ALGORITMOS Y ESTRUCTURAS DE DATOS II 2021

TIPOS ABSTRACTOS DE DATOS (TADs)

PRÁCTICO 2 - PARTE 3: PILAS, COLAS Y ÁRBOLES

Especificación del TAD Pila

```
spec Stack of T where

constructors
     fun empty_stack() ret s : Stack of T
     {-crea una pila vacía.-}
     proc push (in e : T,in/outs : Stack of T)
     {-agrega el elementoe al tope de la pilas. -}

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 des. -}
     {-PRE: not is_empty_stack(s) -}

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

     fun copy_stack (s1 : Stack of T) ret s2 : Stack of T
     {- copia el contenido de la pila s1 en la nueva pila s2 -}

destroy
     proc destroy_stack (in/out s: Stack of T)
     {- elimina la memoria usada por la pila s en caso de ser necesario -}

end specification
```

1) Implementá el TAD Pila utilizando la siguiente representación:

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

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

OBSERVACIÓN: en mi implementación de pilas como listas, yo voy a hacer la siguiente analogía: que el elemento que se encuentra al tope de la pila sea el primero de la lista (pensando la lista como si fuera un arreglo y viéndolo de izquierda a derecha)

```
implement Stack of T where

type Stack of T = List of T

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

     proc push (in e : T, in/out s : Stack of T)
          addl(e, s)
     end proc
     {- Recordar que cuando trabajamos con pilas, el último elemento que se
     agrega, pasa a ser el tope de la pila (pensar en una pila de platos) -}

operations
     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


     fun copy_stack (s1 : Stack of T) ret s2 : Stack of T
          s2 = copy_list(s1)
     end fun


destroy
     proc destroy_stack (in/out s : Stack of T)
          destroy(s)
     end proc

end implementation
```

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)
```

```
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)


constructors
     fun empy_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) {- p es el nodo que va a contener el nuevo
          elemento e -}
          alloc(p)
          p -> elem := e
          p -> next := s
          s := p {- s pasa a apuntar al nuevo primer elemento de la cola -}
     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
          e := s -> elem
     end fun
```

```
    proc pop (in/out s : Stack of T)
        var p: pointer to (Node of T)
        p := s {- hago esta asignación porque necesito tener referencia del elemento que quiero
        eliminar -}
        s := s -> next {- ahora, el primer elemento de la pila s es el segundo -}
        free(p)
    end proc

    proc copy_stack (s1 : Stack of T) ret s2 : Stack of T (REVISAR)
        var s1_aux : pointer to (Node of T)
        var s2_aux : pointer to (Node of T)
        var new_node : Node of T
        if is_empty_stack(s1) then
            s2 := empty_stack()
        else
            s1_aux := s1 {- stack auxiliar para recorrer s1 -}

            {- copio el primer elemento por separado -}
            s2->elem := s1->elem
            s2->next := null

            s2_aux := s2

            while s1_aux != null do
                alloc(new_node)
                new_node->elem := s1_aux->elem
                new_node->next := null
                s1_aux := s1_aux->next
                s2_aux := s2_aux->next
        fi
    end proc

destroy
    proc destroy(in/out s : Stack of T)
        var p : pointer to (Node of T)
        while not is_empty_stack(p) do
            p := s
            s := s -> next
            free(p)
        od
    end proc

end implementation
```

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. -}
     {- PRE: q.size < N-1, tiene que haber lugar en el arreglo para agregar
     un nuevo elemento. -}

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) -}

     fun copy_queue (q1 : Queue of T) ret q2 : Queue of T
     {- copia el contenido de la cola q1 en la nueva cola q2 -}

destroy
     proc destroy_queue (in/out q: Queue of T)
     {- libera la memoria usada por la cola q en caso de ser necesario -}

end specification
```

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
```

OBSERVACIÓN: en mi implementación, el primer elemento de la cola coincide con el primer elemento del arreglo.

```
implement Queue of T where

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

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

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

operations
     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)
```
{- IDEA: todos los elementos de la cola se tienen que mover un lugar hacia adelante. Traducido a arreglos, hay que pisar el primer elemento del arreglo con el segundo, el tercero pasa al segundo lugar y así sucesivamente. -}

```
    for i:= 0 to q.size-2 do
        q.elems[i] := q.elems[i+1]
    od
    q.size := q.size - 1
end proc
```

Ejemplo de prueba:
Tengo inicialmente el arreglo [**1**,**5**,**3**,8,7] y quiero hacer que se “vaya” el 1. Es decir, tengo q.size=3.

1) Primera iteración: q.elems[0] := q.elems[1]. Resultado: [**5**,**5**,**3**,8,7]
2) Segunda y última iteración: q.elems[1] := q.elems[2]. Resultado: [**5**,**3**,**3**,8,7]
3) q.size := q. size - 1. Resultado: [**5**,**3**,3,8,7]

Veamos que efectivamente se va el elemento 1, que era el primero de la cola, pero también es importante observar que al irse el primer elemento de la cola, el tamaño de la cola se reduce en uno.

