TRABAJO PRÁCTICO ÁRBOLES N-ARIOS

PARTE 1

INTRODUCCIÓN

```
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`**

```python
# ----------------------------
# Estructura de Lista Enlazada, comoo vamos a usar listas enlazadas, necesitás tener definida una LinkedList y Node
# ----------------------------
class Node:
    def __init__(self, value=None):
        self.value = value   # Puede ser un TrieNode
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, value):
        """Inserta un nuevo nodo al inicio"""
        new_node = Node(value)
        new_node.next = self.head
        self.head = new_node

    def search(self, key):
        """Busca un TrieNode con cierto key dentro de la lista"""
        current = self.head
        while current:
            if current.value.key == key:
                return current.value
            current = current.next
        return None
```

```python
class TrieNode:
    def __init__(self, key=None, parent=None):
        self.key = key                    # Carácter
        self.parent = parent              # Nodo padre
        self.children = LinkedList()      # Lista de hijos (TrieNodes)
        self.isEndOfWord = False          # Marca de fin de palabra

class Trie:
    def __init__(self):
        self.root = TrieNode("*")  # raíz con símbolo especial
```

EJERCICIO 1

---

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.

```python
def search(T: Trie, element: str) -> bool:
    """Verifica si una palabra está en el Trie T"""
    current = T.root
    for char in element:
        child = current.children.search(char)
        if child is None:
            return False
        current = child
    return current.isEndOfWord
```

```python
def insert(T: Trie, element: str):
    """Inserta una palabra en el Trie T"""
    current = T.root
    for char in element:
        # buscamos si ya existe un hijo con este char
        child = current.children.search(char)
        if child is None:
            # si no existe, lo creamos y lo insertamos en la lista de hijos
            new_node = TrieNode(key=char, parent=current)
            current.children.insert(new_node)
            child = new_node
        # avanzamos al hijo
        current = child
    # marcamos el fin de la palabra
    current.isEndOfWord = True
```

EJERCICIO 2

Complejidad actual

Tu `Trie.search` usa listas enlazadas para los hijos:

```
child = current.children.search(char)
```

- `current.children.search(char)` recorre **todos los hijos** hasta encontrar el carácter.
- Si un nodo tiene hasta `|Σ|` hijos posibles (Σ = tamaño del alfabeto), entonces el peor **caso** es recorrer todos los hijos.

Por eso la complejidad del **peor caso** es:

$$O(m \cdot |\Sigma|)$$

- `m` = longitud de la palabra
- `|Σ|` = número máximo de hijos posibles de un nodo

Cómo reducir a O(m)

Para lograr que la búsqueda sea **O(m)**, tenemos que acceder al hijo correcto **sin recorrer todos los hijos**.

Eso significa: **usar un acceso directo por clave**, en lugar de lista enlazada.

Alternativa: usar un diccionario

En vez de `LinkedList` para los hijos, cada `TrieNode` tendría:

```
self.children = {}  # diccionario
```

- Clave = carácter
- Valor = TrieNode hijo correspondiente

Así, buscar un hijo se hace con:

```
child = current.children.get(char)
```

- Esto **tarda O(1)** en promedio, porque acceder a un diccionario por clave es constante.
- Recorres `m` caracteres de la palabra, entonces:

$$\text{Complejidad total} = O(m)$$

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
def delete(T, element):
    current = T.root

    # 1. Buscar palabra
    for char in element:
        child = current.children.search(char)
        if child is None:
            return False  # palabra no existe
        current = child

    # 2. Marcar nodo final como no fin de palabra
    if not current.isEndOfWord:
        return False  # la palabra no estaba marcada como existente

    current.isEndOfWord = False

    # 3. Eliminar nodos innecesarios hacia atrás
    while current.parent is not None:
        if current.children.head is None and not current.isEndOfWord:
            # eliminar este nodo de la lista de hijos de su padre
            parent = current.parent
            remove_from_linkedlist(parent.children, current.key)
            current = parent
        else:
            break

    return True
```

```python
def remove_from_linkedlist(ll, key):
    current = ll.head
    prev = None
    while current:
        if current.value.key == key:
            if prev:
                prev.next = current.next
            else:
                ll.head = current.next
            return
        prev = current
        current = current.next
```

