GRENOBLE INP UGA

Décembre 2021

TP : vérificateur orthographique

Présenté par

JOLY Léo

NAJDA Mohamad

Professeurs

Mr. DESVIGNES

Mr. COLOMBIER

Méthode de Lecture

Pour la lecture des fichiers, nous avons développé des fonctions dans les fichiers lecture.c et lecture.h qui permettent d'effectuer la lecture des fichiers et le chargement des éléments directement dans une structure de donnée quelconque passé en paramètre.

Il est important de noter que nous avons décidé de ne pas vérifier les mots commençant par une majuscule (éviter de vérifier les noms propres) et les chiffres ce qui enlève une bonne partie des mots. Il faudrait vérifier si le mot n'est pas le début d'une phrase et n'est pas un nom propre pour être plus précis. (néanmoins cela peut facilement être modifié dans ***lecture.c***)

Voici les prototypes des fonctions utilisées :

```
//Lis le fichier et découpe chaque ligne en tableau de mots qui sont ensuite chargés
dans la structure de donnée en paramètre.
int lecture (FILE* texte, void* struct_donne, bool (func)(char*, void*), int nb_max ,
int *total);

//Fonction optimisée pour la lecture du dictionnaire (1 mot par ligne)
void lecture_dico (FILE* texte, void* struct_donne, void (func)(char*, void*));

//Libère la mémoire d'un tableau de char de nb éléments
void free_tab_char (char** c, int nb);

//Transforme la prochaine ligne de pf en tableau de char mots ET alloue la mémoire
char** get_next_line_into_words (FILE* pf, int* nb_word);
//Compte le nombre de mots dans la phrase
int words_count (char* phrase);
```

Éléments

Afin de faciliter le travail dans notre code, nous avons implémenté la structure element (***elem***) qui est composée d'un mot (char*), et pour lequel on a implémenté des fonctions qui nous ont beaucoup facilité le travail et la visibilité du code par la suite. De plus on notera que cette structure elem est modulable si elle doit être modifiée par la suite.

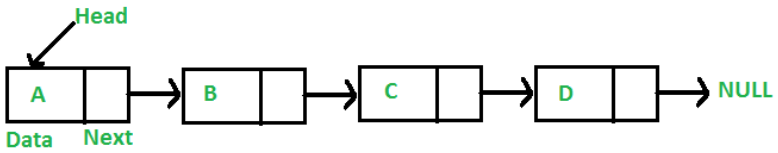
```
//Structure d'un élément (ici juste un mot)
typedef struct mot {
  char* mot;
} *elem;

//Crée un nouvel élément
elem element_new(char* mot);
//Compare deux éléments, utilise strcmp donc 0 = équivalent
int element_compare(void* e1, void* e2);
//Affiche un élément
void element_print(elem e);
//Donne la lettre a l'indice i de l'élément
```

```c
char element_get(elem e, int i);
//Supprime l'élément
void element_delete(void* e);
//Donne la taille du mot de l'élément
int element_length(elem e);
```

Structures de données

I. Liste Chaînée



Présentation

Une liste chaînée est une collection linéaire d'éléments de données dont l'ordre n'est pas donné par leur placement physique en mémoire. Au lieu de cela, chaque élément pointe vers le suivant. Il s'agit d'une structure de données constituée d'un ensemble de nœuds qui représentent ensemble une séquence.

Implementation

Pour implémenter notre liste chaînée, nous avons utilisés des fonctions qui ne dépendent pas d'éléments mais bien d'un pointeur quelconque ***void**** ce qui permet la portabilité du code. Pour qu'il soit fonctionnel, il faut aussi donner en paramètre les fonctions spécifiques qui dépendent de l'objet a stocker dans la structure.

