UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

ESTRUCTURAS DISCRETAS 2020-1

Profesor: M. en C. I. C. Odín Miguel Escorza Soria
Ayudante de laboratorio: Salazar González Edwin Max

Práctica Seis.

DEFINICIONES RECURSIVAS

1. Objetivos de la práctica

Reforzar conceptos de definiciones recursivas.

- Reforzar construcciones recursivas en estructuras como listas y árboles binarios. No se incluyen expresiones lógicas pues se verán en prácticas más adelante.

2. Desarrollo de la práctica

Una definición recursiva es aquella en la que los objetos se definen en términos de ellos mismos. En programación cuando un programa se llama a si mismo entonces se dice que esta definido recursivamente [1, 4].

Las definiciones recursivas toman gran relevancia dentro de las matemáticas y en particular en las Ciencias de la Computación pues se pueden demostrar resultados a partir de la inducción estructural. Esta forma de demostración se verá en la práctica 7.

Las definiciones recursivas tienen dos características importantes [1]:

- Tienen un conjunto de casos base en cuya definición no aparece el objeto mismo.
- Tienen un conjunto de reglas recursivas en cuya definición aparece un nuevo objeto con términos definidos previamente.

Siempre que se usen definiciones recursivas es necesario tener en cuenta los patrones básicos del tipo que se está definiendo. A la acción de empatar un patrón en la definición recursiva se le comoce como *caza de patrones*[1].

Los patrones básicos del tipo definido en la práctica pasada (números naturales) son:

- 0
- $S(n)$

Entonces para definir una función f sobre números naturales basta definir $f(0)$ y $f(S(n))$.

Un ejemplo es la función factorial de un número natural, cuya definición recursiva es [2]:

- $fac(0) = 1$
- $fac(S(n)) = S(n) * fac(n)$

[1] Del inglés *pattern matching*

2.1. Listas

Una lista es un conjunto de elementos de cierto tipo cuya cardinalidad es variada y cada elemento tiene un único predecesor y un único sucesor a excepción del primer y último elemento, en el caso de Haskell las listas son heterogéneas pero pueden ser homogéneas en otros lenguajes [3].

Por definición, una lista es [1]:

representa la lista vacía.

- (x:xs)representa una lista donde x es el primer elemento de la lista y xs el resto de la lista.
- Estos y solo éstos.

2.2. Árboles

Un árbol es una estructura de datos usada para representar relación jerárquica entre datos [3]

La definición recursiva de un árbol es [1]:

- *nil* es el árbol vacío.
- Arbol(e, [A_1, ... ,A_n]) es el árbol cuya raíz tiene a e como elemento y A_1, ... ,A_n como subárboles o nodos hijos.
- Nada más es un árbol.

Para el caso particular de un árbol binario la definición recursiva es [3]:

- *nil* es el árbol binario vacío.
- Arbol(e, A_i, A_d) es el árbol binario cuya raíz tiene a e como elemento y A_i y A_d como subárboles binarios izquierdo y derecho respectivamente.
- Nada más es un árbol binario.

Estas esctructuras cumplen propiedades importantes:

- Existe cierta jerarquía en los nodos. Jerarquía en el sentido que el árbol empieza con un nodo raíz y a partir de este nodo se desprenden los demás nodos, de igual forma, de manera jerárquica.
- Ningún nodo puede tener como nodo hijo al nodo raíz.
- No existen ciclos.

