Licence 3 Informatique 2009-2010
Projet d'Algorithmique I
"Implantation d'un dictionnaire de type T9"

Châtel Grégory, gchatel
Bassinot Hervé, hbassino

5 janvier 2010

Table des matières

Résumé

Pour faciliter la saisie des chaînes de caractères, certains téléphones portables utilisent des dictionnaires de prédiction, comme par exemple le dictionnaire T9. La méthode consiste à saisir chaque mot en tapant une fois le chiffre associé à chaque lettre du mot, comme si on tapait le mot sur un clavier normal. Par exemple, pour saisir “jour”, il faut taper “5687”. A partir de cette suite de chiffres le téléphone, aidé d’un dictionnaire contenant tous les mots du français, trouve le mot qu’on voulait saisir. Dans la plupart des cas, il ne correspond qu’un seul mot à la suite de chiffres, mais parfois plusieurs choix peuvent être proposés : une touche (souvent 0) permet alors de choisir parmi les solutions. Par exemple, les mots “jour”, “loup”, “lots” correspondent à la suite “5687”. Le but du projet est d’implanter le logiciel proposant les mots se trouvant dans le téléphone portable.

1 Présentation de l'application

Le but de l'application "Implantation d'un dictionnaire de type T9" est de permettre de construire un dictionnaire T9 utilisable dans notre application pour chercher des mots à partir d'un fichier texte faisant office de dictionnaire. Le scénario du programme est simple, l'utilisateur lance le programme avec comme argument de la ligne de commande un fichier représentant le dictionnaire à utiliser. Le programme va alors construire l'arborescence correspondant aux mots du fichier. Le programme permet ensuite à l'utilisateur de chercher des mots correspondant à une entrée formée d'une suite de numero de touche, c'est à dire les caractères numériques 2 à 9. Une fois le mot saisi le programme va parcourir le dictionnaire qu'il possède en mémoire et afficher les résultats correspondants à l'entrée de l'utilisateur couplée au options du programme (affichage préfixe, ...).

Le fichier texte transmis au programme devra être écrit sous une forme spécifique. Il ne doit contenir que des caractères alphabétiques non accentués, à raison d'un mot par ligne.

Diverses options seront proposées à l'utilisateur, en effet, il est possible d'effectuer les actions suivantes :

- L'affichage de l'ensemble des mots correspondants à l'entrée saisie au clavier.
- L'affichage de l'ensemble des mots ayant un préfixe correspondant à l'entrée saisie au clavier.
- L'affichage complet des statistiques du dictionnaire (Nombre de mots connus par le programme, nombre de noeuds des arbres du dictionnaire).
- L'enregistrement du dictionnaire dans l'ordre lexicographique dans un fichier.
- L'enregistrement du dictionnaire dans un fichier sous la fome d'un graphe.
- La minimisation de l'arbre du dictionnaire.

Toutes ces fonctionnalités seront détaillées plus précisement dans la suite du rapport. Par défaut, le programme affiche les mots qui correspondent strictement à l'entrée de l'utilisateur, sans minimiser ses arbres.

Durant la réalisation de ce projet, un point d'honneur a été mis à écrire du code source modulaire pour en faciliter la mise à jour, la correction et la lisibilité.

2 Notice d'utilisation du programme

Cette partie du rapport explique en détail comment utiliser le programme ainsi l'ensemble de ses options.

2.1 Compilation du programme T9

Pour permettre la compilation du programme, nous avons écrit un makefile permetant entre autre de compiler l'ensemble des fichiers du projet. Pour construire l'exécutable T9, il faut se placer dans le répertoire principal du projet et taper la commande suivante :

```
make
```

Cette commande va donc compiler l'ensemble des modules du programme avec les options suivantes : -ansi -Wall -pedantic -Werror -O3. Cette combinaison de flag permet d'assurer le respect de la norme ansi et pedantic du langage C ainsi que la compilation du programme sans aucun warning. De plus, l'option -O3 indique à gcc qu'il doit effectuer des optimisations affectant le temps d'exécution du programme.

L'ensemble des fichiers objets générés à la compilation sont stockés dans le répértoire obj. L'éxecutable est créé dans le répertoire principal du projet sous le nom de "T9".

Notre fichier makefile permet également de supprimer l'ensemble des fichiers objets générés lors de la compilation grace à la commande suivante :

```
make clean
```

Nous pouvons également à l'aide du Makefile créer une archive du projet à l'aide de la commande :

```
make archive
```

Cette commande supprime tous les fichiers objet puis creer une archive complète du projet, il est possible de changer le nom de cette archive simplement dans le fichier makefile.

