

NAHMAN MARTINA TP TRIE

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**

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

```
#INSERT
def insert(T, elemento):
    if T.root == None:
        T.root = TrieNode()
        T.root.children = []
    current = T.root
    for i in range(len(elemento)):
        new_node = search_children(current.children, elemento[i])
        if new_node != None:
            current = new_node
        else:
            nodo = TrieNode()
            nodo.key = elemento[i]
            nodo.children = []
            current.children.append(nodo)
            nodo.parent = current
            current = nodo
    current.isEndOfWord = True
```

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.

```
#SEARCH
def search(T, palabra):
    current = T.root
    for letra in palabra:
        new_node = search_children(current.children, letra)
        if new_node != None:
            current = new_node
        else:
            return False
    if current.isEndOfWord == True:
        return True

def search_children(array, character):
    for i in range(len(array)):
        if array[i] != None:
            if array[i].key == character:
                return array[i]
    return None
```

Ejercicio 2

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

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.

```
def delete(T,palabra):
    #caso1 El elemento no se encuentra en el trie
    encontrado=search (T,palabra)
    if encontrado == False:
        return False

    else:
        #buscar el nodo fin de la palabra
        nodo_final=search_ultimonodo(T,palabra)
        nodo_final.isEndOfWord=False
        flag = True
        while flag:
            if nodo_final.children != None or len(nodo_final.children)>0 or nodo_final.parent != T.root or nodo_
                flag = False
            else:
                nodo_final=nodo_final.parent
        nodo_final.parent.remove(nodo_final)
        return True

def search_ultimonodo(T, palabra):
    current = T.root
    """falta si el len de la palabra el 1 devolver ese"""
    for i in range(len(palabra)-1):
        if i == (len(palabra)-1) and new_node.isEndOfWord == True:
            return new_node
        else:
            new_node = search_children(current.children, palabra[i])
```

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 4
def prefijo(T,p):
    pre=[]
    children=cycle(T.root.children)
    current=next(children)
    for i in range (len(p)):
        while current.key != p[i]:
            children=cycle(current.children)
            current=next(children)
        if current.key == p[i]:
            pre.append(current)

    if len(pre) !=0:
        return pre[len(pre)-1]#devuelve el ultimo nodo
    else:
        return None

def patron(T,p,n):
    palabras=[]
    nodo=prefijo(T,p)
    traverse_level(nodo,p,palabras,n)
    return palabras

def traverse_level(nodo,prefijo,palabras,n):
    if nodo.isEndOfWord==True and len(prefijo)==n:
        palabras.append(prefijo)
    for i in range (len(nodo.children)):
        traverse_level(nodo.children[i],prefijo+ nodo.children[i].key,palabras,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. 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.

```
#EJERCICIO 5
#COMPLEJIDAD O(n)

#Recorre Arbol y nos guarda la palabras
def traverse(nodo,prefijo,palabras):
    if nodo.isEndOfWord==True:
        palabras.append(prefijo)
    for i in range (len(nodo.children)):
        traverse(nodo.children[i],prefijo+ nodo.children[i].key,palabras)

def get_words(T):
    words = []
    traverse(T.root, '', words)
    return words

def is_sublist(lst1, lst2):
    return set(lst1).issubset(set(lst2))
```

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 **dcka** no son invertidas ya que difieren en un carácter.

```
#EJERCICIO 6
def palabras_invertidas(T):
    words= get_words(T)
    return has_inverted_list(words)

"""[::-1] se utiliza para obtener la list invertida"""
def has_inverted_list(lst):
    for sublist in lst:
        if sublist[::-1] in lst:
            return True
    return False
```

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".

```
#EJERCICIO 7
def autoCompletar(T, cadena):
    Node= T.root
    cadaux=""
    cont = 0
    aux=NodeFinal(T,cadena,Node,cont)
    Completar=Complete(aux.children,cadaux)
    return Completar

#recorro los nodos hasta encontrar uno que es una lista con mas de un elemto
def Complete(node,cadena):
    if node == None:
        return cadena
    if len(node)>1:
        return cadena
    else:
        for i in range (0,len(node)):
            if node[i].isEndOfWord == True:
                cadena = cadena+node[i].key
                return cadena
            else:
                cadena = cadena+node[i].key
                return Complete(node[i].children,cadena)
```

```
#recorro el arbol y devuelvo nodo final de palabra(puede no ser endofword)
def NodeFinal(T,element,node,cont):
    if node.children != None:
        if cont<len(element):
            for i in range (0,len(node.children)):
                if node.children[i].key == element[cont]:
                    cont = cont + 1
                    if cont == len(element):
                        return node.children[i]
                    return NodeFinal(T,element,node.children[i],cont)
```