Giraudo Ignacio 13855 LCC 2023

# 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`

```python
#Ejercicio 1
def insert(T, element):
    if T.root.children is None:
        T.root.children = LinkedList()
    insert_r(T.root.children, None, element, 0)

def insert_r(list, node, str, str_index):
    if str_index >= len(str): return
    char = str[str_index]
    list_node = search_list_with_trie_nodes(list, char)
    trie_node = None
    if list_node is not None:
        trie_node = list_node.value

    if trie_node is None:
        new_trie_node = TrieNode(node, LinkedList(), char, len(str) - 1 == str_index)
        add(list, new_trie_node)
        insert_r(new_trie_node.children, new_trie_node, str, str_index + 1)
    else:
        if str_index == len(str) - 1:
            trie_node.isEndOfWord = True
        insert_r(trie_node.children, trie_node, str, str_index + 1)

```

**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.

```python
def search_list_with_trie_nodes(L, char):
    current = L.head
    while current != None:
        if current.value.key == char:
            return current
        current = current.nextNode
    return

def search(T, element):
    list_node = search_r(T.root.children, element, 0)
    trie_node = None
    if list_node is not None: trie_node = list_node.value
    if trie_node is None: return False
    return trie_node.isEndOfWord

# retorna el último nodo de la palabra a buscar
def search_r(list, str, str_index):
    if str_index >= len(str): return
    char = str[str_index]
    list_node = search_list_with_trie_nodes(list, char)

    if list_node is None:
        return
    else:
        if str_index == len(str) - 1:
            return list_node
        return search_r(list_node.value.children, str, str_index + 1)
```

## 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).

**Una versión de search con complejidad O(m) sería usando arrays en vez de LinkedList y asignándole a cada elemento del alfabeto una key para identificarla dentro del array. Así al buscar accederíamos con una key al array, O(1), y luego simplemente debemos recorrer la longitud de la palabra a buscar, lo cual daría la complejidad de 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.

```python
#Ejercicio 3

def delete(T, element):
    list_node = search_r(T.root.children, element, 0)
    trie_node = None
    if list_node is not None: trie_node = list_node.value
    if trie_node is None or not trie_node.isEndOfWord: return False

    # El elemento es parte de otro elemento más largo
    if trie_node.children is not None:
        if trie_node.children.head is not None:
            trie_node.isEndOfWord = False
            return

    # El elemento está presente y es único (ninguna parte del elemento contiene a otro)
    while trie_node.parent is not None:
        delete_list(trie_node.parent.children, trie_node)
        # El elemento está presente y tiene al menos un elemento incluido.
        if trie_node.parent.isEndOfWord or trie_node.parent.children.head is not None: break
        trie_node = trie_node.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**.

```python
def print_word_with_length(T, element, n):
    node = search_r(T.root.children, element, 0)
    if node is not None:
        stack = LinkedList()
        push(stack, element)
        print_word_with_length_r(node.value.children.head, stack, n)

def print_word_with_length_r(node, stack, n):
    if node is None: return
    word = access(stack, 0) + node.value.key
    if node.value.isEndOfWord and len(word) == n:
        print(word)
    if node.value.children is not None:
        push(stack, word)
        print_word_with_length_r(node.value.children.head, stack, n)
        pop(stack)
    print_word_with_length_r(node.nextNode, stack, 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.

```python
def are_from_the_same_document(T1, T2):
    words1 = T1.get_words()
    words2 = T2.get_words()
    return is_sublist(words1, words2)


def is_sublist(list1, list2):
    node1 = list1.head
    while node1 is not None:
        if search(list2, node1.value) is None:
            return False
        node1 = node1.nextNode
    return True
```

Analizar el costo computacional.

**El orden de complejidad es O(m|Σ| + m*n), siendo m la cantidad de nodos de T1 y n la de T2. Es O(m |Σ|) ya que recorre todo el trie para obtener las palabras y O(m^2) ya que en el peor de los casos compara todas las palabras de cada árbol.**

# 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 has_inverted_words(T):
    words = T.get_words()
    node = words.head
    while node is not None:
        inv_word = invert(node.value)
        if search_list(words, inv_word) is not None:
            return True
        node = node.nextNode
    return False

def invert(str):
    inv_str = ""
 for i in range(len(str), 0, -1):
        inv_str += str[i-1]
    return inv_str
```

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 auto_complete(T, str):
    list_node = search_r(T.root.children, str, 0)
    trie_node = None
    if list_node is not None: trie_node = list_node.value
    ret_str = String("")

    while trie_node is not None:
        # Para evitar que se sume el último caracter de str
        if trie_node is not list_node.value:
            ret_str = concat(ret_str, String(trie_node.key))
        if trie_node.children is None or trie_node.isEndOfWord:
            return ret_str
        if trie_node.children is not None and length(trie_node.children) > 1:
            return ret_str
        trie_node = trie_node.children.head.value
    return String("")
```