Mauro José Sorbello Diaz Legajo: 14006

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(unacadena[1]))`~~
~~`>>> s`~~

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.

```
def insert(T, element):
    if T.root == None:
        #Si esta vacio creamos una lista y le asignamos un nodo en esa lista con el valor del primer caracter
        T.root = []
        newNode = TrieNode()
        T.root.append(newNode)
    return insertRec(T, element, T.root[0], 0, T.root)

def insertRec(T, element, node, j, parent):
    if j < len(element):
        #Primero nos fijamos si el node es una lista o esta vacio
        if node.children == None:
            #Si esta vacio creamos una lista y le asignamos un nodo en esa lista con el valor del primer caracter
            node.children = []
            newNode = TrieNode()
            newNode.parent = parent
            newNode.key = element[j]
            node.children.append(newNode)
        i = 0
        check = 0
        #Despues checkeamos si la lista tiene un nodo con la key del primer caracter
        node1 = node
        while (node1.children[i] != None) and (check == 0):
            if node1.children[i].key == element[j]:
                check = 1
            else:
                if i + 1 < len(node1.children):
                    i +=1
                else:
                    i = 0
                    if node1.children[i] != None:
                        node1 = node1.children[i]
                    if node1.children == None:
                        break
        #Si no tiene un nodo con la key del primer caracter, lo agregamos:
        if check == 0:
            newNode = TrieNode()
            newNode.parent = parent
            newNode.key = element[j]
            node.children.append(newNode)
            j +=1
            return insertRec(T, element, node.children[len(node.children) - 1], j, newNode)
        else:
            #Si lo tienen devolvemos el nodo hijo
            j +=1
            return insertRec(T, element, node.children[i], j, node)
    #Asignamos al fin de palabra.
    node.isEndOfWord = True
```

```python
def search(T, element):
    #Si esta vacio
    if T.root[0].children == None:
        return False
    #Si no esta vacio:
    return searchRec(T, element, T.root[0].children, 0)

def searchRec(T, element, node, j):
    # Si el nodo está vacío
    if node is None:
        return False
    #Verificar que la letra este:
    for i in range(0, len(node)):
        if node[i].key == element[j]:
            ind = i
            break
        else:
            ind = None
    #Si el ind es None, no se encuentra la letra, por ende es Falso
    if ind == None:
        return False
    else:
        #En el caso de que j sea menor que len(element) (Sigue habiendo letras que buscar)
        if j < len(element) - 1:
            #Si el hijo es None y quedan mas letras es falso
            if node[ind].children == None:
                return False
            #Si el hijo no es nulo, hacemos recursion
            else:
                return searchRec(T, element, node[ind].children, j+1)
        #Si j es igual al len(element) significa que ya no quedan letras por evaluar, entonces nos fijamos que el ultimo sea un endofword
        if node[ind].isEndOfWord:
            return True
        else:
            return False
```

Ejercicio 2

Sabiendo que el orden de complejidad para el peor caso de la operación search() es de O(m |Σ|). 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, element):
    find = search(T, element)
    #El elemento no se encuentra en el trie.
    if find == False:
        return False
    else:
       #El elemento está presente y es parte de otro mas largo
        L = lastNodeandUnique(T,element,T.root[0].children, 0)
        for j in range(0, len(L)):
            node = L[j]
            if L[j] == False:
                unico = False
                indUnico = j
        #El elemento es un prefijo
        if node[0].children != False:
            node.isEndOfWord = False
            return True
        #El elemento no es unico
        unicoNode = T.root[0].children
        if unico == False:
            while node[0] != unicoNode[indUnico]:
                node = node[0].parent
                node[0].pop
            return True
        else:
            #El elemento es unico
            for i in range(0, len(unico)):
                if unicoNode[i].key == element[j]:
                    indice = i
                break
            while node[0] != unico[indice]:
                node = node[0].parent
                node[0].pop
            return True

def lastNodeandUnique(T, element, node, j):
    for i in range(0, len(node)):
        if node[i].key == element[j]:
            indice = i
            break
        else:
            indice = None
    lista = []
    #En el caso de que j sea menor que len(element) (Sigue habiendo letras que buscar)
    if j < len(element) - 1:
        if len(node[indice].children) - 1 > 0:
            lista.append(indice, False)
        return searchRec(T, element, node[indice].children, j+1)
    else:
        lista.append(node[indice])
        return lista
```

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**.