implémenter d'une manière très similaire à notre liste chaînée du TD tout en remplaçant nos valeurs (int) par des éléments elem.

```c
typedef struct node{
  void* e ;         //pointer vers notre element "elem"
  struct node* next; //pointer vers notre prochain node dans la liste
}*liste;

// retourne un NULL , sert à l'initialisation de notre liste
liste liste_create();
//affichage de notre liste
void liste_afficher(liste l );
//Fonction pour vérifier l'existence d'un élément spécifique dans notre liste
void* liste_element_exist(void* e, liste l, int (cmp_func) (void*, void*));
//Fonction qui retourne l'élément d'indice donné en argument
void* liste_get_element(liste l, int indice);
```

```
//Fonction qui permet l'ajout d'un élément à la fin de notre liste (pas utiliser dans
notre test finale car complexité O(n))
void liste_add_last(void* e , liste* l);
//Fonction qui permet l'ajout d'un élément au début de notre liste (complexité O(1))
void liste_add_first(void* e, liste* l);
//fonction qui permet la destruction de notre liste avec un free de tous les pointeurs
utilisés
void liste_destroy(liste l, void (del_func)(void*));

///Fonction utilisée dans notre test final qui vérifie l'existence d'un élément en
retournant un bool et l'efface après
bool verifListe(char* mot, void* struct_donne);
///Fonction utilisée dans notre test final afin de former la liste des mots du
dictionnaire
void lectureListe(char* mot, void* struct_donne);
```

Complexité

Phase de construction :

Lors de la création de notre liste et de son remplissage avec les mots du dictionnaire, la complexité est de O (1) car la fonction d'ajout en tête de liste ne fait que créer un nouveau nœud et fait le lien avec l'ancien premier de la liste. Il n'y a pas de parcours de liste ici donc pas de complexité.

Phase de vérification :

La complexité de la liste chaînée est linéaire : O(n), avec un n qui vaut jusqu'à 323 925 dans notre cas ce qui signifie que la liste chaînée est une très mauvaise solution pour la vérification orthographique. En effet, on attend près de 40 minutes pour vérifier le texte final en entier.

Mémoire

Notre liste chaînée contient deux informations principales (la valeur → elem et le pointeur) par nœud. Cela signifie que la quantité de données stockées augmente linéairement avec le nombre de nœuds dans la liste.

Chaque nœud ou chaînon doit contenir l'adresse du mot contenu (8 octets + 8 octets par lettre) et l'adresse du chaînon suivant (8 octets). Soit si on prend en moyenne des mots de 4.8 lettres, un chaînon = 46.4 octets en moyenne.

II. Hash Table

Présentation

La table de Hachage est une structure linéaire qui s'apparente à un tableau où chacune des cases contient une liste chainée d'éléments. Pour trouver la bonne case du tableau, on calcule un Hash (la clé) qui est un entier calculé à partir de l'élément. Cette clé est modulo la taille de la table de hachage et varie de manière conséquente entre deux éléments même très similaires.

Implementation

Pour implémenter cette table de hachage, nous avons utilisés des fonctions qui ne dépendent pas d'éléments mais bien d'un pointeur quelconque ***void**** ce qui permet la portabilité du code. Pour qu'il soit fonctionnel, il faut aussi donner en paramètre les fonctions spécifiques qui dépendent de l'objet a stocker dans la structure.

```
typedef struct {
  liste* table;
  unsigned capacite;         /* capacité de la table */
  unsigned nb_elements;      /* nombres d'éléments dans la table */
  unsigned capacite_initiale; /* utile lors du redimensionnement */
} table_hachage;

//Crée une nouvelle table
table_hachage hash_new(unsigned capacite);
//Fonction de hachage pour un elem
int hash_str(void* element, unsigned capacite);

//Vérifie si l'élément est déjà présent dans la table
void* hash_est_present(void* element, table_hachage* ht, int (hash_func)(void*, unsigned
int),int (cmp_func) (void*, void*));
//ajoute un élément sans redimensionner la table
void hash_inserer_sans_redimensionner(void* element, table_hachage* ht, int
(hash_func)(void*, unsigned int),int (cmp_func) (void*, void*));
//Affiche toute la table
void hash_afficher_table(table_hachage* ht);
//Détruit la table et ses éléments en utilisant delete_func
void hash_destroy(table_hachage* ht, void (delete_func)(void*));
//Ajoute un élément en redimensionnant si besoin (nb elements >= capacité)
void hash_inserer_redimensionner(void* element, table_hachage* ht, int
(hash_func)(void*, unsigned int),int (cmp_func) (void*, void*));

//Fonction vide pour supprimer la table sans supprimer ses éléments
void empty(void* e);

//Fonctions utiles pour le test final
void lectureHash(char* mot, void* struct_donne);
bool verifHash(char* mot, void* struct_donne);
```

