AYED II
Lautaro Gastón Peralta

Práctico 2.3

1. Implementá el TAD Pila utilizando la siguiente representación:

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

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

CONSTRUCTORES

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

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

OPERACIONES

```
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
  e := head(s)
end fun

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

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)

CONSTRUCTORES

```
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 : pointer to (Node of T)
    alloc(p)
    p->elem:=e
    p->next:=s
    s:=p
end proc
```

OPERACIONES

```
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
   e:=s->elem
end fun

proc pop(in/out s: Stack of T)
   var p : pointer to (Node of T)
   p:=l
   l:=l->next
   free(p)
end proc
```

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*.

a)

CONSTRUCTORES

```
fun empty_queue() ret q: Queue of T
   q.elems:=array[0..N-1] of T
   q.size := 0
end fun
```

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

OPERACIONES

```
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[0]
end fun

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

b)

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

CONSTRUCTORES

```
fun empty_queue() ret q: Queue of T
  q.elems:=array[0..N-1] of T
  q.fst:=0
  q.size := 0
end fun

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

OPERACIONES

```
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[fst]
end fun

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

```
type Direction = enumerate
                    left
                    right
                 end enumerate
type Path = List of Direction

{-CONSTRUCTORES-}

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->left:= tl
   t->value:=e
   t->rigth:= tr
end fun

{-DESTRUCTOR-}
proc destroy_tree (in/out t:Tree of T)
   if not is_empty_tree(t) then
      destroy_tree(t->left)
      destroy_tree (t->right)
      free(t)
   fi
end proc
```

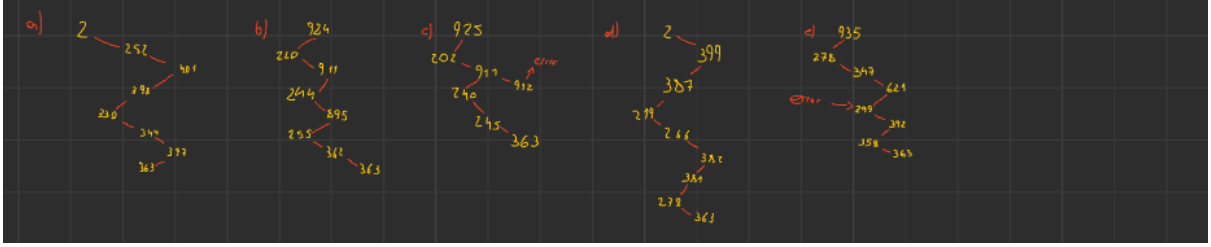
{-OPERADORES-}

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

fun root (t:Tree of T) ret b: bool
   e:= t->value
end fun

fun left (t:Tree of T) ret tl : Tree of T
   tl := t->left
end fun

fun right (t : Tree of T) ret tr : Tree of T
   tr := t->right
end fun

fun height (t : Tree of T) ret n : nat
   n:=0
    if not is_empty_tree(t) then
     n:= 1+(height(t->left) max height(t->right))
   fi
end fun

fun is_path (t : Tree of T, p : Path) ret t0 : Tree of T
    if (is_empty_tree(t)) then
       b := false // Si el árbol es vacío, no hay camino válido.
    else if (is_empty_list(p)) then
       b := true // Si la ruta está vacía, hemos llegado al final del camino, así que es un
                 camino válido.
    else if (head(p) = Left) then
          if (t->left != null) then
                b := is_path(t->left, tail(p)) // Continuamos la búsqueda en el subárbol
                                                  izquierdo.
          else
                b := false // Si no hay subárbol izquierdo, la ruta no es válida.
          fi
    else if (head(p) = Right) then
          if (t->right != null) then
                b := is_path(t->right, tail(p)) // Continuamos la búsqueda en el subárbol
                                                  derecho.
          else
                b := false // Si no hay subárbol derecho, la ruta no es válida.
          fi
     fi

fun subtree_at (t : Tree of T, p : Path) ret t0 : Tree of T
```

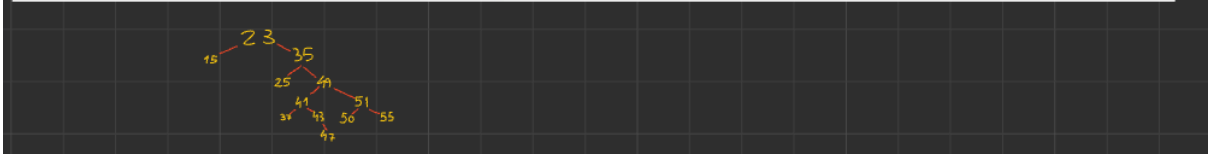
```
   if (is_empty_tree(t) or is_empty_list(p)) then
         t0 := null // Si el árbol es vacío o la ruta está vacía, no hay subárbol
   else if (head(p) = Left) then
         t0 := subtree_at(t->left, tail(p))
   else if (head(p) = Right) then
         t0 := subtree_at(t->right, tail(p)) fi
end fun

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

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?

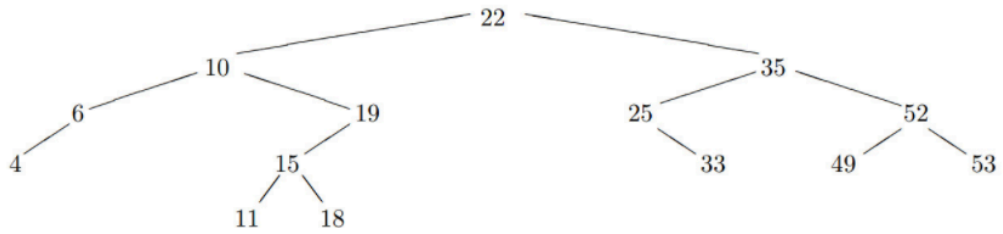
   (a) 2, 252, 401, 398, 330, 344, 397, 363. Si
   (b) 924, 220, 911, 244, 898, 258, 362, 363. NO
   (c) 925, 202, 911, 240, 912, 245, 363. No
   (d) 2, 399, 387, 219, 266, 382, 381, 278, 363. NO
   (e) 935, 278, 347, 621, 299, 392, 358, 363. No



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.



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.

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

FALTA HACER:

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.