2.2 Execution du programme T9

L'ensemble du jeu d'essais du programme est situé dans le repertoire corpus. Le programme ne charge pas de dictionnaire par défaut, il est donc necéssaire de lui passer le nom d'un dictionnaire à chaque lancement.

2.2.1 Execution normale

Pour exécuter le programme avec son comportement par défaut, on utilise la ligne de commande suivante :

```
./T9 corpus/dela-fr-public.dic.simple.txt
```

Ainsi cette commande va initialiser le dictionnaire du programme avec le fichier dela-fr-public.dic.simple.txt situé dans le répertoire corpus.

2.2.2 Execution avec options

Les options supportées par le programme sont les suivantes :

- -t : L'enregistrement du dictionnaire dans l'ordre lexicographique dans un fichier externe.
- -n : L'affichage des statistiques du dictionnaire.
- -b : L'enregistrement du dictionnaire dans un fichier sous forme de graphe.
- -x : L'affichage de l'ensemble des mots ayant un préfixe correspondant à l'entrée saisie.
- -m : La minimisation des arbres du programme.

Pour lancer le programme avec une option on utilise la ligne suivante :

```
./T9 corpus/dela-fr-public.dic.simple.txt -option
```

Où -option est le nom de l'option à appliquer au programme.

Exemple : Cette commande lance le programme T9 avec l'option d'affichage prefixe.

```
./T9 corpus/dela-fr-public.dic.simple.txt -x
```

Il est également possible de cumuler les options avec la syntaxe suivante :

```
./T9 corpus/dela-fr-public.dic.simple.txt -xbm
```

Ceci lance le programme avec les options -x -b et -m du programme. Le fait que les options de la ligne de commande soient traitées en utilisant getopt autorise une grande souplesse dans le passage des arguments au programme.

2.2.3 Gestion des erreurs d'exécution

Si une option passée par la ligne de commande au programme n'est pas valide, l'éxecution prend fin immediatement. De même, si plusieurs fichiers dictionnaires sont passés en argument ou bien si le fichier représentant le dictionnaire ne peut pas être ouvert, l'exécution se termine. En cas d'erreur, les messages suivant sont affichés :

```
Erreur : Argument transmis incorrect.
```

Indique que les options transmises sont incorrectes ou que plusieurs noms de dictionnaire sont présents.

```
Erreur : Nom de fichier invalide.
```

Indique que le dictionnaire ne peut pas être ouvert.

2.3 Déroulement du programme T9

Durant le lancement du programme, les arbres représentant le dictionnaire sont construit et les éventuelles options sont appliquées. Durant la phase de construction des arbres, le programme récupère un à un les mots du dictionnaire puis les insère dans les arbres. Notre programme ne prend en compte que les caractères alphabétiques minuscules non accentués. Si certains mots du dictionnaire ne sont pas valides, le programme les passes, un message d'erreur est cependant affiché à l'utilisateur. Lorsque le programme a terminé son initialisation et qu'il est pret à fonctionner, il l'indique à l'utilisateur grâce au message suivant :

```
Saisi OK
```

L'utilisateur peut maintenant entrer les mots qu'il souhaite rechercher sous la forme d'une suite de caractère de 2 à 9 puis valider en appuyant sur la touche entrée.

Si les caractères entrés ne sont pas valides, le programme affiche le message d'erreur suivant :

```
Mot non valide !!
```

Si aucun mot ne correspond à la chaîne entrée par l'utilisateur, le programme affiche le message suivant :

```
Aucun mot trouve !!
```

Dans le cas contraire le programme affiche l'ensemble des mots trouvés à raison d'un mot par ligne.

3 Structures de données

3.1 Stockage du dictionnaire

Pour stocker les mots du dictionnaires, nous avons choisi d'utiliser les structures suivantes :

```
#define NB_LETTRE 4
#define NB_TOUCHE 8

typedef enum statut {
    TERMINAL,
    NON_TERMINAL
} Statut;

typedef enum visite {
    VISITE,
    NON_VISITE
} Visite;

typedef struct noeudAuxiliaire {
    struct noeudAuxiliaire * fils[NB_LETTRE];
    Statut statut;
    Visite visite;
    unsigned int hache;
    unsigned int numero;
} NoeudAuxiliaire, * ArbreAuxiliaire;

typedef struct noeudPrincipal {
    struct noeudPrincipal * touche[NB_TOUCHE];
    ArbreAuxiliaire arbreAux;
    Visite visite;
    unsigned int hache;
    unsigned int numero;
} NoeudPrincipal, * ArbrePrincipal;
```

Nous avons choisi une structure d'arbre lexicographique utilisant un tableau de fils. Cette méthode a pour avantage d'être très simple à manipuler contrairement à une représentation par fils gauche frère droit qui fait intervenir un parcours de liste et qui utilise plus de place dans le cas de noeuds à fort facteur de branchement.

