Algoritmo y Estructura de Datos II

Alumno:Gonzalez Sanchez Gabriel
Legajo: 12007

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.

```
#Inserta un elemento en un Trie
#Ingresa un Trie y un elemento
def insert(T,element):
  if T.root == None:
    Node = TrieNode()
    Node.children = LinkedList()
    T.root = Node
  insertR(T,element,0,T.root)
```

```
def insertR(T,element,charposition,currentNode):

  #Recorro los hijos
  currentChild = currentNode.children.head
  flag = False
  while currentChild!=None and flag == False:
    #Primer Caso: Encuentra una coincidencia de caracteres
    if currentChild.value.key == element[charposition]:
      Node = currentChild.value
      flag = True
    currentChild = currentChild.nextNode



  if flag == False:
    #Segundo Caso: No hay antecedentes
    Node = TrieNode()
    Node.key = element[charposition]
    Node.children = LinkedList()
    Node.parent = currentNode
    add(currentNode.children, Node)

  #Determino si el caracter agregado es el final de la palabra
  if charposition == len(element)-1:
    Node.isEndOfWord = True
  else:
    #Continuo con el siguiente caracter
    insertR(T,element,charposition+1,Node)

#Busca un elemento en un Trie
#Devuelve True si existe el elemento, devuelve None si no existe
def search(T,element):
  if T.root == None:
    return False
  else:
    return searchR(T.root.children.head,0,element)









def searchR(currentNode,charPosition,element):
```

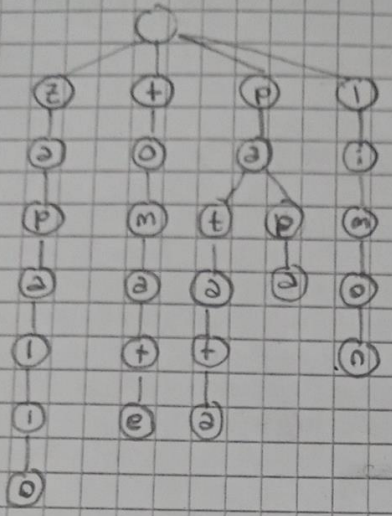
```
flag = False
while currentNode!=None and flag==False:

  #veo coincidencia
  if currentNode.value.key == element[charPosition]:
    flag = True
    #Caso que exista la palabra
    if currentNode.value.isEndOfWord == True and charPosition==(len(element)-1):
      return True
    elif currentNode.value.isEndOfWord == None and charPosition==(len(element)-1):
      return False
    elif charPosition<(len(element)-1):
      charPosition += 1
      return searchR(currentNode.value.children.head,charPosition,element)

  currentNode = currentNode.nextNode
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).



En este ejemplo itera 4 veces por las comparaciones necesarias por la longitud de palabra y agregara una comparación por cada prefijo diferente

Supongamos que cada nodo tiene k hijos y como máximo $|\Sigma|$ hijos por lo que:

$$T(n) = \sum_{i=0}^{m} k_i \quad \text{cuando } k = |\Sigma| \text{ es } T(n) = m \cdot |\Sigma| = c \cdot m$$

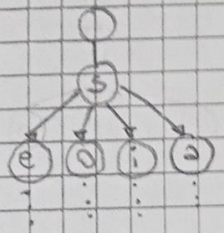
ya que $|\Sigma| \cdot m$ representa una cota superior para el resto de casos menores su complejidad es $O(m)$

Ejercicio 2.: Sabiendo que el orden de complejidad para el peor caso de search es de $O(m \cdot |\Sigma|)$. Propongo una version de la operación search cuyo complejidad sea $O(m)$

search: verifica la presencia de un elemento s de longitud m en T

El peor caso se da cuando las cadenas de caracteres no comparten prefijos

Ej: Diccionario = {'Señor', 'Sopa', 'Siesta', 'Sapo'}



Su prefijo común es $ pero al momento de recorrer el segundo caracter es necesario recorrer la lista de hijos de $. En este ejemplo son 4 pero podrían ser como máximo $|\Sigma|$.

Los casos promedios son de $O(m)$ ya que en general los alfabetos no se constituyen por palabras que conformen combinaciones como 'hk!zw'. Así que buscar 'papa' en un diccionario = {zapallo, tomate, patata, limon, papa} es de complejidad $O(m)$.

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.