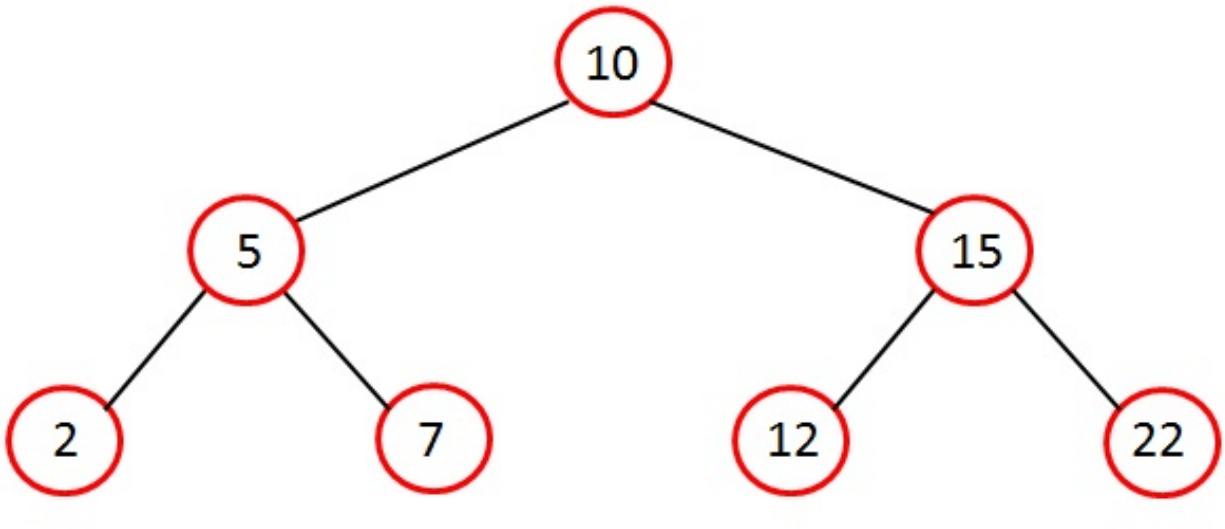

Figura 1: Ejemplo de un árbol binario.

2.3. Expresiones lógicas

En la práctica 1 se especificó que una expresión lógica es aquella que toma un valor de verdadero o falso. Su definición recursiva es la siguiente:

- var: una variable.
- $true$: el valor de verdadero.
- $false$: el valor de falso.
- $\neg A$: la negación de una expresión lógica. A corresponde a una expresión.
- $A \wedge B$: La conjunción de dos expresiones. A y B son cualesquiera dos expresiones.
- $A \vee B$: la disyunción de dos expresiones. A y B son cualesquiera dos expresiones.
- $A \rightarrow B$: la implicación de dos expresiones. A y B son cualesquiera dos expresiones.
- $A \leftrightarrow B$: la doble implicación de dos expresiones. A y B son culesquiera dos expresiones.

3. INSTRUCCIONES

Descarga el archivo Practica6.hs y resuelve los ejercicios definidos sobre éste.

4. EJERCICIOS

- Ejercicio 1: resolver la función *belongs* que recibe un elemento e y una lista L. Regresa $True$ si e esta en la lista L, $False$ en otro caso

```
> belongs 3 [0,1,2]
False
> belongs 3 [0,1,2,3,4]
True
```

Código 1: Ejemplo de llamada función *belongs*.

- Ejercicio 2: resolver la función *get_nth* que recibe un número n y una lista L. Regresa el n-ésimo elemento de la lista L.

```
> get_nth 0 ['S','k', 'i', 'n', 's']
'S'
> get_nth 12 ['T', 'a', 'e', 'k', 'w', 'o', 'n', 'd', 'o']
*** Exception: Indice mas grande que tam de la lista.
> get_nth 3 [0,2,4,7,12,22]
7
```

Código 2: Ejemplo de llamada función *get_nth*.

- Ejercicio 3: resolver la función *delete_nth* que recibe un número n y una lista L. Regresa la lista L después de eliminar el n-ésimo elemento.

```
> delete_nth 0 [48,88,1]
[88,1]
> delete_nth 50 ['C', 'h', 'r', 'o', 'n', 'o', 's']
['C', 'h', 'r', 'o', 'n', 'o', 's']
```

Código 3: Ejemplo de llamada función *delete_nth*.

