Algoritmos y Estructuras de Datos II - 1° cuatrimestre 2016

# Práctico 2 - Parte 3

1. Dada la especificación de árbol binario vista en la clase teórica, extenderla con (algunas de) las siguientes operaciones:

   **nodos** que devuelve la cantidad de nodos de un árbol

   **es-hoja** que determina si un árbol es hoja, es decir, sus subárboles izquierdo y derecho son vacíos

   **altura** que devuelve la longitud del camino que va desde un árbol hasta su hoja más lejana

   **es-descendiente** que dados dos árboles determina si el primero es descendiente del segundo

   **es-ancestro** que dados dos árboles determina si el primero es ancestro del segundo

   **profundidad** que dados un árbol y un subárbol del mismo determina la longitud del camino que va desde el árbol hasta el subárbol

   **nivel** que dado un árbol y una profundidad (como un número natural) devuelve una lista de los subárboles de esa profundidad en el árbol

   **descendiente** (↓) que dados un árbol y una posición (como una secuencia de ceros y unos, posiblemente vacía) devuelve el subárbol en esa posición

   **elemento** (.) que dados un árbol y la posición de un nodo no vacío del mismo (como una secuencia de ceros y unos, posiblemente vacía) devuelve el elemento en esa posición

2. Se tiene la siguiente implementación de árbol binario:

   **type** node = **tuple**
   left: **pointer to** node
   value: elem
   right: **pointer to** node
   **end tuple**
   **type** bintree = **pointer to** node

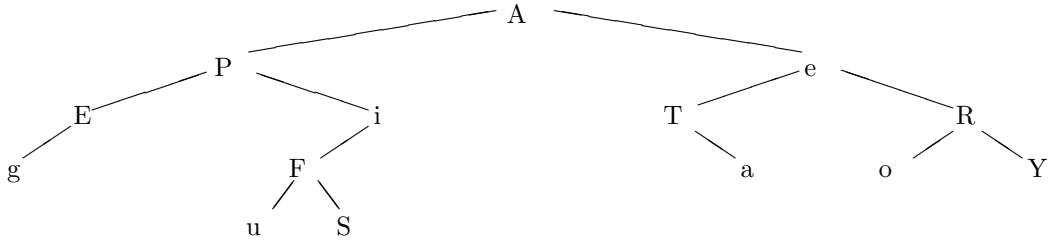
   (a) Implementar (algunas de) las operaciones especificadas en el ejercicio 1, considerando que se encuentran implementadas las demás operaciones (por ejemplo, de la forma vista en el teórico), y las funciones auxiliares necesarias para verificar las precondiciones requeridas.

   (b) Analizar la complejidad de cada una de las operaciones implementadas (incluyendo la verificación de sus respectivas precondiciones).

3. Las tres formas más usadas para procesar todos los elementos de un árbol, partiendo de su nodo raíz, son *pre-order*, que procesa primero el elemento del nodo y luego los elementos de los subárboles (primero el izquierdo y luego el derecho); *in-order*, que procesa primero los elementos del subárbol izquierdo, luego el elemento del nodo y por último los del subárbol derecho; y *post-order*, que procesa los elementos de los subárboles (primero el izquierdo y luego el derecho) y por último el elemento del nodo.

   (a) Escribir los procedimientos *pre-order*, *in-order* y *post-order* para ejecutar el procedimiento **proc** p(**in/out** e: elem) en todos los elementos del árbol t: bintree.

   (b) Si el procedimiento p simplemente muestra el valor de cada elemento procesado, ¿cuál será el resultado de aplicar cada uno de los procedimientos del ítem anterior al siguiente árbol?



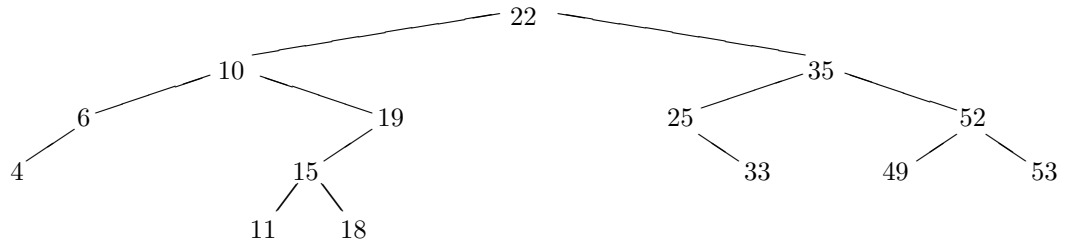
4. Extender la especificación e implementación de los ejercicios 1 y 2 con la operación **es-ABB** que determina si un árbol es un ABB (árbol binario de búsqueda).

5. 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) $935, 278, 347, 621, 299, 392, 358, 363$.

6. 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.

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




8. Escribir una implementación de la función *search* en un ABB que no utilice recursión.

9. Implementar el TAD *diccionario* utilizando como estructura de datos un ABB.

10. Extender la especificación e implementación de los ejercicios 1 y 2 con la operación **es-heap** que determina si un árbol es un *heap*.

11. Un *heap binario*, además de la propiedad de ordenación, debe cumplir la propiedad de *forma*: todos los niveles del árbol, excepto posiblemente el último, deben estar completos; y en caso de que el último nivel no lo esté, dicho nivel consistirá de una parte izquierda completa y una parte derecha vacía.

    Teniendo en cuenta estas dos propiedades características de un *heap*:

    (a) ¿Cuál es el número máximo y mínimo de nodos que puede tener un *heap* de altura $h$?
    (b) ¿Dónde está ubicado el elemento mínimo de un *heap*? ¿Y el máximo?
    (c) ¿Un arreglo ordenado de forma descendente implementa un *heap*? ¿Un *heap* implementado con un arreglo, da siempre lugar a un arreglo ordenado de manera descendente?
    (d) El arreglo $[23, 17, 14, 6, 13, 10, 1, 5, 7, 12]$ ¿es un *heap*?
    (e) Si un elemento $e$ ocurre dos veces en un heap, ¿debe una de esas ocurrencias ser hija de la otra?
    (f) Dada una *cola de prioridades* $Q$ representada con el *heap binario* $[15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]$. Graficar el *heap* paso a paso cuando:
        i. se ejecuta *dequeue(Q)*,
        ii. se ejecuta *enqueue(Q,10)*.

12. Cuando se implementa un *heap binario* con un arreglo $a$, el procedimiento sink(a) "hunde" el elemento a[1] de manera de que la estructura resultante mantenga la propiedad de ordenación del *heap*. Representar gráficamente la evolución, paso a paso, del *heap* al ejecutar sink(a), cuando $a = [3, 17, 27, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]$.

13. ¿Qué sucede si se ejecuta sink(a) cuando a[1] es más grande que sus hijos? ¿Qué pasa si queremos hundir el elemento a[i] cuando i > n/2, donde n es el tamaño de a?