Dans la structure de l'arbre auxiliaire, on trouve :

- Un tableau de pointeurs sur noeud auxiliaire représentant les fils de ce noeud.
- Un champs statut qui indique si le noeud est terminal ou non.
- Un champs visite qui indique si le champs a déjà été visité lors du calcul du nombre de noeuds auxiliaires (ce champs est rendu nécessaire par la fusion et la minimisation, il permet de ne pas compter plusieurs fois le même noeud).
- Un champs hache qui contiendra la valeur du haché du noeud lors de la minimisation de l'arbre.
- Un champs numéro qui contiendra le numéro du noeud lors de la minimisation (ce fonctionnement sera expliqué plus loin dans le rapport).

Dans la structure de l'arbre principal, on trouve :

- Un tableau de pointeurs sur noeud principal représentant les fils de ce noeud.
- Un champs arbreAux qui pointe vers l'arbre auxiliaires qui contient les positions des lettres à utiliser pour obtenir les mots reconnus par ce noeud principal.
- Un champs visite ayant le même usage que celui de la structure d'arbre auxiliaire.
- Un champs hache ayant le même usage que celui de la structure d'arbre auxiliaire.
- Un champs numero ayant le même usage que celui de la structure d'arbre auxiliaire.

3.2 Gestion des arguments

Pour la gestion des arguments, on utilise la structure suivante :

```
#define NB_ARGUMENT 5

typedef struct argument{
    char options[NB_ARGUMENT];
    char* nomFichier;
} Argument;
```

Cette structure contient une table de caractères qui seront utilisés comme des booléens lors du traitement des arguments du programme. Chaque case de ce tableau corresponds à une option valide pour le programme, cette case contiendra vrai après traitement si l'option en question était présente sur la ligne de commande.

3.3 Lecture du dictionnaire

La lecture du dictionnaire se fait grâce à la structure suivante :

```
typedef struct dictionnaire {
    FILE* fichier;
    char nom[TAILLE_FICHIER];
    char buffer[TAILLE_BUFFER];
    int curseur;
    int nbcaractere;
    int taillemot;
} Dictionnaire;
```

Cette structure de données nous permet de faciliter la lecture des mots dans le fichier dictionnaire. Elle contient les champs suivant : le fichier texte contenant l'ensemble des mots du dictionnaire, le nom du fichier, un buffer qui sert à stocker les caractères du fichier pour éviter un trop grand nombre d'appels système, un curseur représentant la position durant le parcours du buffer, un champ nbcaractere qui indique si le buffer est plein ou non et un champ taillemot qui renseigne sur la taille du mot alloué dynamiquement pour d'éventuelles réallocations. Nous décrirons plus en details dans la suite du rapport comment cette structure est utilisée.

3.4 Fusion

Voici la structure de liste chainée dont la fusion a besoin :

```
typedef struct maillonNoeudPrincipal {
    NoeudPrincipal* noeud;
    struct maillonNoeudPrincipal* suivant;
} MaillonNoeudPrincipal, *ListeNoeudPrincipal;
```

La structure de donnée utilisée pour la fusion est un tableau de liste chainée. Ces listes contiennent un pointeur sur un noeud principal. Leur usage sera décrit plus tard dans le rapport.

3.5 Minimisation

Voici les structures de données utilisées pour la minimisation des arbres :

```
typedef enum {VIDE, OCCUPE} Etat;

typedef struct caseTable {
    void* adresse;
    unsigned int hache;
    Etat etat;
} CaseTable;

typedef struct table {
    CaseTable* tableau;
    unsigned int tailleTable;
    int (*comparer)(void*, void*);
} Table;
```

L'algorithme de minimisation des arbres de ce projet emploie une table de hachage. Pour implémenter cet algorithme, nous avons choisi d'utiliser du hachage fermé, par conséquent, il a été necessaire d'ajouter un champs etat à la structure de case de la table. Ce champs indique si la case en question est utilisée ou non. Chaque case contient un pointeur générique, nous avons choisi ce type de pointeur pour stocker les adresses pour pouvoir stocker différents types de structures (noeud principaux et auxiliaires dans notre cas). Chaque case connaît la valeur du hache de l'élément qu'elle contient pour des raisons pratiques. En effet, il n'est pas possible d'accéder à la valeur du hache du noeud contenu depuis le pointeur générique.

