CC3001 Algoritmos y Estructuras de Datos
Profesores: Nelson Baloian, Patricio Poblete, e Iván Sipirán
Auxiliares: Valentina Alarcón Yáñez, Samuel Chávez Fierro, Antonia G. Calvo, Cristián Llull, y Raimundo Lorca Correa

Auxiliar 6

16 de mayo de 2025

P1. Heap con método "modificar" (ejercicio 5.2)

Agregue a la clase Heap un método `modificar(k,x)` que al ser invocado, cambie la prioridad del elemento del casillero k, dándole como nuevo valor x y asegurando que el heap siga cumpliendo las restricciones de orden. Esta operación debe funcionar en tiempo O(log n) en el peor caso.

P2. Chequear ABB

Los **Á**rboles de **B**úsqueda **B**inaria son árboles binarios (es decir, que cada nodo tiene a lo más dos hijos) que cumplen la siguiente propiedad: dado un nodo padre, todos los nodos descendientes a la izquierda tienen valor menor que el padre, y todos los nodos descendientes a la derecha tienen valor mayor que el padre.

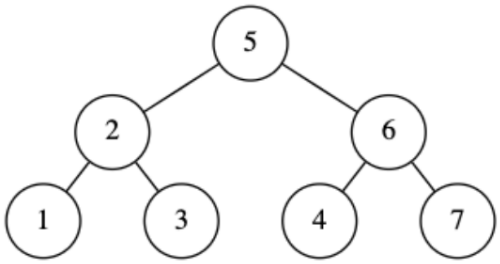


Figura 1: Árbol que NO ES de búsqueda binaria

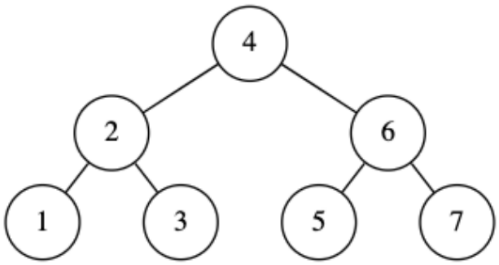


Figura 2: Árbol de búsqueda binaria

Se pide que programe diferentes métodos en la clase Árbol (que es una implementación de árbol binario), que le permita verificar si la estructura es un ABB o no. Los métodos que debe programar son los siguientes:

(a) Un método `check_order` que revisa que en el árbol, los nodos estén ordenados de menor a mayor, de izquierda a derecha. Esto se puede hacer realizando un recorrido en postorden o en inorden.

(b) Otro método `check_abb` que verifica que en el árbol, todos los nodos del subárbol izquierdo de un nodo sean menores que éste, y al mismo tiempo, todos los nodos del subárbol derecho son

mayores que él. Esto se puede realizar chequeando por intervalos. Le puede ser útil saber los límites del intervalo en cada subárbol. Para ello, utilice la clase Árbol definida en los apuntes.

P3. Transpose y Move to front

Tamara y Mariana quieren guardar más datos de lo recomendado para un diccionario tal como se vio en cátedra. Además, saben que necesitarán ciertos valores más seguido que otros.

Tamara plantea como solución que al buscar un elemento, este luego se mueva uno más cerca del inicio, para poder acceder más rápido la siguiente vez que se busque.

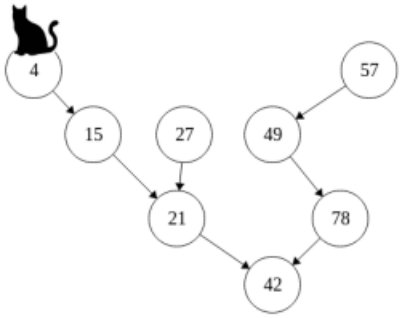
(a) Implemente la idea de Tamara en una función `transpose(x, a, n)`. Esta debe buscar x dentro del arreglo a que contiene n elementos. Si se encuentra el elemento, este se mueve una posición más adelante, lo que llamamos transposición. En caso contrario, se debe agregar x al final del arreglo y luego realizar la transposición.

Mariana ve la proposición de Tamara y decide formular una propia. En vez de mover el elemento buscado una posición más cerca del inicio, Mariana propone moverlo al frente del arreglo para acceder más rápido.

(b) Repita lo anterior pero ahora implementando la idea de Mariana en una función `move_to_front(x, a, n)`

P4. Gatito en un árbol (Propuesto)

Al gatito de Alan le gusta trepar al árbol binario de su jardín. El problema es que todavía no sabe cómo bajarse de ahí, así que siempre tienen que buscarlo. Debido a lo frondoso del árbol, no tiene más remedio que buscarlo rama por rama (nodo por nodo). Como Alan quiere subir lo menos posible al realizar la búsqueda por temor a caer, cada vez que trepa una rama él quiere buscar todos los nodos en ese nivel (esto es una Búsqueda a lo ancho, BFS). Tu objetivo es ayudar a Alan a salvar su gatito del árbol.



Los nodos del árbol se definen en la siguiente clase:

```
class NodoG :
  def __init__ (self, gatito, izq = None, der = None):
    self.val = str(val) + 'gatito ' if gatito else str(val)
    self.gatito = gatito # True si está el gatito, False si no
```

```
        self.izq = izq
        self.der = der
```

Para bajar al gatito del árbol, utilice la clase Cola vista en clase, y cree un programa que muestre en pantalla los valores de los nodos que ha recorrido Alan hasta encontrar a su gatito, de izquierda a derecha en cada nivel. Por ejemplo, en el árbol de la figura, la salida sería: 42 21 78 15 27 49 4.