Complexité

La complexité de la table dépend de la taille de celle-ci et de la qualité et complexité de la fonction de hachage. Nous avons ici utilisé une table redimensionnable pour éviter de la surcharger et d'avoir à prévoir à l'avance la taille de la table. Néanmoins ce n'est pas gratuit, lorsque le nombre d'éléments dans la table atteint le nombre de case, nous doublons le nombre de case en créant une nouvelle table de hachage. Il faut alors recalculer le hash de tous les anciens éléments dans la nouvelle donnant une complexité en O(n) + complexité de hash. Néanmoins, le redimensionnement étant peu fréquent , nous allons théoriquement le négliger.

Phase de construction :

On peut considérer la table de hachage comme un tableau, l'accès à une case mémoire est donc de O(1). A cela on ajoute la complexité de la fonction de hachage pour trouver la case, celle-ci somme et multiplie par une constante les éléments de la chaîne de caractère donnée. La taille moyenne d'un mot en français étant de 4,8 caractères, on peut considérer son coût constant et raisonnablement petit.

Etant donné le nombre de cases face au nombre d'éléments, on peut négliger le parcours de la liste dans la case donnée car celle-ci devrait être très petite dans tous les cas (sauf en cas de mauvaise fonction de hachage).

Phase de vérification :

La complexité de cette phase est exactement la même que lors de la phase d'ajout d'éléments sans le redimensionnement. On est donc proche de O(1) + complexité de hash. Cette structure nous donne alors de très bon résultats en termes de complexité et de temps de recherche.

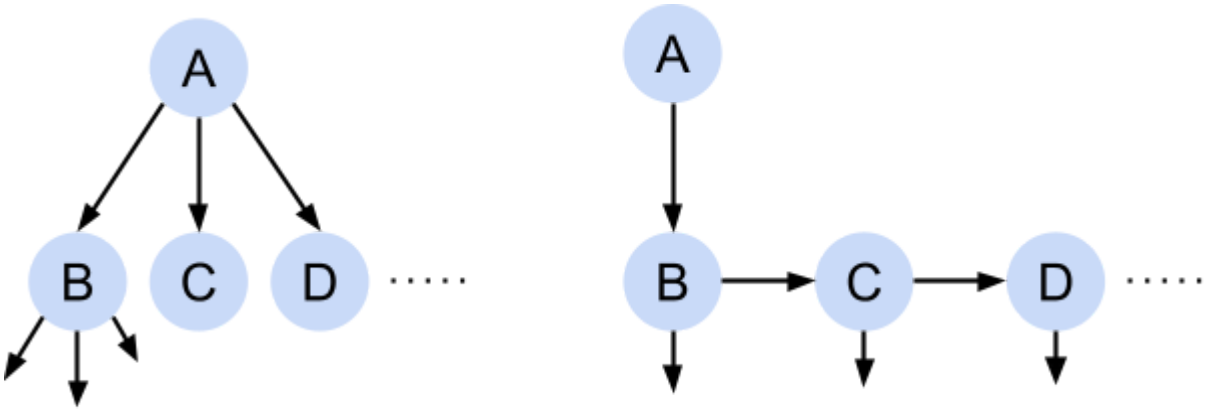
Mémoire

Pour la mémoire, chaque élément se voit attribuer un nœud d'une liste comme pour la liste chaînée. Néanmoins, chaque case vide ou non de la table de hachage se doit d'avoir une liste chaînée allouée dans la mémoire ce qui est très gourmand. C'est le prix pour avoir une complexité si faible. On a donc en plus des 50 octets par cases occupés, 8 octets pour chaque cases vides de la table de hachage. On peut estimer cette perte a ¼ de notre table en moyenne.

