TC2006 Lenguajes de Programación

Tarea 4: Programación Intermedia en Racket

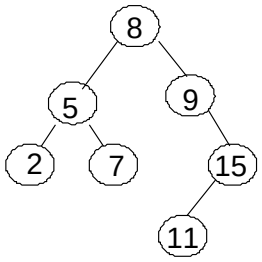
Fecha de entrega: martes 14 de abril de 2020

(* Equipos de 2 integrantes *)

En esta tarea pondrás en práctica tus conocimientos sobre programación recursiva con estructuras de datos y funciones de orden superior en Racket. Uno de los integrantes del equipo deberá subir las soluciones de los problemas a Canvas dentro del archivo de Racket **M_Tarea4.rkt**, donde deberán sustituir la M del nombre, por sus matrículas separadas con guiones bajos (_). El código fuente deberá estar documentado internamente mediante comentarios que incluyan, al menos, las matrículas y nombres de los integrantes del equipo, así como la descripción de lo que calcula cada función y el significado de cada parámetro formal de las mismas ***(10 puntos).***

Problemas sobre Estructuras de Datos (60 puntos)

1. Un Árbol Binario (AB) puede ser representado en Racket, por medio de una lista en el siguiente formato: (raíz subárbol-izquierdo subárbol-derecho). Por ejemplo, si se tiene definido el siguiente AB: Su representación en Racket sería:




```
(define AB
    '(8  (5  (2  ()  ())
             (7  ()  ()))
         (9  ()
             (15 (11  ()  ())
                 () ))))
```

Utilizando **recursión explícita** y SIN utilizar primitivos de orden superior (map, apply, …) programar las siguientes funciones:

a. Implementar el predicado **arbol-binario?** que dada una lista determine si es consistente con el formato de listas establecido para representar árboles binarios. Probar con:

```
> (arbol-binario? '()) => #t
> (arbol-binario? AB)  => #t
> (arbol-binario? '(a (b (() ()) ()) (d () ()))  => #f
```

b. Implementar la función **obten-menores** que dados un **árbol binario** y un valor como argumentos, cree una lista con los valores de los nodos que contengan valores menores que el valor dado como argumento. Los valores en la lista resultante pueden, o no, estar ordenados.
Probar con:

```
> (obten-menores AB 2)   => ()
> (obten-menores AB 8)   => (5 3 7)
> (obten-menores AB 20)  => (8 5 2 7 9 15 11)
```

c. Implementar la función **rama+larga** que dado un **árbol binario** regrese una lista con los valores que se encuentran en su rama más larga. En caso de empate, que regrese los valores de su primer rama más larga.
Probar con:

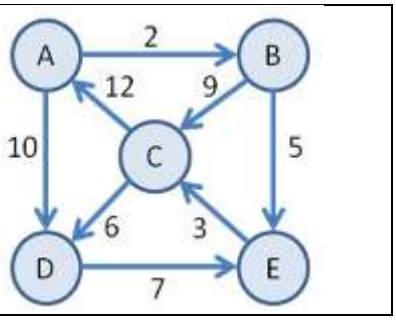
```
> (rama+larga '()) => ()
> (rama+larga AB)  => (8 9 15 11)
> (rama+larga '(a (b (c () ()) ()) (d (e () ()) ())))  => (a b c)
```

2. Un grafo dirigido y ponderado puede ser representado en Racket por medio de una lista de arcos. La lista de representación del grafo en este caso, tiene un registro por cada nodo del grafo, y este registro a su vez es una lista con el siguiente formato:

```
(nodo_x ( (nodo_adyacente_1 peso_arco_1)
          (nodo_adyacente_2 peso_arco_2)
           ...
          (nodo_adyacente_n peso_arco_n)))
```

Donde $nodo_x$ es el identificador de un nodo origen en el grafo, $nodo\_adyacente_i$ es el identificador de un nodo destino y $peso\_arco_i$ es la ponderación del arco que une al $nodo_x$ con el $nodo\_adyacente_i$.
Bajo este formato, el siguiente grafo tendría la representación en Racket que se muestra:

```
(define g
'((A ((B 2) (D 10)))
  (B ((C 9) (E 5)))
  (C ((A 12) (D 6)))
  (D ((E 7)))
  (E ((C 3)))
 ))
```



a) Implementar la función **nodos-destino** que liste los nodos destino que tienen a N como nodo origen directo.
Probar con:

```
> (nodos-destino g 'A) => (B D)
> (nodos-destino g 'D) => (E)
> (nodos-destino g 'F) => ()
```

b) Implementar la función **nodos-origen** que cree una lista con los nodos en un grafo a partir de los cuales se puede llegar directamente al nodo N.
Probar con:

```
> (nodos-origen g 'A) => (C)
> (nodos-origen g 'C) => (B E)
> (nodos-origen g 'F) => ()
```

c) Implementar la función **elimina-arco** que recibe un grafo y dos identificadores de nodos como entrada y regresa el grafo sin el arco especificado (si existe).
Probar con:

```
> (elimina-arco g 'B 'C)
=> ((A ((B 2) (D 10))) (B ((E 5)))
    (C ((A 12) (D 6))) (D ((E 7)))(E ((C 3)))))
> (elimina-arco g 'B 'D)
=> ((A ((B 2) (D 10))) (B ((C 9) (E 5)))
    (C ((A 12) (D 6))) (D ((E 7)))(E ((C 3)))))
> (elimina-arco g 'F 'C)
=> ((A ((B 2) (D 10))) (B ((C 9) (E 5)))
    (C ((A 12) (D 6))) (D ((E 7)))(E ((C 3)))))
```

Problemas sobre Funciones de Orden Superior *(30 puntos)*

3. SIN utilizar recursividad explícita implementar las funciones que se describen a continuación utilizando adecuadamente (si se requieren) los primitivos de orden superior ***map***, ***apply*** y la forma especial ***lambda***. Todas las funciones trabajarán sobre matrices representadas en forma de listas de sublistas, donde cada sublista corresponde a un renglón de la matriz, por ejemplo, la lista ((1 2 3)(4 5 6)) representa la matriz.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

a. Implementar la función **cuenta-ceros** que cuente los ceros que se encuentren en una matriz de cualquier tamaño.
Probar con:

```
> (cuenta-ceros '()) => 0
> (cuenta-ceros '((0 1)(2 3))) => 1
> (cuenta-ceros '((4 0 3 1)(5 1 2 1)(6 0 1 1))) => 2
```

b. Implementar la función **minmax** que regrese una lista con el menor y el mayor valor encontrado en una matriz de cualquier tamaño. Asumir que la matriz al menos tiene un elemento.
Probar con:

```
> (minmax '((2))) => (2 2)
> (minmax '((0 1)(2 3))) => (0 3)
> (minmax '((4 0 -3 1)(5 -1 2 1)(6 0 1 1))) => (-3 6)
```

c. Implementar la función **multmat** que multiplique dos matrices. Asumir que las matrices si se pueden multiplicar, o sea son cuadradas o rectangulares donde el número de columnas de la primera es igual al número de renglones de la segunda.
Probar con:

```
> (multmat '((1 2 3)(0 2 1)) '((4 0 3 1)(5 1 2 1)(6 0 1 1)))
      => ((32 2 10 6)(16 2 5 3))
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 2 & 1 \end{pmatrix} \times \begin{pmatrix} 4 & 0 & 3 & 1 \\ 5 & 1 & 2 & 1 \\ 6 & 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 32 & 2 & 10 & 6 \\ 16 & 2 & 5 & 3 \end{pmatrix}$$