- Ejercicio 4: resolver la función *delete_all* que recibe un elemento e y una lista L. Elimina todas las apariciones de e en la lista L.

```
> delete_all 3 [0,1,2,3,3,3,3,3,3,3,3,3,4,5,6]
[0,1,2,4,5,6]
```

Código 4: Ejemplo de llamada función *delete_nth*.

- Ejercicio 5: resolver la función *filter_lab* que recibe un predicado p y una lista L. Regresa la lista con todos los elementos que cumplan con el predicado p.

```
> filter_lab (> 5) [10,2,7,4,6,8,]
[10,7,6,8]
```

Código 5: Ejemplo de llamada función *filter_lab*.

Para los siguientes ejercicios el árbol construido será el presentado en la figura 1.

- Ejercicio 6: resolver la función *num_elem* que recibe un árbol binario T y regresa el número de elementos de T

```
> num_elem (Node 10 (Node 5 (Node 2 Empty Empty) (Node 7 Empty↩
    Empty))   (Node 15 (Node 12 Empty Empty) (Node 22 Empty ↩
   Empty)))
7
```

Código 6: Ejemplo de llamada función *num_elem*.

- Ejercicio 7: resolver la función *num_leaves* que recibe un árbol binario T y regresa el número de hojas de T

```
> num_leaves (Node 10 (Node 5 (Node 2 Empty Empty) (Node 7 ↩
   Empty Empty))   (Node 15 (Node 12 Empty Empty) (Node 22 ↩
   Empty Empty)))
4
```

Código 7: Ejemplo de llamada función *num_leaves*.

- Ejercicio 8: resolver la función *belongsT* que recibe un elemento e y un árbol binario T. Regresa $True$ si e es elemento de T, $False$ en otro caso.

```
> belongs 7 (Node 10 (Node 5 (Node 2 Empty Empty) (Node 7 ↩
   Empty Empty))   (Node 15 (Node 12 Empty Empty) (Node 22 ↩
   Empty Empty)))
True
> belongs 88 (Node 10 (Node 5 (Node 2 Empty Empty) (Node 7 ↩
   Empty Empty))   (Node 15 (Node 12 Empty Empty) (Node 22 ↩
   Empty Empty)))
False
```

Código 8: Ejemplo de llamada función *belongsT*.

- Ejercicio 9: resolver la función *preorder* que recibe un un árbol binario T y regresa la lista de sus elementos recorridos en preorden.

```
> preorder (Node 10 (Node 5 (Node 2 Empty Empty) (Node 7 Empty↩
    Empty))   (Node 15 (Node 12 Empty Empty) (Node 22 Empty ↩
   Empty)))
[10,5,2,7,15,12,22]
```

Código 9: Ejemplo de llamada función *preorder*.

- Ejercicio 10: resolver la función *inorder* que recibe un un árbol binario T y regresa la lista de sus elementos recorridos en preorden.

```
> inorder (Node 10 (Node 5 (Node 2 Empty Empty) (Node 7 Empty ↩
   Empty))   (Node 15 (Node 12 Empty Empty) (Node 22 Empty ↩
   Empty)))
[2,5,7,10,12,15,22]
```

Código 10: Ejemplo de llamada función *preorder*.

- Ejercicio 11: resolver la función *postorder* que recibe un un árbol binario T y regresa la lista de sus elementos recorridos en preorden.

```
> postorder (Node 10 (Node 5 (Node 2 Empty Empty) (Node 7 ↩
   Empty Empty))   (Node 15 (Node 12 Empty Empty) (Node 22 ↩
   Empty Empty)))
[2,7,5,12,22,15,10]
```

Código 11: Ejemplo de llamada función *preorder*.

- Ejercicio 12: resolver la función *add* que recibe un un elemento e y un árbol binario T y añade a e al árbol T.