```
fun copy_queue (q1 : Queue of T) ret q2 : Queue of T
    if is_empty_queue(q1) then
        q2 := empty_queue()
    else
        for i := 0 to N-1 do
            q2.elems[i] := q1.elems[i]
        od
    fi
    q2.size := q1.size
end fun

destroy
    proc destroy_queue (in/out q: Queue of T)
    skip
    end proc
```

3) b) Implementá el TAD Cola utilizando un arreglo como en el inciso anterior, pero asegurando que todas las operaciones estén implementadas en orden constante.
Ayuda 1: 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.

ACLARACIÓN: que una operación sea de orden CONSTANTE quiere decir que es SIEMPRE del mismo orden N SIN IMPORTAR el tamaño del dato de entrada.

Ahora bien, veamos que en la implementación anterior la función que es de orden lineal, ya que depende del tamaño del dato de entrada, es dequeue(). ¿Por qué es esto? Pues porque es fácil de ver que para eliminar el primer elemento de la cola, tengo que recorrer TODO el arreglo, moviendo una posición a la izquierda cada elemento.

Veamos qué se puede hacer para que dequeue sea de orden constante:

En la implementación anterior, es claro, como dije anteriormente, que enqueue() es de orden constante y dequeue() es de orden lineal. Veámoslo con un ejemplo:

1. Cola vacía: [54, 67, -89, 13, 127, 15] size = 0
2. Encolo el 12: [**12**, 67, -89, 13, 127, 15] size = 1
3. Encolo el 81: [**12**, **81**, -89, 13, 127, 15] size = 2
4. Encolo el 9: [**12**, **81**, **9**, 13, 127, 15] size = 3
5. Desencolo: [**81**, **9**, 9, 13, 127, 15] size = 2 (tuve que correr todos los elementos de la cola un lugar a la izquierda)
6. Enocolo el -11: [**81**, **9**, **-11**, 13, 127, 15] size = 3

   Acá quiero desencolar el 81, pero no tener que mover el resto de los elementos. Algo así:

7. Desencolo: elems = [??, **9**, **-11**, 13, 127, 15] size = 2
8. Desencolo de nuevo: [??, ??, **-11**, 13, 127, 15] size = 1

Lo que se podría hacer, usando la ayuda 1, es agregar un campo más a la tupla que contenga el índice en donde comienza la cola dentro del arreglo, es decir el índice del primer elemento de la cola. De esta forma, me quedaría:

```
type Queue of T = tuple
                    elems : array [0 .. N-1] of T
                    size : nat
                    start : nat {- posición dentro del arreglo en donde
                    comienza la cola -}
                  end tuple
```

Ahora bien, hay un problema con esto. ¿Qué pasa si sigo encolando? Sigo con el arreglo con el que estaba trabajando antes:

9. Encolo el 55: [??, ??, **-11**, **55**, 127, 15] size = 2, start = 2

10. Encolo el -123: [??, ??, **-11**, **55**, **-123**, 15] size = 3, start = 2
11. Encolo el 789: [??, ??, **-11**, **55**, **-123**, **789**] size = 4, start = 2

Ahora, ¿puedo seguir encolando? Sí, pues todavía me quedan dos lugares dentro del arreglo! Por lo tanto, a partir de este punto, puedo seguir encolando usando las primeras posiciones del arreglo. Sería como suponer que después del último elemento del arreglo sigue el 1er elemento del arreglo: **Arreglos circulares**. Con esto, puedo seguir encolando así:

12. Encolo el 28: [**28**, ??, **-11**, **55**, **-123**, **789**] size = 5, start = 2
13. Encolo el 3: [**28**, **3**, **-11**, **55**, **-123**, **789**] size = 6, start = 2

Usando la ayuda 2, analizo en dónde, es decir en qué posición fue que encolé los elementos anteriores:

“Algunas veces se le llama, sugerentemente, *aritmética del reloj*, ya que los números «dan la vuelta» tras alcanzar cierto valor llamado *módulo*.”

En este caso, mi módulo es la longitud del arreglo!!! Pues veamos que una vez que size + start llegan a N, que es la longitud del arreglo, ahí es cuando se empieza a “dar la vuelta” y a agregar los elementos por el inicio del arreglo. Así, teniendo en cuenta esto y analizando los pasos 12 y 13, puedo concluir que en ambos pasos, encolo en la posición (q.size + q.start) % N.

Y retrocediendo un poco, veo que en todos los pasos encolé en la posición (q.size + q.start) % N

Luego:

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

Ahora, haciendo más pasos, veamos qué pasa cuando desencolo:

14. Desencolo: [**28**, **3**, ?? , **55**, **-123**, **789**] size = 5, start = 3
15. Desencolo: [**28**, **3**, ?? , ?? , **-123**, **789**] size = 4, start = 4
16. Desencolo: [**28**, **3**, ?? , ?? , ?? , **789**] size = 3, start = 5
17. Desencolo: [**28**, **3**, ?? , ?? , ?? , ??] size = 2, start = 0
18. Desencolo: [??, **3**, ?? , ?? , ?? , ??] size = 1, start = 1
19. (...)

Es claro que siempre que desencole, el tamaño de la cola se va a reducir en uno:

