

Pauta Auxiliar 7, Algoritmos y Estructuras de Datos - CC3001

Fecha: 4 de octubre de 2019

Pregunta 1: Checkear AVL

Un árbol AVL es un tipo especial de ABB auto-balanceable, en que todos sus nodos cumplen con la condición de que la diferencia de alturas entre su hijo izquierdo y derecho nunca es mayor que uno.

De manera formal, un árbol T , con hijos T_i y T_d es AVL si cumple con las siguientes condiciones:

- $|altura(T_i) - altura(T_d)| \leq 1$
- T_i es AVL
- T_d es AVL

Escriba una función `es_avl(raiz)` en Python, que dada la raíz de un árbol diga si éste cumple con las condiciones para ser AVL o no.

```
In [0]: def check_avl(raiz):  
    # Caso Base  
    if raiz == None:  
        return True, 0  
    # Checkeamos si es AVL a la izquierda y a la derecha y además revisamos  
    # la altura de cada subárbol para realizar la comparación  
    es_avl_izq, h_izq = check_avl(raiz.izq)  
    es_avl_der, h_der = check_avl(raiz.der)  
    # Recursión  
    return es_avl_der and es_avl_izq and abs(h_izq-h_der) <= 1, 1 + max(h_izq, h_der)
```

Pregunta 2

Un *heap* es un árbol binario que permite almacenar valores sin la necesidad de utilizar punteros (utilizando sólo un arreglo). En el caso del *min heap*, su principal característica es que el padre cuenta siempre con **menor** prioridad que sus hijos, esto es, el **mínimo** se encuentra en la **raíz**. Cuenta con todos sus niveles llenos a excepción de probablemente el último. Además, en el último nivel, los nodos se encuentran lo más a la izquierda posible. Un *heap* se utiliza para implementar una cola de prioridad.

Considere la siguiente secuencia de números: 22, 52, 20, 10, 15, 38, 60, 33, 40, 73, 45, 95

- Tome los nodos de a uno e insértelos en el *min heap*. Dibuje el estado del arreglo en cada inserción.
- Extraiga tres veces el mínimo y reordene el *heap* de tal forma que siga manteniendo forma y orden de *heap*
- Dibuje el árbol representado por el *heap*
- Implemente una cola de prioridad utilizando un *heap* en Python, que sea capaz de agregar un elemento y de retornar el mínimo.

```
In [0]: import numpy as np

class PriorityQueue:
    def __init__(self):
        self.nodes = np.zeros(10, dtype=int)
        self.size = 0

    def swim(self, a, j):
        while j >= 1 and a[j] < a[(j-1)//2]:
            a[j], a[(j-1)//2] = a[(j-1)//2], a[j]
            j = (j-1) // 2

    def sink(self, a, j, n):
        while 2*j + 1 < n:
            k = 2*j + 1
            if k + 1 < n and a[k+1] < a[k]:
                k += 1
            if a[j] <= a[k]:
                break
            a[j], a[k] = a[k], a[j]
            j = k

    def add(self, value):
        n = self.size
        if self.size == len(self.nodes):
            a = np.empty(2*n, dtype=int)
            for i in range(n):
                a[i] = self.nodes[i]
            self.nodes[n] = value
            self.swim(self.nodes, self.size)
            self.size += 1

    def get_minimum(self):
        if self.size == 0:
            raise Exception("Sorry, the Priority Queue is empty.")
        j = self.size
        a = np.copy(self.nodes)[0]
        self.nodes[0] = self.nodes[j - 1]
        self.nodes = self.nodes[:j-1]
        self.size -= 1
        self.sink(self.nodes, 0, self.size)
        return a
```

```
In [4]: # Creamos una data de prueba. Imprimir Los números que se insertan de menor a
        # mayor comprueba que la clase funciona correctamente.
        # Al final arrojará un error, dado que estamos extrayendo de una cola de prior
        # idad vacía. Esto también es correcto.
        cola = PriorityQueue()

        cola.add(8)
        cola.add(5)
        cola.add(7)
        cola.add(3)
        cola.add(9)

        print(cola.get_minimum())
        print(cola.get_minimum())
        print(cola.get_minimum())
        print(cola.get_minimum())
        print(cola.get_minimum())
        print(cola.get_minimum())

3
5
7
8
9
```

```
-----
Exception                                Traceback (most recent call last)
<ipython-input-4-6cb7fe214d53> in <module>()
      12 print(cola.get_minimum())
      13 print(cola.get_minimum())
----> 14 print(cola.get_minimum())

<ipython-input-3-b8260a7fe4dc> in get_minimum(self)
      33     def get_minimum(self):
      34         if self.size == 0:
----> 35             raise Exception("Sorry, the Priority Queue is empty.")
      36         j = self.size
      37         a = np.copy(self.nodes)[0]

Exception: Sorry, the Priority Queue is empty.
```

Pregunta 3: K-ésimo elemento de un ABB

Dado un puntero a la raíz de un árbol de búsqueda binaria, se quiere retornar su k-ésimo menor elemento. Por ejemplo, en el caso de la imagen, si k vale 3, se debe retornar 6 y si k es igual a 5, se debe retornar 9. Para realizar esto, cree una clase `NodoArbol` de manera que cada nodo sea capaz de guardar la cantidad de nodos con los que cuenta su subárbol izquierdo. Además programe la función `insertar(raiz, valor)` que inserte un elemento en el ABB y mantenga dicha cantidad actualizada. Utilice la cantidad de nodos a la izquierda para diseñar un algoritmo que permita encontrar el k-ésimo elemento en tiempo $O(h)$ en donde h es la altura del árbol. Ver árbol de ejemplo en el enunciado

In [0]:

```
In [0]: # Creamos el árbol, guardando el valor de nodos a la izquierda
class NodoArbol:
    def __init__(self, val, izq=None, der=None, nodosizq = 0):
        self.val = val
        self.izq = izq
        self.der = der
        self.nodosizq = nodosizq

# Función insertar
def insertar(raiz, val, nodosizq = 0):
    if(raiz == None):
        return NodoArbol(val)

    if(raiz.val > val):
        raiz.nodosizq+=1
        raiz.izq = insertar(raiz.izq, val)

    if(raiz.val < val):
        raiz.der = insertar(raiz.der, val)

    return raiz

# Encontrar k-ésimo viendo el valor de la raíz
def encontrar_kesimo(raiz, k):
    if(raiz == None):
        return None

    if(raiz.nodosizq == k-1):
        return raiz

    elif(raiz.nodosizq >= k):
        return encontrar_kesimo(raiz.izq, k)

    else:
        return encontrar_kesimo(raiz.der, k - raiz.nodosizq - 1)
```