy_tree(t→right)
        free(t)
    fi
```

Operadores:

```
fun copy_tree(t : Tree of T) res copy
      alloc(res)
      res→value ≔ t→value
      res→left ≔ copy_tree(l→left)
      res→right ≔ copy_tree(l→right)

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

fun root(t : Tree of T) ret e : T
      e ≔ t→value

fun left(t : Tree of T) ret tl : Tree of T
      tl ≔ copy_tree(t→left)

fun right(t : Tree of T) ret tr : Tree of T
      tl ≔ copy_tree(t→right)

fun height(t : Tree of T) ret n : Nat
      if is_empty_tree(t) →
            n ≔ 0
      else
            n ≔ 1 + (height(t→left) `max` height(t→right))
      fi

fun side(t : Tree of T, dir : Direction) ret tl : Tree of T
      if dir = Left →
            tl ≔ t→left
      else
            tl ≔ t→right
      fi

fun is_path(t : Tree of T, p : Path) ret b : Bool
      if is_empty_tree(t) ∨ is_empty_list(p) →
            b ≔ is_empty_list(p) ∧ is_empty_tree(t)
      else
            b ≔ is_path(side(t, head(p)))
      fi

fun subtree_at(t : Tree of T, p : Path) ret t0 : Tree of T
      t0 ≔ empty_tree
      if is_empty_list(p) →
            t0 ≔ copy_tree(t)
      else
            p1 ≔ copy_list(p)
            list_tail(p1)
            t0 ≔ subtree_at(side(t, head(p)), p1)
            destroy_list(p1)
      fi
```

```
fun elem_at(t : Tree of T, p : Path) ret e : T
      root(subtree_at(t, p))
```

5)

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 es 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.

(a) Especificá 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.

(b) Implementá el TAD diccionario utilizando la siguiente representación:

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))
```

Como invariante de representación debemos asegurar que el árbol representado por la estructura sea binario de búsqueda de manera que la operación de buscar un valor tenga orden logarítmico. Es decir, dado un nodo n, toda clave ubicada en el nodo de la derecha n.right, debe ser mayor o igual a n.key. Y toda clave ubicada en el nodo de la izquierda n.left, debe ser menor a n.key. Debes tener especial cuidado en la operación que agrega pares al diccionario.

5a)

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

Constructores:

```
fun empty_dict() ret d : Dict of (K, V)
{- Devuelve el diccionario vacío -}

proc add_to_dict(in k : K, v : V, in/out d : Dict of (K, V))
{- Agrega (k, v) a d, si k ya se encuentra en d, se reemplaza el valora asociado por v -}
```

Destructor:

```
proc destroy_dict(d : in/out Dict of (K, V))
```

Operadores:

```
fun is_empty_dict(d : Dict of (K, V)) ret b : bool
{- Chequea si d es vacío -}

fun is_key_in(k : K, d : Dict of (K, V)) ret b : bool


fun elem_of_key(k : K, d : Dict of (K, V)) ret v : V
{- PRE: is_key_in(k, d) -}

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

fun set_of_keys(d : Dict of (K, V)) ret s : Set of K

fun copy_dict(d : Dict of (K, V)) ret res : Dict of (K, V)
```

5b)

```
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))
```

Constructores:

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

```
proc add_to_dict(in k : K, v : V, in/out d : Dict of (K, V))
      if (is_empty_dict(d)) →
            alloc(d)
            d→key ≔ k
            d→value ≔ v
            d→left ≔ empty_dict()
            d→right ≔ empty_dict()
      else if (k = d→key) →
            d→value ≔ v
      else if (k < d→key) →
            add_to_dict(k, v, d→left)
      else
            add_to_dict(k, v, d→right)
      fi fi fi
```

Destructor:

```
proc destroy_dict(d : in/out Dict of (K, V))
      if ¬is_empty_dict(d) →
            destroy_dict(d→left)
            destroy_dict(d→right)
            free(d)
            d ≔ null
```

Operadores:

```
fun is_empty_dict(d : Dict of (K, V)) ret b : bool
      b ≔ d = empty_dict()

fun is_key_in(k : K, d : Dict of (K, V)) ret b : bool
      b ≔ False
      if (¬is_empty_dict(d)) →
            if (k = d→key) →
                  b ≔ True
            else if (k < d→key) →
                  b ≔ is_key_in(k, d→left)
            else
                  b ≔ is_key_in(k, d→right)
            fi fi
      fi

fun elem_of_key(k : K, d : Dict of (K, V)) ret v : V
      if (k = d→key) →
            v ≔ d→value
      else if (k < d→key) →
            v ≔ elem_of_key(k, d→left)
      else
            v ≔ elem_of_key(k, d→right)
      fi fi
```

```
proc unir(in/out d : Dict of (K, V), in d1 : Dict of (K, V))
      if ¬is_empty_dict(d1) →
            add_to_dict(d1→key, d1→value, d)
            unir(d, d1→left)
            unir(d, d1→right)
      fi

proc delete(in k : K, in/out d : Dict of (K, V))
      if (¬is_empty_dict(d)) →
            if (k = d→key) →
                  var temp : Dict of (K, V)
                  temp ≔ d→right
                  d ≔ d→left
                  unir(d, temp)
                  destroy_dict(temp)
            else if (k < d→key)
                  delete(k, d→left)
            else
                  delete(k, d→right)
            fi fi
      fi
```