  Nota: no olviden que se esta trabajando con árboles binarios de búsqueda y pueden asumir que siempre recibirá números diferentes (no puede haber dos elementos iguales en el árbol).

```
> add  20 (Node 10 (Node 5 (Node 2 Empty Empty) (Node 7 Empty ↩
   Empty))   (Node 15 (Node 12 Empty Empty) (Node 22 Empty ↩
   Empty)))
Node 10 (Node 5 (Node 2 Empty Empty) (Node 7 Empty Empty))   (↩
   Node 15 (Node 12 Empty Empty) (Node 22 (Node 20 Empty Empty↩
   ) Empty))
```

Código 12: Ejemplo de llamada función *preorder*.

- Ejercicio 13: resolver la función *delete* que recibe un un elemento e y un árbol binario T y elimina a e del árbol T.

  Nota: no olviden que se esta trabajando con árboles binarios de búsqueda y pueden asumir que siempre recibirá números diferentes (no puede haber dos elementos iguales en el árbol).

```
> delete 10 (Node 10 (Node 5 (Node 2 Empty Empty) (Node 7 ↩
   Empty Empty))   (Node 15 (Node 12 Empty Empty) (Node 22 ↩
   Empty Empty)))
Node 7 (Node 5 (Node 2 Empty Empty) Empty) (Node 15 (Node 12 ↩
   Empty Empty) (Node 22 Empty Empty))
```

Código 13: Ejemplo1 de llamada función *delete*.

```
> delete 10 (Node 10 (Node 5 (Node 2 Empty Empty) (Node 7 ↩
   Empty Empty))   (Node 15 (Node 12 Empty Empty) (Node 22 ↩
   Empty Empty)))
```

```
Node 5 (Node 2 Empty Empty) (Node 15 (Node 12 (Node 7 Empty ↩
   Empty) Empty) (Node 22 Empty Empty))
```

Código 14: Ejemplo2 de llamada función *delete*.

```
> delete 10 (Node 10 (Node 5 (Node 2 Empty Empty) (Node 7 ↩
   Empty Empty))   (Node 15 (Node 12 Empty Empty) (Node 22 ↩
   Empty Empty)))
Node 12 (Node 5 (Node 2 Empty Empty) (Node 7 Empty Empty)) (↩
   Node 15 Empty (Node 22 Empty Empty))
```

Código 15: Ejemplo3 de llamada función *delete*.

Cualquiera de las 3 formas de borrar es correcta, solo hay que ser consistentes y recordar que al ser árboles binarios de búsqueda el orden importa.

5. ESPECIFICACIONES

✓ Respetar las firmas de las funciones. Cambiarlas podría ser motivo de anulación del ejercicio.

× Cualquier plagio de prácticas será evaluado con 0, sin hacer indagaciones. **Creen** su propio código.

× Cualquier práctica entregada posterior a la fecha límite NO será tomada en cuenta.

Se deberá contar con un difectorio cuyo nombre sea PracticaSeis. Dentro del directorio se deben tener:

- README.txt, donde se incluya nombre y número de cuenta de los integrantes del equipo junto con comentarios que crean pertinentes sobre la práctica.
- PracticaSeis.hs, script requerido para esta práctica.

Compirmir el directorio con el formato **ApellidoNombreP6**. Comprimir con extensión .tar.gz o .zip

Solamente un integrante del equipo deberá enviar la práctica pero deberán enviar el correo con copia a su compañero de trabajo.

Enviar la práctica al correo ciclomax9@ciencias.unam.mx con el asunto `[LC-Apellido-Nombre-P6]`.

"When debugging, novices insert corrective code; experts remove defective code" - Richard Pattis

Suerte ☺

Referencias

[1] Miranda, Favio y Viso, Elisa, *Matemáticas Discretas*. México DF, México: Las prensas de ciencias, 2010.

[2] Rosen, Kenneth H., *Matemáticas Discretas*. Colombia: Concepción Fernández, 2004.

[3] Joyanes A., Luis, *Fundamentos de programación. Algoritmos, estructura de datos y objetos*. España: McGrawHill, 2008

[4] Pes, Carlos, *Diseño de Algoritmos en Pseudocódigo y Ordinogramas*. CA, USA: Tutorial de Algoritmos de Abrirllave, 2017.