

1 Programmation modulaire

Le langage de programmation utilisé durant l'unité d'enseignement « Algorithmique du texte » est le C. Afin de concevoir du code réutilisable, on privilégie la programmation modulaire. Donc à une sous-tâche identifiée correspond un `.c` contenant les fonctions et un `.h` contenant la déclaration des structures de données et les prototypes des fonctions.

Tous les fichiers pourront être compilés à l'aide d'un `makefile`.

2 Rappels sur les makefile

Syntaxe

```
cible : liste de dépendances  
(tab) action
```

Utilisation de variables

```
ID_VAR = valeur  
$(ID_VAR)
```

3 Trie

Un trie (terme provenant de *information retrieval*) est un arbre utilisé pour stocker des mots d'un ensemble. Chaque branche d'un trie est étiquetée par une lettre. Chaque nœud q d'un trie est associé avec le mot unique constitué par la concaténation des étiquettes des branches de la racine à q . La racine est associée au mot vide. Les préfixes communs des mots de l'ensemble sont donc factorisés. Un trie peut être considéré comme un Automate Fini Déterministe (AFD) ce qui implique que les branches issues d'un même nœud sont étiquetées par des lettres différentes.

Il existe plusieurs méthodes pour représenter les tries. On en considère trois :

- une matrice de transitions ;
- un tableau de listes d'adjacence ;
- mixte (une table de transitions pour la racine et un tableau de listes d'adjacence pour les autres nœuds).

Exercice 1. Pour les trois méthodes mentionnées ci-dessus, utiliser les structures de données suivantes pour représenter un trie :

Matrice de transitions :

```
struct _trie {
    int maxNode;      /* Nombre maximal de noeuds du trie */
    int nextNode;     /* Indice du prochain noeud disponible */
    int **transition; /* matrice de transition */
    char *finite;     /* etats terminaux */
};
```

Listes d'adjacence :

```
struct _list {
    int targetNode; /* cible de la transition */
    unsigned char letter; /* etiquette de la transition */
    struct _list *next; /* maillon suivant */
};
```

```
typedef struct _list *List;
```

```
struct _trie {
    int maxNode;      /* Nombre maximal de noeuds du trie */
    int nextNode;     /* Indice du prochain noeud disponible */
    List *transition; /* listes d'adjacence */
    char *finite;     /* etats terminaux */
};
```

Mixte :

```
struct _trie {
    int maxNode;      /* Nombre maximal de noeuds du trie */
    int nextNode;     /* Indice du prochain noeud disponible */
    int *transitionRoot; /* transitions de la racine */
    List *transitionOthers; /* transitions des autres noeuds */
    char *finite;     /* etats terminaux */
};
```

Pour tous :

```
typedef struct _trie *Trie;
```

Écrire les primitives permettant :

- de créer un trie (cette primitive prend en entrée le nombre maximal de noeuds du trie);
Trie createTrie(int maxNode);
- d'insérer un mot dans un trie;
void insertInTrie(Trie trie, unsigned char *w);
- tester si un mot est contenu dans un trie.
int isInTrie(Trie trie, unsigned char *w);

On ne représente pas l'alphabet, on prend en compte l'ensemble des octets possibles à l'aide du type `unsigned char`.

Exercice 2. Utiliser les primitives précédentes pour construire un trie pour :

- les préfixes d'un mot;
- les suffixes d'un mot;
- les facteurs d'un mot (aide : insérer successivement tous les suffixes du mot du plus long au plus court).