Dictionnaire - Correction

Samy Jaziri

1 Introduction

Préliminaires

```
typedef unsigned int uint;
typedef struct mot {
    char* chaine;
    uint taille;
} mot;
```

```
bool distinguer(mot* m1, mot* m2) {
    return *(m1->chaine) < *(m2->chaine);
}
```

```
mot* prefixer(mot* m, mot* prefixe) {
    uint i;
    mot* m_pre = (mot*) malloc(sizeof(mot));
    m_pre->taille = m->taille + prefixe->taille;
    m_pre->chaine = (char*) malloc(sizeof(char) * ( m_pre->taille + 1 )) ;
    m_pre->chaine[m_pre->taille] = '\0';
    for (i = 0 ; i < prefixe->taille ;++i)
        m_pre->chaine[i] = prefixe->chaine[i];
    for (i = 0 ; i < m->taille ;++i)
        m_pre->chaine[prefixe->taille + i] = m->chaine[i];
    return m_pre;
}
```

```
mot* prefixe(mot* m, uint pos) {
    mot* m_pre = (mot*) malloc(sizeof(mot));
    m_pre->taille = pos;
    m_pre->chaine = (char*) malloc(sizeof(char) * ( m_pre->taille + 1)) ;
    m_pre->chaine[m_pre->taille] = '\0';
    for (uint i = 0 ; i < m_pre->taille ;++i)
        m_pre->chaine[i] = m->chaine[i];
    return m_pre;
}
```

```
mot* suffixe(mot* m, uint pos) {
    mot* m_suf = (mot*) malloc(sizeof(mot));
    m_suf->taille = m->taille - pos;
    m_suf->chaine = (char*) malloc(sizeof(char) * ( m_suf->taille + 1)) ;
    m_suf->chaine[m_suf->taille] = '\0';
    for (uint i = 0 ; i < m_suf->taille ;++i)
        m_suf->chaine[i] = m->chaine[pos + i];
    return m_suf;
}
```

```
mot* copie(mot* m) {
    return suffixe(m, 0);
}
```

2 Arbres PATRICIA

```
typedef struct noeud noeud;

typedef struct branche {
    mot* etiquette;
    noeud* noeud_fils;
    struct branche* suivant;
} branche;

struct noeud {
    char* contenu;
    branche* liste_fils;
};

typedef struct dictionnaire {
    noeud* racine;
} dictionnaire;
```

```
dictionnaire* creer_dictionnaire_vide() {
    dictionnaire* dict = (dictionnaire*) malloc(sizeof(dictionnaire));
    dict->racine = (noeud*) malloc(sizeof(noeud));
    dict->racine->contenu = NULL;
    dict->racine->liste_fils = NULL;
    return dict;
}
```

3 Recherche dans un dictionnaire

On remarque que deux fils d'un nœud ne peuvent pas débuter par la même mettre, par définition d'un arbre PATRICIA. On cherche donc à chaque étape un fils qui commence par le même caractère que la clé. On vérifie que l'arête est étiqueté par un préfixe de la clé, auquel cas on suit cette arête et on cherche récursivement avec pour clé, le suffixe restant après avoir éliminé le préfixe trouvé de la clé.

```
char* recherche(noeud* n, mot* cle, char* defaut) {
    if(cle->taille == 0) {
        if(n->contenu)
            return n->contenu;
        return defaut;
    }

    for (branche* b = n->liste_fils;
         b;
         b = b->suivant ) {
        if(!distinguer(b->etiquette, cle)) {
            if (cle->taille < b->etiquette->taille)
                return defaut;
            for (uint i = 0; i < b->etiquette->taille; ++i)
                if(b->etiquette->chaine[i] != cle->chaine[i])
                    return defaut;
            mot* suffixe = (mot*) malloc(sizeof(mot));
            suffixe->taille = cle->taille - b->etiquette->taille;
            suffixe->chaine = (cle->chaine + b->etiquette->taille);
            char* valeur = recherche(b->noeud_fils, suffixe, defaut);
            free(suffixe);
            return valeur;
        }
    }
    return defaut;
}
```

```
char* get(dictionnaire* dict, mot* cle, char* defaut) {
    return recherche(dict->racine, cle, defaut);
}
```