```
     q.size := q.size - 1
```

Pero ahora, ¿qué pasa con el q.start? Nuevamente, teniendo en cuenta la Ayuda 2, se puede ver que cada vez que desencolo, la variable q.start cambia de la siguiente forma:

```
     q.start := (q.start + 1) % N
```

Finalmente, la implementación del TAD Cola de forma tal que todas las operaciones sean de orden constante es la siguiente:

```
implement Queue of T where

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

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

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

operations
     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)
          var start_pos: nat
          start_pos := (q.start + 1) % N
          q.start := start_pos
          q.size := q.size - 1
     end proc

destroy
     proc destroy_queue (in/out q: Queue of T)
          skip
     end proc

end implementation
```

Especificación del TAD Árbol

```
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 el elemento e y subárboles tl y tr. -}

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 : TreeofT)rettl : Tree of T
     {-Devuelve el subárbol derecho det. -}
     {-PRE:not is_empty_tree(t) -}

     fun height(t : Tree of T) ret n : Nat
     {- Devuelve la distancia 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)rett0 : 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) -}

destroy
     proc destroy_tree (in/out t : Tree of T)
     {- libera la memoria usada por el árbol t en caso de ser necesario -}
```

```
end specification
```

Además, para los caminos en un árbol binario, se usa:

```
type Direction = enumerate
                    Left
                    Right
                 end enumerate

type Path = List of Direction
```

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)
```

```
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 → left := tl
        t → value := e
        t → right := tr
    end fun

operations
    fun is_empty_tree(t : Tree of T) ret b : Bool
```

```
     b := t = null
end fun

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

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

fun right(t : TreeofT)rettl : Tree of T
     tr := t → right
end fun

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

fun is_path(t : Tree of T, p : Path) ret b : Bool
     var p_aux : Path
     var t_aux : Tree of T
     if is_empty_tree(t) then
          b := false
     else
          p_aux := copy_list(p)
          t_aux := t
          b := true
          while not is_empty_list(p_aux) ∧ b do
               if head(p_aux) = Left then
                    b := is_empty_tree(t_aux→left)
                    t_aux = t_aux→left
               else if head(p_aux) = Right then
                    b := is_empty_tree(t_aux→right)
                    t_aux = t_aux→right
               fi
               tail(p_aux)
          od
     fi
     destroy(p_aux)
end fun
```

VERSIÓN RECURSIVA DE subtree_at

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

```
        if is_empty_tree(t) or is_empty(p) then
            t0 := t
        else
            {- sabemos que el árbol es no vacío y el path es no vacío -}
            p_aux := list_copy(p)
            tail(p_aux)
            if head(p) = Left then
                t0 := subtree_at(t->left, p_aux)
            else
                t0 := subtree_at(t->right, p_aux)
            fi
           destroy(p_aux)
        fi
    end fun

    VERSIÓN ITERATIVA DE subtree_at
    fun subtree_at(t : Tree of T, p : Path) ret t0 : Tree of T
        var p_aux : Path
        p_aux := copy_list(p)
        t0 := t
        while not is_emtpy_list(p_aux) ∧ not is_empty_tree(t0) do
            if head(p_aux) = Left then
                t0 := t→left
            else if head(p_aux) = Right then
                t0 := t→right
            fi
            tail(p_aux)
        od
        destroy(p_aux)
    end fun


    fun elem_at(t : Tree of T, p : Path) ret e : T
        var subtree : Tree of T
        subtree = subtree_at(t,p)
        e = subtree→value
    end fun

destroy
    proc destroy_tree (in/out t : Tree of T)
        if t != null then
            t->left = abb_destroy(t->left)
            t->right = abb_destroy(t->right)
            free(t)
            t = null
        fi
    end proc
```

5) Un Diccionario es una estructura de datos muy utilizada en programación. Consiste de una colecciones 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 por 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 operacin 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 operacion que devuelve un conjunto con todas las claves contenidas en el diccionario.

5)a) Especifica el TAD Diccionario indicando constructores y operaciones.

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