Arbres prefixes - trie.

Contrôle terminal de TP – durée 1h30

NOTES : éléments de notation

— la bonne utilisation du pouvoir expressif du langage *C* sera prise en compte,
— la présentation et la lisibilité du code écrit seront prises en compte.

1 Contexte

En informatique, un *trie* ou arbre préfixe, est une structure de données ayant la forme d'un arbre n-aire. Cette structure est utilisée pour stocker une table associative où les clés sont des chaînes de caractères. Le terme de *trie* vient de l'anglais re*trie*val, signifiant extraction, recherche. Contrairement à un arbre binaire de recherche, aucun nœud dans le *trie* ne stocke la clé à laquelle il est associé. C'est le chemin de la racine jusqu'à un nœud dans l'arbre qui détermine la clé correspondante.

Pour tout nœud n d'un *trie*, les clés de ses descendants ont en commun le même préfixe qui est la clé du nœud n. La racine est associée à une clé vide, représentant la chaine de caractères vide.

Les applications d'un *trie* sont nombreuses. Cette structure de données peut servir à implanter un tableau associatif, à trouver des redondances dans certains algorithmes de compression (par exemple dans les algorithmes de compression par dictionnaire à fenêtre glissante comme LZ77), à implanter des algorithmes de correction orthographique, de complétion automatique, de recherche préfixe, suffixe ou approximative ...

L'objectif de cet examen est d'implanter un *trie* en langage C pour la gestion d'un dictionnaire dont les clés sont des chaines de caractères représentant des mots. Sans perte de généralité, nous ferons abstraction des données associées aux nœuds et nous ne nous intéresserons qu'à la gestion des mots seuls.

À partir du code fourni dans l'archive `controleTP.tar.gz` associée à ce sujet, construite sur le même modèle que les archives de TP et proposant une représentation interne (ainsi que les opérateurs fondamentaux de gestion de cette représentation) des composants d'un *trie* ainsi qu'un TAD permettant de gerer un tableau dynamique de chaines de caractères, l'objectif de ce contrôle est de programmer l'opérateur d'ajout d'une clé dans un *trie*, la recherche de l'existence d'une clé dans un *trie* ainsi que la programmation de différents parcours du Trie pour fournir des fonctionnalités spécifiques.

1.1 Description du code fourni

Le code fourni contient :

— Un sous répertoire `Code` contenant :
  — un fichier Makefile permettant de compiler l'application et de produire les résultats attendus,
  — un fichier `main.c` proposant un programme de test de l'implantation,
  — un fichier `trie.h` définissant l'interface publique d'un *trie*,
  — un fichier `trie.c` définissant l'implantation et l'interface privée du TAD et à compléter pour ce contrôle,
  — les fichiers `stringarray.h` et `stringarray.c` définissant un module de gestion de tableau dynamique de chaines de caractères.
— Un sous répertoire `Test` contenant des fichiers de test.

Le code fourni ne devra en aucun cas être modifié. Seul devra être complété le fichier `trie.c` en dessous du cartouche à compléter par votre nom, prénom et numéro d'étudiant.

```
/*****************************************************************************/
/**                                Control start here                       **/
/*****************************************************************************/
/**
 *  Nom :                       Prenom :                    Num Etud  :
```

1.2 Description des algorithmes à mettre en place.

1.2.1 Structure de données

Sans perte de généralité, nous considérerons uniquement le stockage de mots consistués de lettres majuscule, de 'A' à 'Z'. Un *trie* est alors un arbre 26-aire. Un nœud n de l'arbre, à un niveau i donné, correspond à une lettre qui est en position i dans au moins un mot du dictionnaire. Les nœuds sont représentés par le TAD privé `Node`. Les liens entre les nœuds correspondent aux lettres des clés et sont stockées dans un tableau de pointeurs `Node* links[26]` de telle sorte que le lien correspondant à la lettre 'A' soit le premier élément de ce tableau, celui à la lettre 'B' le second et ainsi de suite. Le chemin qui relie la racine du *trie* au nœud n défini donc une chaine de caractère *préfixe* du nœud n. A partir du nœud n, on peut accéder, **pour chaque mot** partageant ce préfixe, à la lettre suivante dans le mot, au niveau $i+1$. Chaque nœud possède donc éventuellement plusieurs nœuds fils, un par mot issu de ce préfixe. Un *trie* est donc une structure de données récursive. A chaque nœud est aussi associé un indicateur booléen de fin de mot qui est `true` si ce nœud désigne la dernière lettre d'un mot, `false` sinon.

Prenons l'exemple de la figure 1. La racine possède deux fils non nuls, un pour la lettre 'E' et un pour la lettre 'P'. Cela signifie qu'il y a au moins un mot commençant par 'E' et un mot commençant par 'P' dans l'arbre. Cela signifie aussi qu'il n'y a aucun mot commençant par 'A' dans le *trie*. Suivons le chemin en suivant les liens 'E', puis 'P', puis 'I'. Le nœud désigné par ce chemin est marqué (grisé sur la figure) comme terminal. Cela veut dire que la chaîne de caractère 'EPI' est un mot du dictionnaire. Ce noeud a un fils sur le lien 'E' qui est aussi marqué comme terminal, le mot 'EPIE' fait donc aussi parti du dictionnaire.

