```python
class Trie:
    root = None


class TrieNode:
    parent = None
    children = None
    key = None
    isEndOfWord = False


def newNode(key):
    new = TrieNode()
    new.children = []
    new.key = key
    return new


"""EJERCICIO 1"""


def isLetterOnChildren(node: TrieNode, letter: str) -> [bool, TrieNode | None]:
    """Returns a tuple: bool is found and the node itself (None if not found)."""
    for n in node.children:
        if n.key == letter:
            return True, n
    return False, None


def insert(T, element):
    if T.root is None:
        T.root = newNode(None)
    node = T.root
    for letter in element:
        letterFound, new = isLetterOnChildren(node, letter)
        if not letterFound:
            new = newNode(letter)
            node.children.append(new)
        new.parent = node
        node = new
    node.isEndOfWord = True
    return


def searchWordLastLetterNode(T, element) -> [bool, TrieNode | None]:
    """Given a word, returns True if it is on the Trie and the last letter node, otw False
and None"""
    if T is None or T.root is None or not element:
        raise Warning("Empty Trie, root or word")
    node = T.root
    for letter in element:
        letterFound, node = isLetterOnChildren(node, letter)
        if not letterFound:
            return False, None
    return True, node


def search(T, element):
    return searchWordLastLetterNode(T, element)[0]


"""EJERCICIO 2:
Para que la operación search() tenga un órden de complejidad O(m), es decir, que la
cardinalidad del
alfabeto (|S|) no influya en el tiempo de búsqueda, podríamos modificar la estructura Trie
de forma tal que cada
campo children se exprese mediante un TAD en la que la búsqueda de un elemento dentro de
ella sea de O(1). De esta
forma, lo único que resta verificar es que todas las m letras/partes del elemento estén en
```

```
el Trie. Esto puede
lograrse utilizando, por ejemplo, un TAD que en clase hemos empezado a ver y satisface
esta necesidad, hablamos del
Diccionario, de esta forma el acceso por key es inmediato."""

"""EJERCICIO 3"""


def delete(T, element):
    wordFound, lastLetterNode = searchWordLastLetterNode(T, element)
    if not wordFound:
        return False
    lastLetterNode.isEndOfWord = False

    # A word can be a leaf nor an inner node
    if lastLetterNode.children is []:  # is inner, job done
        return False
    node = lastLetterNode  # is leaf
    while True:
        if node is None or node.isEndOfWord:  # reached the root or a new word ending
            return True
        nodeParent = node.parent
        nodeParent.children.remove(node)
        node = nodeParent


"""EJERCICIO 7"""


def autoCompletar(T, prefijo):
    """Devuelve la parte restante de una raíz de 2 o + palabras que tengan a -prefijo-
como parte de su raíz."""
    cadenaEncontrada, nodoUltimaLetra = searchWordLastLetterNode(T, prefijo)
    if not cadenaEncontrada:
        return ''
    resto = ''
    while True:
        if len(nodoUltimaLetra.children) > 1:
            return "'" if not resto else resto
        nodoSigue = nodoUltimaLetra.children[0]
        resto += nodoSigue.key
        nodoUltimaLetra = nodoSigue


"""EJERCICIO 4"""


def withPrefix(T: Trie, prefix: str, size: int) -> list:
    """Returns a list with all words with -prefix- and length -size-."""

    def withPrefixR(node, size, finalList, remaining):
        if size == 0 and node.isEndOfWord:
            finalList.append(remaining)

        for child in node.children:
            withPrefixR(child, size - 1, finalList, remaining + child.key)  # decrease
remaining while adding key
        return finalList  # return when loop finishes (all possible situations where
verified)

    wordFound, lastLetterNode = searchWordLastLetterNode(T, prefix)
    if not wordFound or size <= 0 or size <= len(prefix):
        return []  # prefix not found or input errors

    remainingSize = size - len(prefix)  # so to see when it reaches zero
    return withPrefixR(lastLetterNode, remainingSize, [], prefix)


"""EJERCICIO 5"""
```

```python
def sameDocument(T1: Trie, T2: Trie) -> bool:
    """Tells if T1 and T2 have all same words, iow, they are the same Trie.
    El costo es de O(m*n*log(n)), donde m es el número total de nodos y n es el número de hijos por cada nodo m.
    El costo n*log(n) se debe a que cada llamada aplica dos ordenamientos de listas."""

    def nodeKey(node):
        return node.key

    def sameDocumentR(node1, node2):
        if len(node1.children) != len(  # here are all the logical cases where a doc-node is considered != from another
                node2.children) or node1.key != node2.key or node1.isEndOfWord != node2.isEndOfWord:
            return False
        keySorted1 = sorted(node1.children, key=nodeKey)
        keySorted2 = sorted(node2.children, key=nodeKey)
        for i in range(len(keySorted1)):  # checks that all nodes satisfy sameDocument condition
            if not sameDocumentR(keySorted1[i], keySorted2[i]):  # only if a False condition is found, return
                return False
        return True  # else, continue up to here - all nodes and their children checked

    if T1 and T2:
        return sameDocumentR(T1.root, T2.root)
    return False


"""EJERCICIO 6"""


def hasReversedStrings(T):
    node = T.root
    stack = [(node, '')]
    while stack:  # while there is still a node to be searched
        node, word = stack.pop()  # bring back the last node
        if node.isEndOfWord and search(T, word[::-1]):  # when an EOW was found, search its reversed form
            return True
        for child in node.children:
            stack.append((child, word + child.key))  # append a node so to analyze it
    return False
```