III. Arbre Préfixe

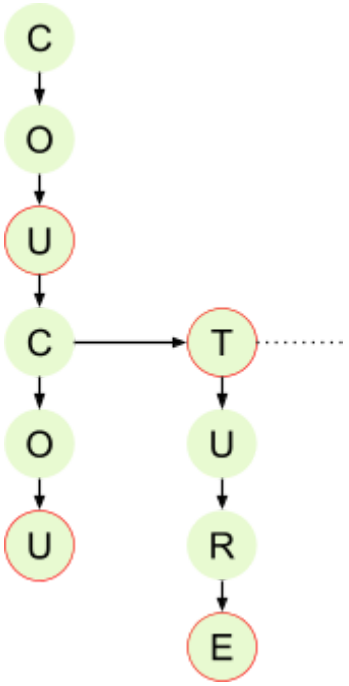
Présentation

Pour les structures d'arbres, nous avons décidé d'utiliser des structures a base d'arbre binaire et non d'arbre n-aire (avec une liste de n fils). Ce choix a été fait par simplicité d'implémentation et pour optimiser la mémoire en évitant d'ajouter des listes à la structure d'arbre. De plus, les deux structures peuvent d'identifier de la même manière avec le système de frère et fils :



Arbre n-aire Arbre binaire

En se basant sur l'arbre binaire de recherche, nous avons implémenté notre arbre de préfixe en triant par ordre alphabétique nos nœuds. Cette ordonnance permet notamment d'accélérer l'ajout dans l'arbre et sera utile lors de la compression en radix. Voici un exemple sommaire de notre arbre préfixe :



Les mots insérés dans notre arbre sont :

- cou
- coucou
- cout
- couture

Résultat parcours préfixe (fils puis frères)

c→o→u→c→o→u→t →u→r→e

Implementation

```
typedef struct noeud
{
    elem val;
    struct noeud* frere;
    struct noeud* fils;
    bool final;   //afin d'indiquer si un etat est final
    bool reloc;  //Marqueur pour la compression (relocation)
}noeud;
typedef noeud* arbre;

//initialiser la racine de notre arbre
noeud* creer_noeud(elem valeur);
//Fonction qui indique si notre arbre est vide
bool arbre_est_vide(arbre a);
//Fonction qui permet la destruction de notre arbre avec un free de tous les pointeurs
utilisés
void detruire_arbre(arbre a);
//foncrion pour indiquer le nombre d'etage de notre arbre
unsigned hauteur(arbre a);
//fonction  pour indiquer le nombre de noeuds dans notre arbre
unsigned nb_noeuds(arbre a);
//fonction  pour indiquer le nombre de feuilles dans notre arbre
unsigned nb_feuilles(arbre a);
//affichage de notre arbre
void parcours_prefixe(arbre a);
//comparation de 2 arbres données en parametre
int noeud_cmp(void* a1, void* a2);
```

D'un arbre binaire générique à un arbre préfixe :

```
//fonction qui sert à chercher un élément spécifié en paramètre dans l'arbre préfixe
bool recherche_arbre_prefix(arbre a , elem e);
//fonction qui sert à inserer un élément spécifié en paramètre dans l'arbre préfixe et
selon son ordre alphabetique
void ajout_prefix(arbre* a, elem e);
//FONCTIONS POUR LE TEST FINAL//
//fonction utilisée dans notre Test final qui vérifie l'existence d'un mot du text dans
le dictionnaire
bool verifArbre(char* mot, void* struct_donne);
//fonction utilisée dans notre Test final afin de construire notre arbre préfixe
void lectureArbre(char* mot, void* struct_donne);
```

La fonction `void ajout_prefix(arbre* a, elem e)` est divisée en deux parties :

- La 1er partie ou on cherche s' il existe déjà le préfixe ou une partie du mot qu'on veut insérer. Si c'est le cas, on parcourt l'arbre jusqu'à la fin des similitudes puis on passe à la 2eme partie, si il n'y a pas de similitude on passe directement à la 2eme partie.
- La 2eme partie commence par la création d'un frère ou d'un fils, dépend du nœud précédent où on est arrivé dans la première partie puis on insère le reste des lettres de notre mot comme fils (on rappelle que le préfixe ajoute les lettres une par une). Une fois terminée on met à jour le bool ***final*** de notre dernier caractère/nœud.

La fonction `bool recherche_arbre_prefix(arbre a , elem e)` part du même concept de la première partie de la fonction d'ajout d'élément et vérifie si le mot nœud terminant la recherche existe dans l'arbre et est ***final***.