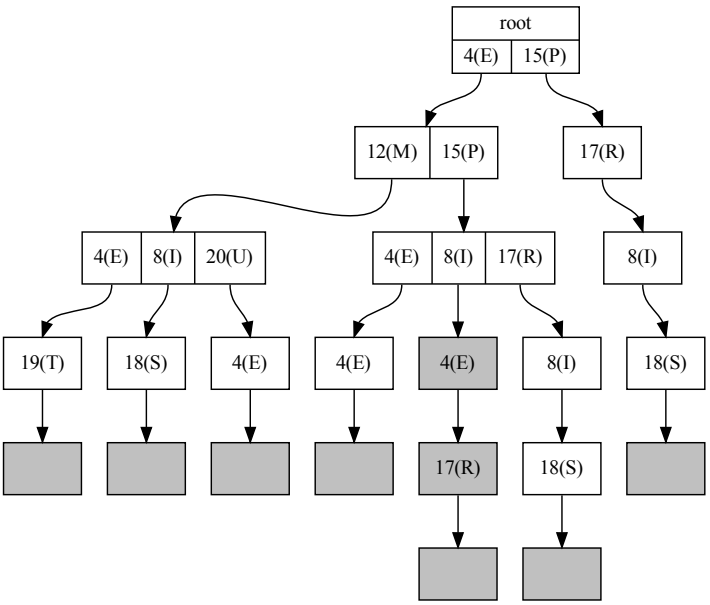

FIGURE 1 – *Trie* contenant les mots 'EMUE' 'EMIS' 'EPEE' 'EPIE' 'EMET' 'EPI' 'PRIS' 'EPIER' 'EPRIS'. Les noeud grisés sont les noeuds terminaux, c'est à dire correspondant à un mot du *trie*. La racine est indiqué *root*. Le contenu d'un nœud fait apparaitre l'indice et la lettre des liens vers ses fils s'il en a.

1.2.2 Ajout d'un élément au *trie*.

La fonction `void trie_add(Trie* t, const char* word)`, un constructeur du TAD, a pour objectif d'insérer dans le *trie* t un nœud dont la clé est définie par la valeur *word*.

L'insertion d'une valeur dans un *trie* correspond, comme pour les arbres de recherche implantés en TP, à une étape de recherche de la feuille sur laquelle insérer le nouveau nœud suivie d'une insertion du nouveau nœud. La difference est que dans un *trie*, insérer une nouvelle clé peut produire plusieurs nœuds, un par lettre de la clé. Ces deux étapes consistent en les opérations suivantes :

1. Parcours du trie en suivant les liens correspondant à chaque lettre de la clé tant qu'un tel lien existe. Lors de ce parcours, les lettres de la clé sont utilisées de la première à la p^{ieme} si il existe un chemin

dans le trie (une succession de liens) pour le préfixe composé des lettre 1 à p de la clé.

2. Création d'un nouveau chemin pour insérer le nœud terminal en établissant les liens correspondant aux lettres de la clé restantes à partir de la lettre $p+1$.

Une fois identifié le nœud n accessible après avoir *consommé* p lettres de la clé, un nouveau nœud est ajouté dans l'arbre, en tant que fils du nœud n pour le lien correspondant à la $(p+1)^{ieme}$ lettre de la clé. Puis un nœud fils est ajouté à ce nouveau nœud pour le lien correspondant à la $(p+2)^{ieme}$ lettre de la clé, et ainsi de suite, récursivement, jusqu'à épuisement des lettres de la clé. Le dernier nœud atteint, qu'il soit déjà présent dans le *trie* ou qu'il ait été créé lors de l'insertion de la clé est alors marqué comme terminal.

1.2.3 Parcours de l'ensemble des nœuds d'un *trie*.

Ce parcours récursif en profondeur d'abord, est similaire au parcours préfixe en profondeur d'abord étudié sur les arbres binaires de recherche, les appels récursifs se faisant pour tous les liens d'un nœud. La fonction de traitement associée au parcours est appelée sur tous les nœuds visités.

1.2.4 Recherche de l'existence d'un mot dans un *trie*.

La recherche d'un mot dans un *trie* consiste à suivre récursivement les liens correspondant aux lettres du mot recherché et à renvoyer vrai si le nœud atteint est terminal (le mot est présent dans le *trie*).

1.2.5 Parcours des mots d'un *trie*.

Lorsque l'on parcourt un *trie* selon ses mots, il faut parcourir, en profondeur d'abord et avec un traitement préfixe, tous les nœuds n du *trie* en reconstruisant la clé correspondant au chemin entre la racine de l'arbre et le nœud n. Si ce nœud est terminal, la fonction de traitement associée au parcours est appelée avec la clé reconstruite.