```
#Elimino elemento de una Trie
#Ingresa un Trie y un elemento
#Devuelve True si logra eliminar el elemento y False si no lo logra
def delete(T,element):
  if T.root == None:
    return False
  else:
    lastNode = findEndEle(T,element)
    if lastNode != None:
      return deleteword(T,element,lastNode)
    else:
      return False
```

```
#Busca el nodo correspondiente al ultimo caracter de un elemento en un Trie
#Ingresa un Trie y un elemento
#Devuelve el nodo
def findEndEle(T,element):
  if T.root == None:
    return None
  else:
    return findEndEleR(T.root.children.head,0,element)

def findEndEleR(currentNode,charPosition,element):
  flag = False
  while currentNode!=None and flag==False:
    if currentNode.value.key== element[charPosition]:
      flag = True
      if currentNode.value.isEndOfWord == True and charPosition == (len(element)-1):
        return currentNode.value
      elif currentNode.value.isEndOfWord == None and charPosition == (len(element)-1):
        return None
      elif charPosition < (len(element)-1):
        charPosition += 1
        return findEndEleR(currentNode.value.children.head, charPosition,element)
    currentNode = currentNode.nextNode
  return None

#Elimina un elemento de un Trie
#Ingresa un Trie, un elemento y el nodo correspondiente al ultimo caracter del elemento
def deleteword(T,element,currentNode):
  if length(currentNode.children)!=0:
    currentNode.isEndOfWord = None
  else:
    flag = False
    while currentNode.value!=None and flag == False:
      parent = currentNode.parent
      if length(parent.children)>1:
        flag = True
      deleteLL(parent.children,currentNode)
      currentNode = parent
  return True
```

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**.

```
#Busca elementos en un trie con patron p y longitud n
#Ingresa un Trie, un patron y la longitud
#Devuelve None si no es posible buscar, delocontrario devuelve una linked list
def findPattern(T,p,n):
  if T.root == None:
    return None
  if len(p)>n:
    return None
  else:

    L = LinkedList()
    lastNode = findEndEle(T,p)
    if lastNode!=None:
      if p == n:
        add(L,p)
      else:
        currentNode = lastNode.children.head
        while currentNode!=None:
          findPatternR(currentNode,p,n,L)
          currentNode = currentNode.nextNode
    return L
```

```
def findPatternR(currentNode,word,maxLen,L):
  if currentNode!=None:
    if len(word)<maxLen:
      word = word + currentNode.value.key
    if len(word)==maxLen and currentNode.value.isEndOfWord==True:
      add(L,word)
    elif len(word)==maxLen and currentNode.value.isEndOfWord==None:
      return
    else:
      currentNode = currentNode.value.children.head
      while currentNode!=None:
        findPatternR(currentNode,word,maxLen,L)
        currentNode = currentNode.nextNode
  return
```

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.

```
#Determina si todos los elementos de un Trie T1 estan dentro de otro Trie T2
def subTrie(T1, T2, string):
  flag = True
  currentNode = T1.children.head
  while currentNode != None:
    word = string + currentNode.value.key
    if currentNode.value.isEndOfWord:
      flag = search(T2, word)
    if flag == False:
      return False
    else:
      printTrie(currentNode.value, word)
    currentNode = currentNode.nextNode
```

Análisis de Complejidad: El algoritmo recorre el árbol T1 para conformar un elemento de T1 y luego utiliza search para buscar ese elemento en T2, sindo m el tamaño de una palabra en T1, k la cantidad de palabras en T1, |A| el alfabeto de T1 y |B| el alfabeto de T2, entonces al algoritmo en el peor caso le tomaría una complejidad |A|*m en construir la palabra y una complejidad |B|*m en buscar la palabra en T2. Este caso seria solamente para una palabra de T1, por lo que que deberia ocurrir para todas las palabras por lo que tomaría una complejidad k*(|A|*m+|B|*m)