Auxiliar 10 - Splay Trees y Árboles Digitales

Profesores: Iván Sipiran
Nelson Baloian
Patricio Poblete
Auxiliares: Alonso Almendras, Albani Olivieri
Vicente Olivares, Ricardo Valdivia
Sebastián Acuña, Martín Paredes

P1. Splay Trees

Un *splay tree* (o árbol biselado) es otro tipo de árbol de búsqueda binaria autobalanceante, cuya propiedad más interesante es que los últimos nodos buscados e insertados están más cerca de la raíz que los nodos más antiguos. En particular, el último nodo insertado/accedido x va a ser la nueva raíz del árbol. Esto se logra mediante una operación llamada *splaying* sobre x. Si una llave buscada x no se encuentra en el árbol, se realiza la operación *splay(x')*, con x' el último nodo visitado.

Esta estrategia garantiza que cualquier secuencia de m operaciones en un árbol que llega a tener n elementos, partiendo de un árbol vacío, toma tiempo $O(m \log n)$. Es importante notar que no garantiza que alguna secuencia en particular no va a tomar $O(n)$, sino que el costo acumulado dividido por el número de operaciones da un promedio de $O(\log n)$ por operación. Se dice que una estructura de este tipo es eficiente en el sentido *amortizado*.

Los tres casos de la operación de *splaying* se encuentran en el **anexo**.

a) Inserte en un *Splay Tree* vacío la siguiente secuencia: 1, 2, 3, 4, 5, 6, 7, 8.

b) Acceda al nodo con valor 1, realizando el *splaying* correspondiente. Comente acerca de la complejidad de búsqueda en este caso en particular.

c) Practique accediendo a los nodos con valor: 4, 5. Termine insertando el 9.

P2. Tries

Un *trie* (o árbol digital) es un tipo de árbol de búsqueda, que tiene la particularidad de no guardar valores en sus nodos, sino que la información está dada por la posición que tiene un nodo dentro del árbol. La búsqueda se realiza desde la raíz, eligiendo la i-ésima arista a recorrer según i-ésimo caracter del valor que se desea buscar. Note que la búsqueda tarda O(largo de la palabra buscada).

Según el alfabeto que se escoja se puede construir distintos tipos de tries; Por ejemplo el trie con el alfabeto $\Sigma = \{0, 1\}$ se dice que es un árbol digital y cada palabra ingresada debe tener a todos sus caracteres en Σ. En lo que sigue, ocuparemos la codificación binaria de nuestro alfabeto latino ($\{A, B, ..., Z\}$) dada por el código ASCII como entrada del *trie*.

A continuación se muestra un ejemplo de *trie*, al que se le insertaron las codificaciones ASCII de las letras A, B, E, I, G, L y R:

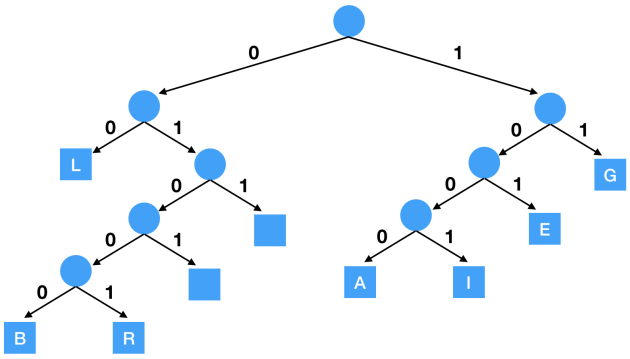


Figura 1: Árbol digital que contiene las letras A, B, E, I, G, L y R, según codificación ASCII

Dada la siguiente definición de nodo de Trie:

```
import numpy as np
class NodoiTrie:
  def __init__(self, izq = NodoeTrie(), der = NodoeTrie()):
      self.info = ''
      self.izq = izq
      self.der = der

class NodoeTrie:
  def __init__(self, info= None):
      self.info = info

class Trie:
  def __init__(self, raiz = NodoeTrie())
      self.raiz = raiz
```

Se le pide lo siguiente:

a) Implemente una función de inserción, que luego de insertar imprima el camino en el árbol hasta la posición de inserción.

b) Implemente una función de búsqueda, que entregue una tupla con el valor booleano si el elemento está en el árbol y el camino correspondiente.

c) Pruebe su implementación ingresando las secuencias binarias de las letras A, B, E, I, G, L, R en formato ASCII.

d) Tal como se menciona en el enunciado, se podría modificar el *trie* de modo que procese un alfabeto distinto. Discuta como implementar esta idea para ingresar secuencias de palabras del idioma español. (**Propuesto:** ¿Cómo se podría solucionar el caso en que se ingresa una palabra que es prefijo de otra?)

P3. Árboles de búsqueda digital

Considere un árbol de búsqueda digital (ABD) con la siguiente estructura:

```
class Hoja:
  def __init__():
    pass

class Nodo:
  def __init__(self, izq = None, valor="", der = None):
    self.valor = valor
    self.izq = izq
    self.der = der

class ABD:
  def __init__(self, raiz=None):
    self.raiz = raiz
```

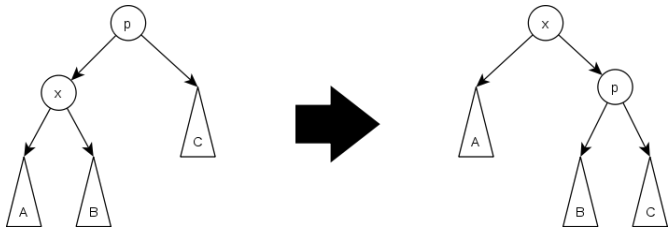
a) Implemente un algoritmo de inserción en el árbol.

b) Implemente un algoritmo de búsqueda en el árbol.

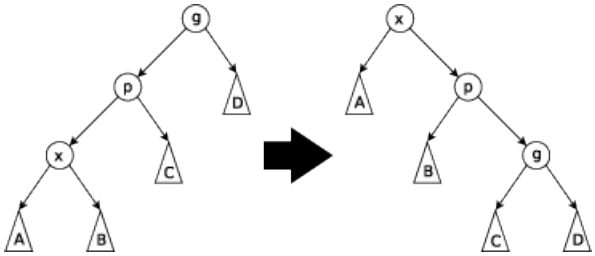
c) (**Propuesto**) Implemente un algoritmo de eliminación en el árbol.

Anexo: Casos de *splaying*

- **Caso 1: *zig*** *El padre de x es la raíz*
  Se hace una rotación simple sobre el padre (al igual que en árboles AVL), dejando a x como la nueva raíz.



- **Caso 2: *zig-zig*** *Tanto x como su padre son hijos del mismo lado (izquierdo o derecho) de su abuelo*
  El *zig-zig* se convierte en *zag-zag*, resultando en un árbol con x en la raíz, el padre de x como su hijo, y el abuelo de x como su nieto.



- **Caso 3: *zig-zag*** *El nodo x es hijo del lado derecho de su padre, y el padre es hijo izquierdo del abuelo (están en zigzag). El caso zag-zig es análogo*
  Se hace una rotación simple entre x y su padre. Luego, se rota x con su abuelo (ahora el padre) como en el caso 1 *(en la imagen, el árbol (p A B) cumple el rol del árbol A resultante en el diagrama del caso 1)*. Esto es igual a la rotación doble de los árboles AVL.