Complexité

Phase de construction :

La complexité de la création d'un trie est O(N*L) , où N est le nombre de mots, et L est une longueur moyenne du mot, on doit effectuer L recherches sur la moyenne pour chacun des W mots de l'ensemble

Phase de recherche :

La complexité de vérification d'un mot se fait alors en **O(log(n)),** avec n difficile à déterminer théoriquement puisqu'il s'agit de toutes les lettres de tous les mots - les lettres partagées.

Mémoire

Pour la mémoire, on a pour chaque noeud :

2 booléens (1 octet chacuns)

2 pointeurs (8 octets chacuns)

1 élément (8 octets + 8 octets pour la lettre)

Donc 34 octets par nœud et pour 323925 mots avec 4.8 lettres en Moyenne : 5,3

Mégaoctets En Moyenne.

IV. Arbre Radix

Présentation

L'arbre radix est par définition très proche de l'arbre préfixe mais il contient plusieurs niveaux de compressions pour la mémoire. Le premier niveau de compression que nous avons implémenté consiste à rassembler les nœuds fils qui n'ont pas de frère et qui ne sont pas finaux (c'est ce qu'on appellera Radix compressé 1 par la suite).

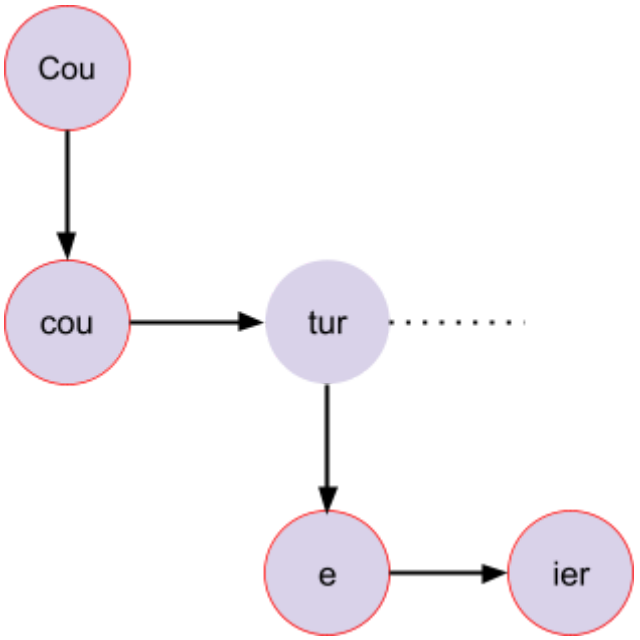
Le deuxième niveau de compression consiste à identifier des rémanences dans les suffixes et à supprimer les nœuds en double. Pour cela nous identifions les groupes identiques de suffixe pour les rassembler en libérant de la mémoire les doublons. Cette méthode est très efficace mais notre choix d'implémenter un arbre binaire a limité certaines compressions (c'est donc le Radix compressé 2).

Implementation

Radix compression 1

```
void transform_prefix_into_radix(arbre* a);
```

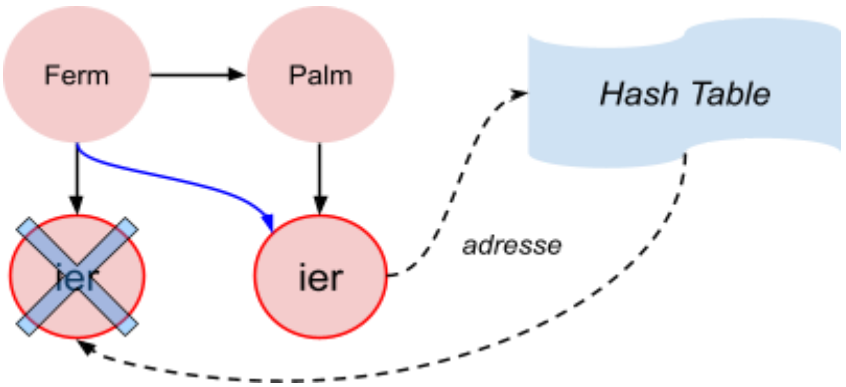
Une simple fonction de compression qui parcourt tous les nœuds des feuilles à la source en compressant si on remplit les conditions établies plus haut.



