HE VD IG

Série 0x2a

Solution

Tries (arbres de préfixes) — optionnelle

INFO2-TIN — 3 mars 2026, 7efbe5

Exercice 1 : Concepts

(a) Un trie (arbre de préfixes) est surtout adapté pour :

A. Rechercher rapidement des mots par préfixe
B. Trier des entiers en place
C. Calculer des distances euclidiennes
D. Compresser des tableaux de nombres réels

(a) **A**

(b) Considérez un trie contenant les mots : `chat`, `chien`, `chou`, `choix`. Quel est le plus long préfixe commun à tous ces mots ?

`ch`

(c) Donnez la complexité en temps (Big-O) d'une recherche d'un mot de longueur `L` dans un trie, en supposant un alphabet de taille fixe.

`O(L)`

Exercice 2 : Définition de structures

(a) Proposez une structure C pour un noeud de trie contenant :
— un tableau de pointeurs vers enfants (alphabet ASCII minuscule `a..z`),
— un indicateur `is_word` qui vaut 1 si le noeud termine un mot.

```
#define ALPHA 26
typedef struct trie_node {
    struct trie_node *child[ALPHA];
    unsigned is_word : 1;
} TrieNode;
```

Exercice 3 : Programmation

(a) Écrire une fonction `trie_new` qui alloue un noeud vide et initialise tous les pointeurs enfants à `NULL` et `is_word` à 0.

```
TrieNode *trie_new(void) {
    TrieNode *node = calloc(1, sizeof(TrieNode));
    return node;
}
```

(b) Écrire une fonction `trie_insert` qui ajoute un mot (`a..z`) dans le trie.
Prototype :

```
int trie_insert(TrieNode *root, const char *word);
```

La fonction retourne 0 si succès, -1 en cas d'erreur d'allocation.

```c
int trie_insert(TrieNode *root, const char *word) {
    TrieNode *node = root;
    for (const char *p = word; *p; p++) {
        int idx = *p - 'a';
        if (idx < 0 || idx >= ALPHA) return -1;
        if (node->child[idx] == NULL) {
            node->child[idx] = trie_new();
            if (node->child[idx] == NULL) return -1;
        }
        node = node->child[idx];
    }
    node->is_word = 1;
    return 0;
}
```

(c) Écrire une fonction `trie_search` qui retourne 1 si le mot existe dans le trie, 0 sinon. Prototype :

```c
int trie_search(const TrieNode *root, const char *word);
```

```c
int trie_search(const TrieNode *root, const char *word) {
    const TrieNode *node = root;
    for (const char *p = word; *p; p++) {
        int idx = *p - 'a';
        if (idx < 0 || idx >= ALPHA) return 0;
        if (node->child[idx] == NULL) return 0;
        node = node->child[idx];
    }
    return node->is_word ? 1 : 0;
}
```

(d) Écrire une fonction `trie_starts_with` qui retourne 1 si un préfixe existe dans le trie, 0 sinon. Prototype :

```c
int trie_starts_with(const TrieNode *root, const char *prefix);
```

```c
int trie_starts_with(const TrieNode *root, const char *prefix) {
    const TrieNode *node = root;
    for (const char *p = prefix; *p; p++) {
        int idx = *p - 'a';
        if (idx < 0 || idx >= ALPHA) return 0;
        if (node->child[idx] == NULL) return 0;
        node = node->child[idx];
    }
    return 1;
}
```

Exercice 4 : Réflexion

(a) Expliquez un inconvénient mémoire des tries, et donnez une amélioration possible.

Les tries consomment beaucoup de mémoire (nombreux pointeurs `NULL`). Une amélioration est d'utiliser une table de hachage ou une liste chaînée pour les enfants présents seulement.