```
def allWords(T, p, n):
    #Buscamos el patron
    node = T.root[0].children
    for i in range(0, len(node)):
        if node[i].key == p:
            ind = i
            break
        else:
            ind = None
    #Si el ind = None, el patron no esta
    if ind == None:
        return None
    else:
        words = []
        return allWordsRec(T, n, words, 0, ind, node, 0)

def allWordsRec(T, n, list, j, indice, node, i):
    if j < n:
        list.append(node[indice])
        if i < len(node[indice].children) - 1:
            node = node[indice].children
            return allWordsRec(T, n, list, j+1, i+1, node, i+1)
        else:
            return allWordsRec(T, n, list, j+1, indice, node[indice].children, 0)
    else:
        return list
```

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.

```python
#Complejidad O(n^2), hay una recursiva dentro de un bucle

def sameDoc (T1, T2):
    #Si ambas son nulas, o alguna de ellas es nula es falso
    if T1.root == None or T2.root == None:
        return False
    else:
        for i in range(0, len(T2.root[0].children) - 1):
            check = sameDocRec(T1.root[0].children, T2.root[i].children)
            if check == False:
                return False
        if check == True:
            return True
```

```python
def sameDocRec(node1, node2):
    for i in range(0, len(node1)):
        if node1[i].key == node2[0].key:
            indice = i
            break
        else:
            indice = None
    #Si el indice es None, no esta la palabra
    if indice == None:
        return False
    else:
        if node2[0].children != None:
            #Si el hijo es None y quedan mas letras es falso
            if node1[indice].children == None:
                return False
            #Si el hijo no es nulo, hacemos recursion
            else:
                return sameDocRec(node1[indice].children, node2[0].children)
        #Si j es igual al len(node2) significa que ya no quedan letras por evaluar, entonces nos fijamos que el ultimo sea un endofword
        if node1[indice].isEndOfWord:
            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 **dcka** no son invertidas ya que difieren en un carácter.

```
def getcadena(T):
    if T.root[0].children == None:
        False
    else:
        for i in range(0, len(T.root[0].children)):
            cadena = []
            cadenaVect = []
            check = getcadenaRec(T, T.root[0].children, T.root[0].children, cadena, i)
            if check == True:
                cadenaVect.append(1)
            if len(cadenaVect) > 0:
                return True
            else:
                return False

def getcadenaRec(T, node, node1, cadena, i):
    letra = node[i].key
    cadena.append(letra)
    string = ""
    if node[i].children != None:
        getcadenaRec(T, node[i].children, node1, cadena, 0)
    if node[i].children == None:
        if node[i].isEndOfWord == True:
            string="".join(cadena)
            string = string[::-1]
            check = search(T, string)
            cadena = []
        return check
```

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"**.

```
def autoCompletar(Trie, cadena):
    if Trie.root == None:
        return None
    else:
        cadena2 = []
        return autoCompletarRec(T, cadena,  T.root[0].children, 0, cadena2)

def autoCompletarRec(T, cadena, node, j, cadena2):
    # Si el nodo está vacío
    if node is None:
        return False
    #Verificar que la letra este:
    for i in range(0, len(node)):
        if node[i].key == cadena[j]:
            ind = i
            break
        else:
            ind = None
    #Si el ind es None, no se encuentra la letra, por ende es Falso
    if ind == None:
        return False
    else:
        #En el caso de que j sea menor que len(cadena) (Sigue habiendo letras que buscar)
        if j < len(cadena) - 1:
            #Si el hijo es None y quedan mas letras es falso
            if node[ind].children == None:
                return False
            #Si el hijo no es nulo, hacemos recursion
            else:
                return autoCompletarRec(T, cadena, node[ind].children, j+1, cadena2)
        else:
            while node[ind].isEndOfWord != True:
                node = node[ind].children
                letra = node[ind].key
                cadena2.append(letra)
            if node[ind].isEndOfWord == True:
                string="".join(cadena2)
                return string
```