La structure de la table contient un tableau de case qui représente la table elle même, un champs tailleTable qui indique la taille de ce tableau et un pointeur sur une fonction de comparaison servant à éliminer les collisions non souhaitées.

3.6 Libération

Voici la structure de donnée utilisée pour la libération :

```
typedef enum {VIDE, OCCUPE} Etat;

typedef struct caseLiberateur {
    void* adresse;
    Etat etat;
} CaseLiberateur;
```

Le fait qu'on minimise les arbres auxiliaires et principaux entraine qu'on mette en place un système de libération particulier pour ne pas libérer deux fois le même noeud. Pour ce faire, on utilise une table de hachage dont l'usage sera décrit plus tard dans le rapport.

4 Algorithmes du projet

4.1 Fonctions d'initialisation du dictionnaire

Pour initialiser les structures de données necessaire au fonctionnement du dictionnaire T9 il faut recuperer tous les mots contenu dans le fichier dictionnaire transmis en argument et les inserer dans l'arbre.

4.2 Recherche normale

L'algorithme de recherche normale est le suivant : L'utilisateur saisi un mot, si celui-ci est valide, on descends dans l'arbre principal en suivant les branches indiquées par ce mot, s'il est impossible de descendre dans l'arbre principal avant qu'on ait lu la totalité du mot, alors aucun mot ne correspond à son entrée. Une fois qu'on a lu la totalité du mot, on examine le noeud de l'arbre principal sur lequel on est arrivé. Si ce noeud pointe sur un arbre auxiliaire, alors il y a des mots qui correspondent à l'entrée de l'utilisateur, pour les trouver, on va parcourir l'arbre auxiliaire en question. On va parcourir toutes les branches faisant la même taille que le mot entré par l'utilisateur. Durant le parcours de ces branches, on mémorise au fur et à mesure dans un buffer les numéros qui correspondent aux branches empruntées. Une fois arrivé à la bonne hauteur et si le noeud courant est terminal, on affiche le mot qui corresponds à la combinaison de caractères entrés par l'utilisateur et le contenu du buffer du parcours de l'arbre auxiliaire.

4.3 Recherche préfixe

L'algorithme de recherche de mot par préfixe est basé sur la même logique que l'algorithme de recherche normale. L'utilisateur saisi un mot, si celui-ci est valide, on cherche le noeud de l'arbre principal qui lui correspond. Une fois qu'on l'a trouvé, on va parcourir l'arbre principal ayant comme racine ce noeud en stockant dans un buffer les numéros correspondant aux branches empruntées. Si durant ce parcours, un noeud courant possède un arbre auxiliaire, on le parcours de la même façon que dans la recherche normale à la différence près que les caractères qui vont être affichés sont déterminés dans un premier temps par la combinaison du mot entré par l'utilisateur et le contenu du buffer du parcours de l'arbre auxiliaire jusqu'à ce que la hauteur courante du parcours de l'arbre auxiliaire soit égale à la taille du mot, puis ensuite l'affichage sera fait en fonction du buffer de parcours de l'arbre principal et du buffer de parcours de l'arbre auxiliaire.

4.4 Fusion

Notre algorithme de fusion se base sur le fait que deux arbres auxiliaires situés à des hauteurs différentes dans l'arbre principal peuvent toujours être fusionnés.

L'algorithme est le suivant : On crée un tableau de listes chainées de taille équivalente à la hauteur de l'arbre principal. On effectue ensuite un parcours en profondeur de l'arbre principal. Durant ce parcours, si un noeud courant possède un arbre auxiliaire, on stocke l'adresse de ce noeud principal dans la liste du tableau dont l'indice correspond à la hauteur courante dans l'arbre principal. Une fois que l'arbre à été entièrement

parcouru, on va fusionner les arbres. Pour ce faire, on stocke dans un tableau un noeud de chaque liste du tableau de liste (tant que c'est possible). Tous les arbres de ce tableaux sont forcement fusionnables puisqu'ils sont situés à des hauteurs différentes dans l'arbre principal. On fusionne ensuite tous les arbres auxiliaires contenus dans les noeuds principaux de ce tableau. On continue tant que le tableau de liste contient des noeuds.

