2.3 Correcteur orthographique avec un arbre lexicographique

Application Comment stocker les mots d'un dictionnaire pour réaliser un correcteur orthographique ? Pour un mot donné, on aimerait pouvoir rapidement trouver un mot proche du dictionnaire. Si l'on stockait les mots du dictionnaire dans une table de hachage, on perdrait toute information de proximité entre les mots. Il vaut mieux les stocker dans un arbre lexicographique, aussi appelé arbre de préfixes ou *trie* (à prononcer en anglais).

Définition Une *trie* est un arbre qui stocke un ensemble de mots. Les arcs d'un nœud vers ses fils sont étiquetés par des lettres distinctes. Chaque mot du dictionnaire correspond alors à un chemin de la racine vers un nœud de l'arbre. Les nœuds ont des marques pour distinguer ceux qui correspondent à des mots du dictionnaire de ceux qui sont seulement des préfixes stricts de mots du dictionnaire, voir figure 2.2.

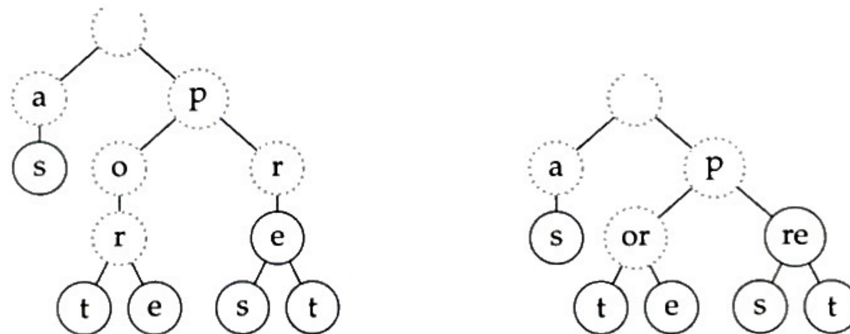

FIGURE 2.2 – Un arbre lexicographique pour les mots *as, port, pore, pré, près, prêt* (mais sans les accents). Pour l'illustration, l'étiquette de l'arête est indiquée sur le nœud du fils. Les parenthèses marquent les nœuds correspondant à des mots du dictionnaire. À droite, une *Patricia trie* pour le même dictionnaire.

Correction orthographique Avec une telle structure, il est facile de chercher un mot du dictionnaire qui soit à distance *dist* d'un mot donné, pour la distance d'édition de Levenshtein définie section 3.2 page 52. Il suffit de simuler les opérations d'édition à chaque nœud, et d'effectuer les appels récursifs de la recherche avec le paramètre *dist - 1*.

Variante Il existe une structure plus compacte qui fusionne les nœuds tant qu'il y a un nœud avec un unique fils. Un nœud est alors étiqueté par un mot, plutôt que par une simple lettre, voir figure 2.2. Cette structure, optimale en mémoire et en temps de parcours, porte le nom de *Patricia trie*.

```python
from string import ascii_letters    # in python2 one would import letters


class Trie_Node:
    def __init__(self):
        self.isWord = False
        self.s = {c: None for c in ascii_letters}


def add(T, w, i=0):
    if T is None:
        T = Trie_Node()
    if i == len(w):
        T.isWord = True
    else:
        T.s[w[i]] = add(T.s[w[i]], w, i + 1)
    return T


def Trie(S):
    T = None
    for w in S:
        T = add(T, w)
    return T


def spell_check(T, w):
    assert T is not None
    dist = 0
    while True:   # tente de distances de plus en plus grandes
        u = search(T, dist, w)
        if u is not None:
            return u
        dist += 1


def search(T, dist, w, i=0):
    if i == len(w):
        if T is not None and T.isWord and dist == 0:
            return ""
        else:
            return None
    if T is None:
        return None
    f = search(T.s[w[i]], dist, w, i + 1)       # correspondance
    if f is not None:
        return w[i] + f
    if dist == 0:
        return None
    for c in ascii_letters:
        f = search(T.s[c], dist - 1, w, i)       # insertion
        if f is not None:
            return c + f
        f = search(T.s[c], dist - 1, w, i + 1)  # substitution
        if f is not None:
            return c + f
    return search(T, dist - 1, w, i + 1)        # suppression
```