PARTE 2

EJERCICIO 4

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

```
class Trie:
    def __init__(self):
        self.root = TrieNode("*")  # raíz con símbolo especial

    # ---------------------------- Ejercicio 4 ----------------------------
    def insert(self, word):
        """Inserta una palabra en el Trie"""
        node = self.root
        for char in word:
            child = node.children.search(char)
            if not child:  # si no existe, lo creo
                child = TrieNode(char, parent=node)
                node.children.insert(child)
            node = child
        node.isEndOfWord = True

    def _find_node(self, prefix):
        """Devuelve el nodo final del prefijo, o None si no existe"""
        node = self.root
        for char in prefix:
            node = node.children.search(char)
            if not node:
                return None
        return node

    def _dfs(self, node, path, n, result):
        """Recorrido en profundidad para buscar palabras de longitud n"""
        if len(path) == n:
            if node.isEndOfWord:
                result.append("".join(path))
            return

        current = node.children.head
        while current:
            child = current.value
            self._dfs(child, path + [child.key], n, result)
            current = current.next

    def palabras_con_prefijo_y_longitud(self, prefijo, n):
        """Devuelve todas las palabras que comienzan con prefijo y tienen longitud n"""
        start_node = self._find_node(prefijo)
        if not start_node:
            return []  # prefijo no encontrado

        result = []
        self._dfs(start_node, list(prefijo), n, result)
        return result
```

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** pertenece 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
class TrieComparator:

    def get_all_words(trie):
        """Devuelve un conjunto con todas las palabras del trie"""
        result = set()

        def dfs(node, path):
            if node.isEndOfWord:
                result.add("".join(path))
            current = node.children.head
            while current:
                child = current.value
                dfs(child, path + [child.key])
                current = current.next

        dfs(trie.root, [])
        return result

    def same_document(T1, T2):
        """Devuelve True si ambos tries representan el mismo documento"""
        palabras_T1 = TrieComparator.get_all_words(T1)
        palabras_T2 = TrieComparator.get_all_words(T2)
        return palabras_T1 == palabras_T2
```

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.

```python
class TrieUtils:

    def get_all_words(trie):
        """Devuelve un conjunto con todas las palabras del trie"""
        result = set()

        def dfs(node, path):
            if node.isEndOfWord:
                result.add("".join(path))
            current = node.children.head
            while current:
                child = current.value
                dfs(child, path + [child.key])
                current = current.next

        dfs(trie.root, [])
        return result

    def has_inverted_pairs(trie):
        """Devuelve True si existen dos palabras invertidas en el trie"""
        words = TrieUtils.get_all_words(trie)
        for word in words:
            if word[::-1] in words and word[::-1] != word:  # evitar palíndromos
                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 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 "" (cadena vacia) si **T** presenta las cadenas **"madera"** y **"mama"**.

```python
class Trie:
    def __init__(self):
        self.root = TrieNode("*")  # raíz con símbolo especial

    # ---------------------------- Ejercicio 7 ----------------------------

    def _find_node(self, prefix):
        """Devuelve el nodo final del prefijo, o None si no existe"""
        node = self.root
        for char in prefix:
            node = node.children.search(char)
            if not node:
                return None
        return node

    def autoCompletar(self, prefijo):
        """Devuelve la continuación única de prefijo, o "" si hay varias o ninguna"""
        node = self._find_node(prefijo)
        if not node:
            return ""   # prefijo no existe

        sufijo = []
        while True:
            # contar hijos
            current = node.children.head
            if not current:  # sin hijos → no hay nada que completar
                return ""

            # ver si hay más de un hijo
            if current.next:
                return ""  # varias ramas → ambigüedad → no autocompleta

            # si llegamos acá → exactamente 1 hijo
            child = current.value
            sufijo.append(child.key)
            node = child

            # si la palabra terminó y ya no hay más ramas, devolvemos el sufijo
            if node.isEndOfWord and not node.children.head:
                return "".join(sufijo)
```