On s'intéresse à la complexité de la recherche au pire cas. On remarque que le nombre de fils d'un noeud est borné par 26. En effet si le nombre de fils dépasse 26, alors deux arêtes sont étiquetées des mots dont les premières lettres sont identiques, ce qui contredit la définition des arbres `PATRICIA`. Ainsi chaque appel de recherche fait un nombre borné de tour de boucle. La fonction `distinguer` est en $\mathcal{O}(1)$. Chaque appel récursif descend d'un niveau. Une exécution de la fonction suit donc un chemin de l'arbre. Au passage de chaque nœud de l'arbre on réalise donc $\mathcal{O}(1)$ opération plus le coût de la vérification que l'étiquette de l'arête est un préfixe de la clé qui fait autant de comparaison que la taille de l'étiquette au pire. La complexité finale d'une telle exécution est donc un $\mathcal{O}(h)$ avec h la taille de chemin auxquel s'ajoute la somme de la taille de toutes les étiquettes du chemin, *i.e* la longueur du mot que défini ce chemin. La taille du chemin est elle même bornée par ce mot. La complexité est donc en $\mathcal{O}(|\gamma|)$ avec γ le mot représenté par le chemin de l'exécution. La complexité globale de l'algorithme est au final en $\mathcal{O}(|\Gamma|)$ avec Γ le plus long mot du dictionnaire.

4 Insertion dans un dictionnaire

- Pour ajouter `facteur` il faut rajouter un fils au nœud dont la clé est `fac`. La branche sera étiquetée `teur`.
- Pour ajouter `fac` il suffit de transformer le nœud intermédiaire déjà existant dont la clé associé est `fac` en un nœud terminal qui contiendra la valeur.
- Pour ajouter `lampe` il faut créer un fils au nœud dont la clé est `la`. La branche sera étiquetée `mpe`.
- Pour ajouter `lame` il faut supprimer la branche que l'on vient de créer pour `lampe`, étiquetée `mpe`. Puis il faut créer un nœud non-terminal, fils du noeud dont la clé est `la` et dont la branche sera étiquetée `m`. Ce nœud non-terminal aura donc deux fils terminaux. L'ancien fils supprimé dont la branche sera étiquetée `pe` et un nouveau fils dont la branche sera étiquetée `e`.

```
void insertion(noeud* n, mot* cle, char* valeur) {
    if(cle->taille == 0) {
        if(n->contenu)
            free(n->contenu);
        n->contenu = valeur;
        return;
    }
    branche** pointeur_precedent = &(n->liste_fils);
    branche* b;
    for (b = n->liste_fils;
         b;
         b = b->suivant ) {
        if(!distinguer(b->etiquette, cle)) {
            if (distinguer(cle,b->etiquette))
                break;
            for (uint i = 1; i < b->etiquette->taille; ++i)
                if(!cle->chaine[i]
                   || b->etiquette->chaine[i] != cle->chaine[i]) {
                    noeud* n_inter = (noeud*) malloc(sizeof(noeud));
                    branche* b_suffixe = (branche*) malloc(sizeof(branche));
                    n_inter->contenu = NULL;
                    n_inter->liste_fils = b_suffixe;
                    b_suffixe->suivant = NULL;
                    b_suffixe->noeud_fils = b->noeud_fils;
                    b->noeud_fils = n_inter;
                    mot* pre = prefixe(b->etiquette, i);
                    mot* suf = suffixe(b->etiquette, i);
                    b_suffixe->etiquette = suf;
                    free(b->etiquette->chaine);
                    free(b->etiquette);
                    b->etiquette = pre ;
                    break;
                }
            mot* suffixe = (mot*) malloc(sizeof(mot));
            suffixe->taille = cle->taille - b->etiquette->taille;
            suffixe->chaine = (cle->chaine + b->etiquette->taille);
            return insertion(b->noeud_fils, suffixe, valeur);
        }
        pointeur_precedent = &(b->suivant);
    }
    branche* nouveau = (branche*) malloc(sizeof(branche));
    nouveau->etiquette = copie(cle);
    nouveau->suivant = b;
    *pointeur_precedent = nouveau;
    nouveau->noeud_fils = (noeud*) malloc(sizeof(noeud));
    nouveau->noeud_fils->contenu = valeur;
    nouveau->noeud_fils->liste_fils = NULL;
}
```

```c
void set(dictionnaire* dict, mot* cle, char* valeur) {
    insertion(dict->racine,cle,valeur);
}
```