{- Pone en k, v el maximo elemento del dict, eliminandolo del mismo-}
{- PRE: ¬is_empty_dict(d) -}

```
proc max_of_dict(in/out d : Dict of (K, V), out k : K, v : V)
      if is_empty_dict(d→right) →
            k ≔ d→key
            v ≔ d→value
            var temp : Dict of (K, V)
            temp ≔ d
            d ≔ d→left
            free(temp)
      else is_empty_dict(d→right) →
            max_of_dict(d→rigth)
      fi
end proc

proc delete(in k : K, in/out d : Dict of (K, V))
      if (¬is_empty_dict(d)) →
            if (k = d→key) →
                  if is_empty_dict(d→right) →
                        var temp : Dict of (K, V)
                        temp ≔ d
                        d ≔ d→left
                        free(temp)
                  else
                        var max_k : K
                            max_v : V
                        max_of_dict(d→right, max_k, max_v)
```

```
            fi
        else if (k < d→key)
            delete(k, d→left)
        else
            delete(k, d→right)
        fi fi
    fi
```

```
fun set_of_keys(d : Dict of (K, V)) ret s : Set of K
    s ≔ emptySet
    if ¬is_empty_dict(d) →
        addToSet(s, d→key)
        unionSet(s, set_of_keys(d→left))
        unionSet(s, set_of_keys(d→right))

fun copy_dict(d : Dict of (K, V)) ret res : Dict of (K, V)
    if is_empty_dict(d) →
        res ≔ empty_dict
    else
        alloc(res)
        res→key ≔ d→key
        res→value ≔ d→value
        res→left ≔ copy_dict(d→left)
        res→right ≔ copy_dict(d→right)
```

6)

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.
   (b) 924, 220, 911, 244, 898, 258, 362, 363.
   (c) 925, 202, 911, 240, 912, 245, 363.
   (d) 2, 399, 387, 219, 266, 382, 381, 278, 363.
   (e) 935, 278, 347, 621, 299, 392, 358, 363.

(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
(d) 2, 399, 387, 219, 266, 382, 381, 278, 363. Si
(e) 935, 278, 347, 621, 299, 392, 358, 363. No

(a) Si
(b) Si

(c) No
(d) Si
(e) No

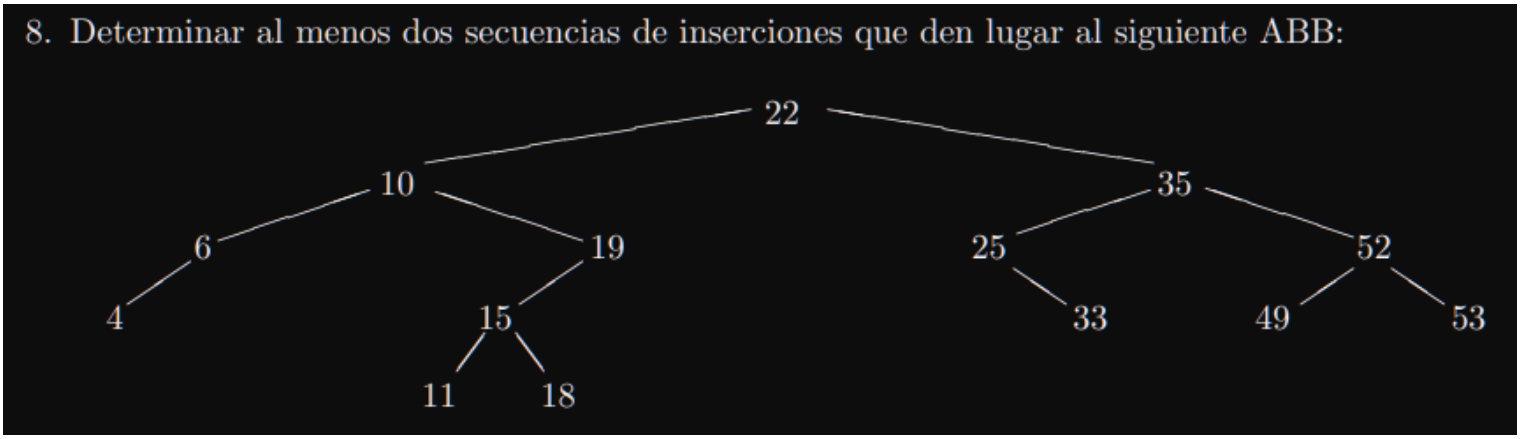
7) 🛠️

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.

23, 35, 49, 51, 41, 25, 50, 43, 55, 15, 47, 37

8)

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



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