2 Travail à réaliser

2.1 Opérateur d'ajout d'une clé dans un *trie*

1. En utilisant les opérateurs définis sur le TAD `Node`, privé au module `Trie`, programmer, **de façon itérative et avec une complexité en temps égale à** $\Theta(n)$, ou n est le nombre de lettres de la clé, l'opérateur `void trie_add(Trie* t, const char* word)` qui ajoute la clé *word* dans le *trie* t.
2. Programmer, **de façon récursive**, l'opérateur `void trie_map(const Trie* t, TrieMapOperator f, void* userdata)` effectuant un parcours de l'ensemble des nœuds du *trie* t et appelant sur chaque nœud visité le foncteur `f` avec comme paramètres le nœud visité, le lien (une lettre) d'accès à ce nœud depuis son père et les données utilisateur `userData`.

Pour vérifier cette première question, vous pouvez compiler (make) et exécuter votre programme en lançant la commande `./trie_test ../Test/testfile_simple.txt`.

Le résultat attendu dans le terminal est le suivant :

```
$./trie_test ../Test/testfile_simple.txt
Building Trie ... EMUE, EMIS, EPEE, EPIE, EMET, EPI, PRIS, EPIER, EPRIS
Exporting the trie to ../Test/testfile_simple.dot ...  Done
...
```

Vous pouvez produire le fichier pdf de visualisation du *trie* que vous avez créé en tapant `make pdf`. Si vous ouvrez le fichier pdf `testfilesimple.dot.pdf` contenu dans le répertoire `../Test`, vous obtiendrez l'arbre de la figure 1.

2.2 Opérateur de recherche de l'existence d'un mot dans un *trie*.

Programmer **de façon itérative, en temps optimal** ($\Theta(n)$) **et espace minimal** ($\Theta(1)$), l'opérateur `bool trie_search(const Trie * t, const char *word)` qui cherche si la clé *word* est présente dans l'arbre *trie*.

Pour vérifier cette question, vous pouvez compiler (make) et exécuter votre programme en lançant la commande `./trie_test ../Test/testfile_simple.txt` . Le résultat attendu dans le terminal est le suivant :

```
$./trie_test ../Test/testfile_simple.txt
...
Searching words into the dictionary :
        EPI : true
        EPIER : true
        EPRISE : false
...
```

2.3 Opérateurs de parcours des mots d'un *trie*

Programmer, **de façon récursive** et avec un stockage minimum, l'opérateur `void trie_visit(const Trie* t, TrieVisitFunctor f, void* userdata)` effectuant un parcours des mots stockés dans le *trie*. Vous pouvez définir une fonction récursive intermédiaire `void trie_visit_helper(const Trie* t, const char* prefix, TrieVisitFunctor f, void* userdata)` qui effectue le parcours en reconstruisant la clé au cours du parcours et qui sera appelée par `trie_visit` avec la racine de l'arbre, un prefixe initial qui sera initialisé au mot vide (la chaine de caractères `""`), le foncteur et les données utilisateur. La fonction `trie_visit_helper` appelera le foncteur sur les nœuds terminaux avec la clé correspondante au nœud reconstruite pendant le parcours et passée lors des appels récursif dans le paramètre `prefix`. Cette reconstruction se fera en deux étapes :

— avant les appels récursifs, préparer le préfixe des appels par

```
size_t lp = strlen(prefix); // size of the prefix, without end of string marker '\0'
char newprefix[lp+2];       // lp + 2 ; size of prefix + new link + '\0'
strcpy(newprefix, prefix);  // copy old prefix into new prefix
newprefix[lp+1] = '\0';     // mark the end of the new prefix
```

— au moment de chaque appel récursif, finaliser le prefixe en positionnant sa dernière lettre par

```
...
newprefix[lp] = i; // the last letter of the new prefix is the followed link
...
```

où i est la lettre correspondant au lien sur lequel est fait l'appel récursif.

Pour vérifier cette question, vous pouvez compiler (make) et exécuter votre programme en lançant la commande `./trie_test ../Test/testfile_simple.txt`.

Le résultat attendu est le suivant :

```
$./trie_test ../Test/testfile_simple.txt
...
Words on trie :  EMET EMIS EMUE EPEE EPI EPIE EPIER EPRIS PRIS
...
```

2.4 Recherche de tous les mots ayant un préfixe commun.

1. Programmer la fonction `const Trie* trie_suffix(const Trie* t, const char* p)` qui renvoie la racine r du sous arbre pour lequel le chemin entre la racine du *trie* et r correspond au prefixe p. Si un tel nœud n'existe pas, cette fonction renvoie `NULL`.
2. En utilisant les fonctions `trie_suffix`, `trie_visit` ainsi que le module `StringArray`, et en definissant un foncteur et une structure passée à ce foncteur adaptés au traitement, programmer la fonction `StringArray* trie_get_completion(const Trie* t, const char* prefix)` qui renvoie un tableau de chaines de caractères (un `StringArray`) contenant tous les mots de l'arbre ayant comme préfixe $prefix$.

Pour vérifier cette question, vous pouvez compiler (make) et exécuter votre programme en lançant la commande `./trie_test ../Test/testfile_simple.txt`.

Le résultat attendu est le suivant :

```
$./trie_test ../Test/testfile_simple.txt
...
Finding words by prefix
        Prefix EPI : EPI, EPIE, EPIER
        Prefix EM : EMET, EMIS, EMUE
```