Radix compression 2

Tout d'abord, le tri alphabétique effectué dans l'arbre préfixe sert à faciliter la détection des paternes, en effet cela permet de supprimer les cas où les arbres seraient les mêmes mais avec des ordre de frère différents.

De plus, nous avons utilisé une table de hachage de parternes déjà identifié pour traiter rapidement les paires en utilisant comme clé la valeur du nœud. Si il y a un nœud équivalent, on vérifie si le fils et le frère de ces deux nœuds sont identiques et donc si la compression est possible. On remarque qu'il ne serait pas indispensable de vérifier le frère dans le cas d'un arbre n-aire mais il n'y a pas le choix pour l'arbre binaire. (En effet, si on déplace le nœud sans les frères, il faut rattacher le frère au doublon en mémoire mais on crée alors de nouveaux mots qui ne sont pas dans le dictionnaire) mais ça ne pose pas de gros problème de compression puisque cela n'impacte qu'un nœud.



On peut libérer la liste de hachage des paternes rencontrés lorsque l'arbre opti est créé car chaque nœud dupliqué est marqué par le **bool *reloc.*** Ce marquage est particulièrement utile lors de la libération mémoire car il permet de ne pas supprimer deux fois les mêmes nœuds et avoir cette information pour un moindre coût dans le cas d'un portage de la structure de donnée.

Nous aurions pu procéder différemment pour la libération de la mémoire (parcourir l'arbre et identifier les doublons) mais cette solution semblait la plus simple à mettre en œuvre. On libère donc tous les nœuds non marqués puis les nœuds marqués qui ont été ajoutés à une liste (ici table de hash pour la rapidité).

Complexité

La complexité de création de l'arbre est en **O(n²)** car nous avons séparé les deux fonctions de compression mais il est théoriquement possible de le faire en O(n). Dans tous les cas, le temps de construction de l'arbre devrait se voir négligeable face au temps de vérification puisqu'il devrait y avoir beaucoup de vérifications et 1 construction d'arbre compressé.

La complexité de vérification d'un mot se fait alors en **O(log(n))** avec un n nettement moins élevé que pour l'arbre radix. Néanmoins, la vérification de la similitude du nœud utilise ***strcmp()*** qui compare caractère par caractère, ce qui augmente considérablement la complexité et s'apparente au final à un arbre préfixe (revient a comparer tous les nœuds d'un préfixe).

Mémoire

Pour la mémoire, la compression fait des merveilles et notamment car la nature des mots français eux même sont constitués de préfix et suffixe partagés de base. On peut alors économiser plus de 90% de la mémoire par rapport à l'arbre préfixe.

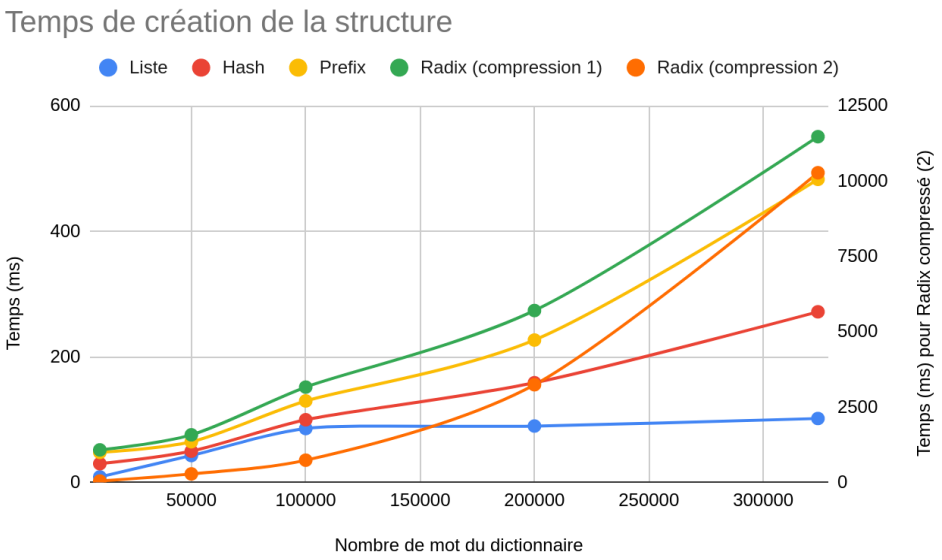
En effet, on gagne déjà de la mémoire en plaçant plusieurs caractères dans le même mot et en dédoublant de nombreux nœuds en double.