5 Liste des éléments

```c
typedef struct maillon {
    mot* cle;
    char* valeur;
    struct maillon* suivant;
    struct maillon* précédent;
} maillon;

typedef struct liste_cle_valeur {
    maillon* debut;
    maillon* fin;
} liste_cle_valeur;
```

```c
liste_cle_valeur* creer_liste_cle_valeur_vide() {
    liste_cle_valeur* l = (liste_cle_valeur*) malloc(sizeof(liste_cle_valeur));
    l->debut = NULL;
    l->fin = NULL;
    return l;
}
```

```c
void detruire_liste_cle_valeur(liste_cle_valeur* l) {
    maillon* m;
    while (l->debut) {
        m = l->debut->suivant;
        free(l->debut);
        l->debut = m;
    }
    free(l);
}
```

```
void prefixer_liste(liste_cle_valeur* l, mot* m) {
    for (maillon* i = l->debut ; i ; i = i->suivant ) {
        mot* pre_cle = prefixer(i->cle, m);
        free(i->cle->chaine);
        free(i->cle);
        i->cle = pre_cle;
    }
}
```

```
void concatener(liste_cle_valeur* l1, liste_cle_valeur* l2) {
    if(l1->fin)
        l1->fin->suivant = l2->debut;
    else
        l1->debut = l2->debut;
    l1->fin = l2->fin;
    free(l2);
}
```

```
liste_cle_valeur* construction_elements(noeud* n) {
    liste_cle_valeur* l_tmp;
    liste_cle_valeur* l = creer_liste_cle_valeur_vide();
    if(n->contenu) {
        maillon* m = (maillon*) malloc(sizeof(maillon));
        m->cle = (mot*) malloc(sizeof(mot));
        m->cle->taille = 0;
        m->cle->chaine = (char*) malloc(sizeof(char) * m->cle->taille );
        m->cle->chaine[m->cle->taille] = '\0';
        m->valeur = n->contenu;
        l->debut = m;
        l->fin = m;
    }
    for (branche* b = n->liste_fils; b; b = b->suivant) {
        l_tmp = construction_elements(b->noeud_fils);
        prefixer_liste(l_tmp, b->etiquette);
        concatener(l,l_tmp);
    }
    return l;
}
```

```
liste_cle_valeur* elements(dictionnaire* d) {
    return construction_elements(d->racine);
}
```

6 Affichage des résultats de la fiche réponse

```
#include <stdio.h>
#include <stdlib.h>
#include "dictionnaire.h"

mot* creer_mot(const char* s) {
    mot* m = (mot*) malloc(sizeof(mot));
    for (uint i = 0; s[i]; ++i)
        ++(m->taille);
    m->chaine = (char*) malloc(sizeof(char) * m->taille);
    for (uint i = 0; i <= m->taille; ++i)
        m->chaine[i] = s[i];
    return m;
}

void detruire_mot(mot* m) {
    free(m->chaine);
    free(m);
}
```

```
int main( int argc, char** argv ) {
    dictionnaire* d = creer_dictionnaire_vide();
    mot* cles[5] = {   creer_mot("nadine"),
                       creer_mot("nadtine"),
                       creer_mot("nad"),
                       creer_mot("adine"),
                       creer_mot("nadi") };
    mot* valeurs[5] = {    creer_mot("cuisine"),
                           creer_mot("mezzanine"),
                           creer_mot("tajine"),
                           creer_mot("comptine"),
                           creer_mot("DEFAUT") };
    for (uint i = 0; i < 4; ++i)
        set(d,cles[i],valeurs[i]->chaine);
    for (uint i = 0; i < 5; ++i)
        printf("Cle : %s - Valeur : %s\n",
               cles[i]->chaine, get(d,cles[i],valeurs[4]->chaine));
    printf("Liste éléments : [ ");
    liste_cle_valeur* l = elements(d);
    for (maillon* m = l->debut; m ; m = m->suivant)
        printf("(%s, %s), ",m->cle->chaine,m->valeur);
    printf("]\n");
    detruire_liste_cle_valeur(l);
    for (uint i = 0; i < 5; ++i) {
        detruire_mot(cles[i]);
        detruire_mot(valeurs[i]);
    }

    return EXIT_SUCCESS;
}
```