4.5 Minimisation

4.5.1 Arbres auxiliaires

Pour minimiser les arbres auxiliaires, nous avons employé l'algorithme qui nous a été donné dans le sujet du projet. Cependant, en application, quelques problèmes se posent avec cet algorithme, en effet, dans l'explication du théorème, la fonction de hachage est supposée parfaite et la table suffisement grande pour ne pas à avoir a utiliser l'opérateur modulo. Cependant, en pratique, ces deux conditions ne sont pas remplies, par conséquent il est possible d'avoir plusieurs noeuds qui possèdent le même haché alors qu'ils ne doivent pas être fusionnés pour pallier à ce problème, nous avons choisi d'utiliser une fonction de comparaison chargée de dire s'il est possible de fusionner deux noeuds. De plus, le fait qu'il est possible d'obtenir des collisions non souhaitées sur les valeurs des hachés entraine le fait que cette fonction de comparaison ne peut pas se servir des valeurs des hachés pour comparer les noeuds. Par conséquent, nous avons du introduire un nouveau champs "numero" dans la structure d'arbre auxiliaire pour identifier un noeud qui va être présent après la minimisation de manière unique.

L'algorithme de comparaison est donc le suivant :

```
booléen compare(noeud1, noeud2) {
    Si les statuts des noeuds sont différents :
    |   Retourner faux.

    Pour chaque numero de fils i :
    |   Si le fils i du noeud1 est null et que le fils i du noeud2 n'est pas null ou
    |   Si le fils i du noeud1 n'est pas null et que le fils i du noeud2 est null ou
    |   Si les deux fils ne sont pas null et qu'ils ont des numeros différents :
    |
    |   |   Retourner faux.

    Retourner vrai.
}
```

Dans cet algorithme, on ne compare pas la valeur du haché des noeuds car cette vérification est faite au niveau de la table de hachage.

Voici l'algorithme de la fonction de hachage des noeuds auxiliaires :

```
int hacher(arbre, maximum) {
    int resultat = 922337206. /* Valeur arbitraire */
    int tailleEntier = taille des entiers en mot machine (sizeof).
    int tailleMotMachine = taille du mot machine (CHAR_BIT).
    int nombreFils = 0.

    Pour chaque fils de numéro i de arbre :
```

```
    |   Si le fils numéro i n'est pas null :
    |   |   tmp = haché du fils numéro i.
    |   |
    |   |   tmp = (tmp >> ((i % tailleEntier) * tailleMotMachine)) |
    |   |         (tmp << ((i % tailleEntier) * tailleMotMachine)).
    |   |   resultat ^= tmp;
    |   |   nbFils++;

    Si nombreFils est pair :
    |   resultat = (resultat >> ((tailleEntier / 2) * tailleMotMachine)) |
    |              (resultat << ((tailleEntier / 2) * tailleMotMachine)).

    resultat = resultat % maximum;

    Si arbre est final :
    |   resultat = resultat | 1.
    Sinon :
    |   resultat = resultat & ~1.

    Retourner resultat.
}
```

Cette fonction de hachage permet de répartir de façon convenable les valeurs qu'elle renvoie dans l'interval des valeurs autorisées. La totalité de l'information disponible est utilisée car aucun dépassement de capacité n'est authorisé. Le fait que l'on change la parité du haché selon le statut (terminal ou non) du noeud entraine que la taille de la table doit être paire. En effet, si la taille de la table est pair, la valeur maximale que l'on peut trouver grâce au modulo est impair et par conséquent changer la valeur de ce dernier bit ne peut pas avoir de conséquence facheuses.

4.5.2 Arbre principal

Durant la réalisation de ce projet, nous avons eu un problème durant la réalisation de la minimisation des arbres principaux. En effet, nous n'avons pas réussi à mettre en place cette fonctionnalité de manière satisfaisante.