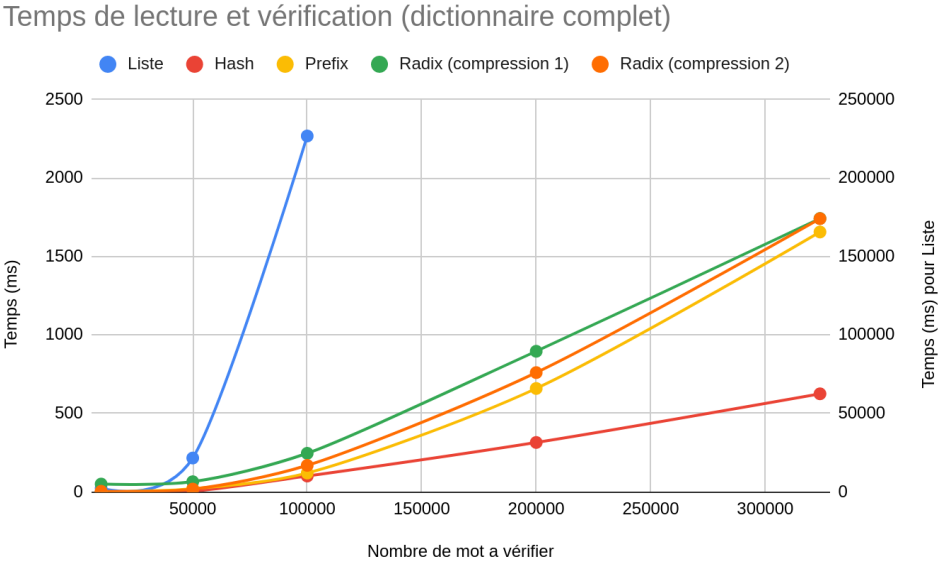
Comparaisons

Complexité / Rapidité

Nous avons utilisé la bibliothèque Time.h pour déterminer la rapidité des fonctions de création de la structure et de recherche d'un mot dans celle-ci. Les résultats sont donnés en ms et les courbes hors compétition (comme le Radix compression 2 ci-dessous) sont placées sur un axe droit différent.

Attention : Tous les résultats d'ordre temporel dépendent énormément de la machine utilisée et peuvent varier énormément entre deux tests consécutifs. Ils sont donc à prendre en compte comme une vue d'ensemble de l'évolution et de la comparaison des complexité entre les structures.

