

<https://replit.com/@LuceroFederico/Ejercitacion-practica-de-Arboles-n-arios#main.py>

## Parte 1

**Importante:** Los ejercicios de esta primera parte tienen como objetivo codificar las diferentes funciones básicas necesarias para la implementar un Trie.

A partir de estructuras definidas como :

```
class Trie:
    root = None

class TrieNode:
    parent = None
    children = None
    key = None
    isEndOfWord = False
```

**Sugerencia 1:** Para manejar múltiples nodos, el campo children puede contener una estructura **LinkedList** conteniendo **TrieNode**

~~Para trabajar con cadenas, utilizar la clase string del módulo **algo.py**.~~

~~uncadena = **String**("esto es un string")~~

~~Luego es posible acceder a los elementos de la cadena mediante un índice.~~

```
print(uncadena[1])
>>>
```

## Ejercicio 1

Crear un módulo de nombre **trie.py** que **implemente** las siguientes especificaciones de las operaciones elementales para el **TAD Trie** .

**insert(T,element)**

**Descripción:** insert un elemento en T, siendo T un Trie.

**Entrada:** El Trie sobre la cual se quiere agregar el elemento (Trie) y el valor del elemento (palabra) a agregar.

**Salida:** No hay salida definida

**search(T,element)**

**Descripción:** Verifica que un elemento se encuentre dentro del **Trie**

**Entrada:** El Trie sobre la cual se quiere buscar el elemento (Trie) y

el valor del elemento (palabra)

**Salida:** Devuelve **False** o **True** según se encuentre el elemento.

## Ejercicio 2

Sabiendo que el orden de complejidad para el peor caso de la operación `search()` es de  $O(m |\Sigma|)$ .  
Proponga una versión de la operación `search()` cuya complejidad sea  $O(m)$ .

```
def search(T,element):
    if T.root == None: ### Si el arbol esta vacio ###
        return False

    TNode = T.root
    for ch in element: ### Recorre cada letra del elemento (palabra) ###
        index = traverse_list(TNode.children,ch) # O(n) #
        if index == None: ### No se encuentra la letra en la lista ###
            return False
        else: ### Se encuentra la letra en la lista ###
            TNode = TNode.children[index] ### Sigue al proximo nodo ###
    if TNode.isEndOfWord == True:
        return True
    else: return False
```

## Ejercicio 3

`delete(T,element)`

**Descripción:** Elimina un elemento se encuentre dentro del **Trie**

**Entrada:** El **Trie** sobre la cual se quiere eliminar el elemento (**Trie**)  
y el valor del elemento (palabra) a eliminar.

**Salida:** Devuelve **False** o **True** según se haya eliminado el elemento.

## Parte 2

### Ejercicio 4

Implementar un algoritmo que dado un árbol **Trie T**, un patrón **p** y un entero **n**, escriba todas las palabras del árbol que empiezan por **p** y sean de longitud **n**.

### Ejercicio 5

Implementar un algoritmo que dado los **Trie T1** y **T2** devuelva **True** si estos pertenecen al mismo documento y **False** en caso contrario. Se considera que un **Trie** pertenecen al mismo documento cuando:

1. Ambos Trie sean iguales (esto se debe cumplir)
2. El Trie **T1** contiene un subconjunto de las palabras del Trie **T2**

3. Si la implementación está basada en LinkedList, considerar el caso donde las palabras hayan sido insertadas en un orden diferente.

En otras palabras, analizar si todas las palabras de T1 se encuentran en T2.

Analizar el costo computacional.

```
""" Analizar el costo computacional """
""" create_word_list = O(n), equal = O(n) + O(n) """

def equal(T1,T2):
    res1 = []
    create_word_list(T1.root,res1,"",None)
    res2 = []
    create_word_list(T2.root,res2,"",None)
    if res1 == res2: ### Compara ambas listas ###
        return True
    else:
        return False
```

## Ejercicio 6

Implemente un algoritmo que dado el **Trie** T devuelva **True** si existen en el documento T dos cadenas invertidas. Dos cadenas son invertidas si se leen de izquierda a derecha y contiene los mismos caracteres que si se lee de derecha a izquierda, ej: **abcd** y **dcba** son cadenas invertidas, **gfdsa** y **asdfg** son cadenas invertidas, sin embargo **abcd** y **dc**u**ka** no son invertidas ya que difieren en un carácter.

## Ejercicio 7

Un corrector ortográfico interactivo utiliza un **Trie** para representar las palabras de su diccionario. Queremos añadir una función de auto-completar (al estilo de la tecla TAB en Linux): cuando estamos a medio escribir una palabra, si sólo existe una forma correcta de continuarla entonces debemos indicarlo.

Implementar la función **autoCompletar(Trie, cadena)** dentro del módulo **trie.py**, que dado el árbol **Trie T** y la cadena **“pal”** devuelve la forma de auto-completar la palabra. Por ejemplo, para la llamada **autoCompletar(T, ‘groen’)** devolvería **“land”**, ya que podemos tener **“groenlandia”** o **“groenlandés”** (en este ejemplo la palabra groenlandia y groenlandés pertenecen al documento que representa el Trie). Si hay varias formas o ninguna, devolvería la cadena vacía. Por ejemplo, **autoCompletar(T, ma’)** devolvería **“”** si T presenta las cadenas **“madera”** y **“mama”**.