Le problème vient du fait que les liens formés après la minimisation ne sont pas corrects, ce qui provoque un gain de mot reconnaissable ainsi qu'une perte de noeuds auxiliaires. Nous pensons que cette erreur proviens de notre algorithme de comparaison de noeud qui est le suivant :

```
booléen compare(noeud1, noeud2) {
    Si les statuts ou les profondeurs des noeuds sont différents :
    |   Retourner faux.

    Pour chaque numero de fils i :
    |   Si le fils i du noeud1 est null et que le fils i du noeud2 n'est pas null ou
    |   Si le fils i du noeud1 n'est pas null et que le fils i du noeud2 est null ou
    |   Si les deux fils ne sont pas null et qu'ils ont des numéros différents :
    |
    |   |   Retourner faux.

    Retourner vrai.
}
```

La fonction de hachage utilisée est du même type que celle des arbres auxiliaires, cependant, elle utilise aussi la valeur du haché de la racine de

l'arbre auxiliaire contenu dans le noeud ainsi que la profondeur pour le calcul du haché.

En utilisant le dictionnaire dela-fr fourni pour le projet, on obtient le résultat suivant :

```
Statistiques de l'arbre ...
        Nombre de mots dans l'arbre             : 610631
        Nombre de noeuds de l'arbre principal   : 670097
        Nombre de noeuds des arbres auxiliaires : 574061
```

Au lieu des résultats suivant sans minimiser les arbres principaux :

```
Statistiques de l'arbre ...
        Nombre de mots dans l'arbre             : 609737
        Nombre de noeuds de l'arbre principal   : 1053622
        Nombre de noeuds des arbres auxiliaires : 654815
```

4.6 Statistiques

4.6.1 Calcul du nombre de mots

L'algorithme utilisé pour compter le nombre de mots reconnus par le dictionnaire est le suivant : On parcours l'arbre principal en mémorisant la hauteur courante, si durant ce parcours, un noeud principal possède un noeud auxiliaire, on va parcourir cet arbre auxiliaire pour compter le nombre de noeuds terminaux situés à une profondeur équivalente à la profondeur courante de l'arbre principal. Le nombre de mots reconnus par l'arbre principal est la somme des résultats de tous ces parcours.

Cet algorithme est valable sur les arbres minimisés ou non.

4.6.2 Calcul du nombre de noeuds des arbres

Pour calculer le nombre de noeuds d'un arbre principal ou auxiliaire d'un arbre, il a été nécessaire d'ajouter un champs visite dans leurs structures. Ce champs permet d'indiquer si un noeud a déjà été visité ou non pour éviter que le même noeud soit compté plusieurs fois. Il est nécessaire d'appeler une fonction chargée de réinitialiser la valeur de ce champs après un parcours durant lequel il a été utilisé.

L'algorithme pour l'arbre principal est le suivant :

```
int compteNoeud(arbre) {
    int somme = 1.

    Si arbre est null ou a déjà été visité :
    |   Retourner 0.

    Marquer le noeud comme visité.

    Pour chaque fils f de arbre :
    |   somme = somme + compteNoeud(f).

    Retourner somme.
}
```

L'algorithme pour les arbres auxiliaires est le suivant : On parcours l'arbre principal, pour chaque noeud principal qui possède un arbre auxiliaire, on parcours ce dernier en comptant le nombre de noeuds et en les marquant de la même façon que dans l'algorithme pour compter le nombre de noeuds de l'arbre principal.

Pour simplifier le programme, cette méthode de calcul est utilisée

4.7 Libération

Pour libérer les arbres durant l'exécution ou à la fin du programme, nous avons du utiliser une structure de donnée qui permet de ne pas libérer plusieurs fois la même adresse. En effet, après la minimisation, plusieurs noeuds peuvent faire référence sur la même adresse, par conséquent la phase de libération est problématique.

Pour résoudre ce problème, nous avons implémenter une table de hachage dans laquelle on stocke les adresses qu'on souhaite libérer. Si un adresse est déjà présente et qu'on souhaite l'inserer de nouveau, la fonction de libération ne fait rien. Après avoir stocker l'ensembles des adresses que l'on souhaite liberer dans la table de hachage on appelle une fonction qui parcours l'ensemble de la table de hachage en la vidant et en libérant les adresses contenues dans toutes les cases pleines qu'elle rencontre.

5 Idées d'amélioration

Grâce à notre developpement il peut être très facile d'augmenter ou de diminuer le nombre de touches et de caractère du programme.

6 Conclusion

En conclusion nous pouvons dire que nous avons essayé de faire le programme le plus simple et intuitif possible. Nous avons developpé les fonctions au plus simple possible avec comme régle : une fonction pour une action. Nous avons fait en sorte que les complexités des fonctions soient les plus faible possible en priviligiant les algorithmes les plus efficaces.