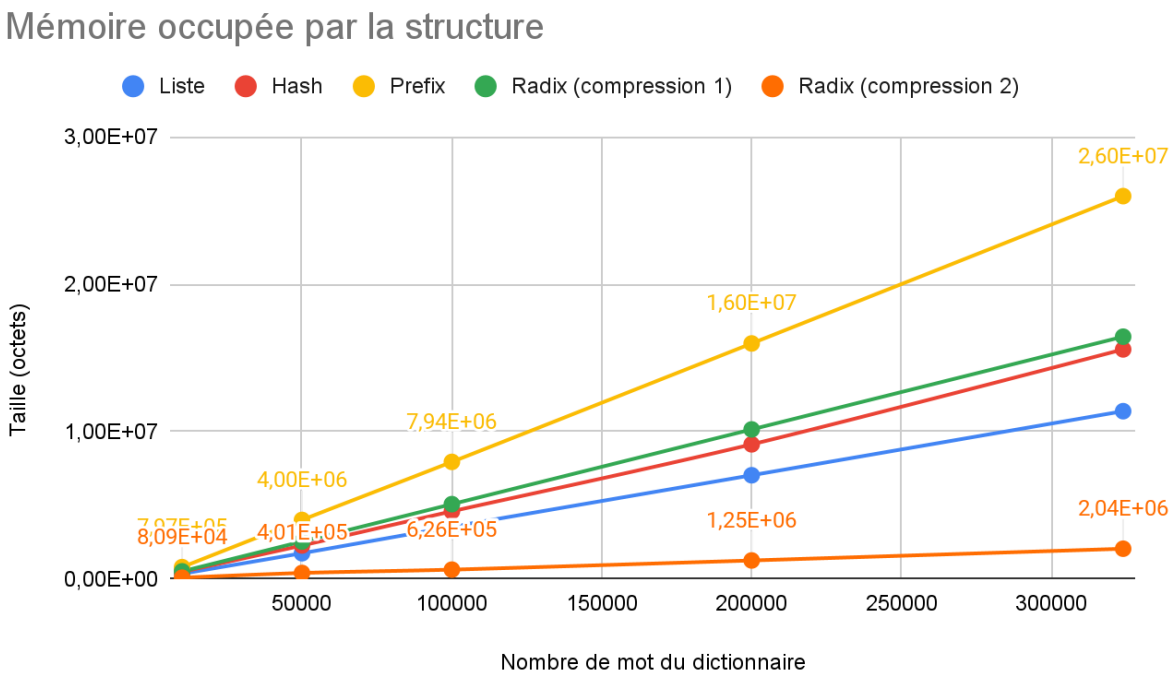




On remarque que pour la création de la structure, l'arbre radix explose le temps de toutes les autres structures, néanmoins ce n'est pas un problème pour notre application puisqu'il s'agit de créer un modèle puis de le réutiliser sans avoir à le recréer, c'est donc un temps normalement négligeable.

Pour le temps de lecture, on voit que la liste est hors jeu et donc une très mauvaise structure pour notre application. On remarque que les 3 arbres sont environ dans les mêmes ordres de grandeur et que la hash table est bien plus rapide et a une pente moins rapide. C'est donc la hash table qui serait la plus intéressante à utiliser si on veut favoriser la rapidité.

Mémoire



Semble très linéaire pour la plupart des structures ce qui fait sens pour les linéaires. Pour les arbres on peut prétendre que la structure de la langue française en général fait que la mémoire évolue de manière linéaire en ajoutant plus de mots.

On obtient 92% d'optimisation entre arbre Préfix et Radix : ceci peut encore être amélioré légèrement en construisant un arbre n-aire. De plus, l'arbre Radix bat toutes les autres structures et cela même avec un nombre réduit de mots. En effet, on gagne près de 80% de mémoire face à la table de hachage ce qui rend donc la structure de donnée très portable et il est alors acceptable de corriger plusieurs langues latines sur un même petit appareil avec peu de mémoire.

Difficultées

- Modification des listes et hash table pour pouvoir prendre n'importe quel adresse (**void ***) en tant qu' élément.
- Libération de la mémoire et vérification de la possibilité de fusion dans l'arbre radix. L'arbre Radix en général.

Voies d'améliorations

- Lecture du fichier à vérifier : on prend un Buffer très large car on ne sait pas combien de mots seront présents dans la ligne : faire en dynamique en prenant mots par mots.
- Homogénéiser toutes les fonctions dans les éléments et structures pour être utilisable par n'importe quel élément (**void***)
- Modifier l'arbre binaire pour un arbre n-aire pour optimiser encore plus la mémoire, ne pas utiliser de marqueur ***reloc*** pour gagner encore de la mémoire mais vérifier pendant la suppression que le nœud n'est pas déjà passé.
- …

Conclusion

En conclusion, si l'espace de stockage n'est pas un problème, la table de hachage est certainement la solution la plus simple et efficace à mettre en œuvre ici. Néanmoins, dans ce type de problème, la structure est certainement amenée à être transférée entre de nombreux utilisateurs ou logiciels afin de vérifier les mots sur plusieurs supports. La mémoire prise par celle-ci n'est donc certainement pas négligeable et se doit d'être la moins coûteuse. C'est donc l'arbre Radix qui nous permet non seulement d'avoir une taille mémoire nettement inférieure aux autres structures mais en plus d'avoir une rapidité de vérification acceptable en O(log(n)).

Merci d'avoir lu ce rapport jusqu'à la fin !