Projet GM3

Conception et implémentation du type abstrait de données Dictionnaire

Elliot Sisteron
Antoni Markovski

À l'attention de Mme. Chaignaud

Introduction

Lorsque l'on veut répertorier un « grand » nombre d'informations, il est souvent bien plus pratique de considérer le stockage numérique. Pour un dictionnaire français, par exemple, la version papier devient rapidement encombrante (1910 pages de mots et définitions pour *Le petit Larousse*, édition 2012). De plus, nous traitons cette information bien plus lentement qu'une machine : la recherche d'un mot dans un dictionnaire papier peut parfois s'avérer laborieuse. Il est aussi important de souligner que la modification (insertion/suppression de mots) d'un dictionnaire papier (qui est régi par l'ordre lexicographique) engendre la création d'une multitude d'éditions... On comprend alors rapidement l'intérêt de « numériser » un tel objet.

Dans ce projet, on cherchera à être capable de manipuler des dictionnaires électroniques de mots avec leurs définitions. Pour cela, nous commencerons par définir rigoureusement le type abstrait de données (ou TAD) *Dictionnaire*. Puis, nous étudierons la conception du type de données Dictionnaire où nous passerons en revue une interprétation bien précise de ce TAD. On débattra alors de son efficacité, sa fiabilité, sa maintenabilité et de sa portabilité.

Table des matières

Le TAD Dictionnaire

Dans cette partie, nous allons définir le type abstrait de données *Dictionnaire* pour ensuite essayer de le conceptualiser.

1.1 Présentation du TAD Dictionnaire

À première vue, il apparaît qu'un dictionnaire est composé de plusieurs couples (un mot et sa définition). Un objet abstrait de type *Dictionnaire* aura donc deux paramètres : *Mot* et *Définition*. En tant qu'utilisateur, on arrive bien à percevoir que les opérations suivantes doivent être réalisables :

— Savoir si un mot est présent dans le dictionnaire (en tant qu'« humain » : rechercher dans l'index si le mot est présent).
— Rechercher une définition à partir d'un mot (en tant qu'« humain » : ouvrir le dictionnaire et localiser le mot recherché pour avoir sa définition).

Mais il faut aussi se placer en tant qu'« éditeur », d'où la nécessité des opérations :

— Créer un dictionnaire « vide » (en tant qu'« humain » : aller chez l'imprimeur et demander un livre vierge).
— Ajouter un mot avec sa définition dans le dictionnaire, en prenant soin de l'insérer en suivant l'ordre lexicographique (en tant qu'« humain » : prendre une plume et écrire le mot avec sa définition au bon endroit, ce qui n'est pas pratique).
— Supprimer un mot du dictionnaire (en tant qu'« humain » : raturer le mot en question avec sa définition).

On peut donc proposer le TAD *Dictionnaire* suivant :

Sorte:	Dictionnaire	
Paramètre(s):	Mot, Définition	
Utilise:	**Booléen**	
Opération(s):	dictionnaire:	$\rightarrow$ Dictionnaire
	estPresent:	Dictionnaire $\times$ Mot $\rightarrow$ **Booléen**
	obtenirDefinition:	Dictionnaire $\times$ Mot $\nrightarrow$ Dictionnaire
	inserer:	Dictionnaire $\times$ Mot $\times$ Définition $\rightarrow$ Dictionnaire
	supprimer:	Dictionnaire $\times$ Mot $\rightarrow$ Dictionnaire
Axiome(s):	- inserer(inserer(d, m, def), m, def) = inserer(d, m, def)	
	- supprimer(supprimer(d, m), m) = supprimer(d, m)	
	- supprimer(inserer(d, m, def), m) = d	
	- obtenirDefinition(inserer(d, m, def), m) = def	
	- obtenirDefinition(supprimer(d, m), m) = creerDefinition()	
	- estPresent(inserer(d, m, def), m) = Vrai	
	- non(estPresent(supprimer(d, m), m)) = Vrai	
	- ...	

On s'intéressera par la suite à une opération de fusion de deux dictionnaires « compatibles » (qui, pour un même mot, se réfèrent à la même définition).

1.2 La structure de données choisie

On commence par choisir d'implémenter les paramètres de ce TAD (*Mot* et *Définition*) par des chaînes de caractères. Dans la suite du projet, on supposera que l'on sait gérer ces chaînes de caractères. On pourra donc tester l'égalité de deux chaînes à l'aide des opérateurs $=$ et $\neq$, mais aussi étudier l'ordre lexicographique à l'aide de $<$, $>$, $\leq$, $\geq$. De plus, on admettra que l'on peut accéder au $i-$ème caratère d'une chaîne à l'aide de l'opérateur [ ] (le même que les tableaux). Enfin, on supposera que l'on possède une fonction *longueur* qui renvoie la longueur d'une chaîne passée en paramètre.

Pour implémenter le TAD *Dictionnaire* en un type de données, on va utiliser une structure de données bien précise : une implémentation avec une liste chaînée ordonnée de mots avec leurs définitions (ce qui est un peu lourd en terme de complexité car l'accès à un mot est en $O(n)$). On se propose de diviser cette liste chaînée de couples de mots/définitions en 26 parties dans un tableau indicé par les lettres de l'alphabet. Ainsi, la liste chaînée de tous les mots commençant par A (avec leurs définitions) se trouvera dans la case du tableau indicée par le caractère 'A', etc.

Nous allons conceptualiser en détail cette structure de données.

1.3 Préliminaire : à propos des mots

On a vu que l'on pouvait représenter les mots et les définitions du TAD *Dictionnaire* par des chaînes de caractères. Toutefois, il nous faudra être prudent en considérant cette représentation pour des mots : par exemple, une chaîne de caractères peut très bien contenir des espaces ou de la ponctuation. C'est pour cette raison que l'on se donne la fonction suivante (facilement réalisable. . .) :

Fonction estUnMot (ch : **Chaîne) : Booléen**

On supposera que l'on a aussi une fonction *maj* qui mettra en majuscule ASCII un caractère passé en paramètre :

Fonction maj (c : **Caractère) : Caractère**

Par exemple :
- maj('b') = 'B'
- maj('à') = 'A'
- maj('É') = 'E'
- maj('L') = 'L'. . .

Implémentation avec des listes chaînées

Dans cette partie, on propose d'implémenter le TAD *Dictionnaire* à l'aide de listes chaînées.

2.1 Conception préliminaire

2.1.1 Le type Dictionnaire

On se donne un type de liste chaînée contenant un mot et sa définition et nous l'appelons ListeChaineeMD (MD pour Mot/Définition) :

Type Noeud = **Enregistrement**
  mot : **Chaîne**
  definition : **Chaîne**
  suivant : ^Noeud
FinEnregistrement

Type ListeChaineeMD = ^Noeud

On propose alors le type de données Dictionnaire suivant :

Type Dictionnaire = **Tableau[**'A'..'Z'**] de** ListeChaineeMD

On peut se représenter, sur un exemple, le type Dictionnaire par le schéma suivant :

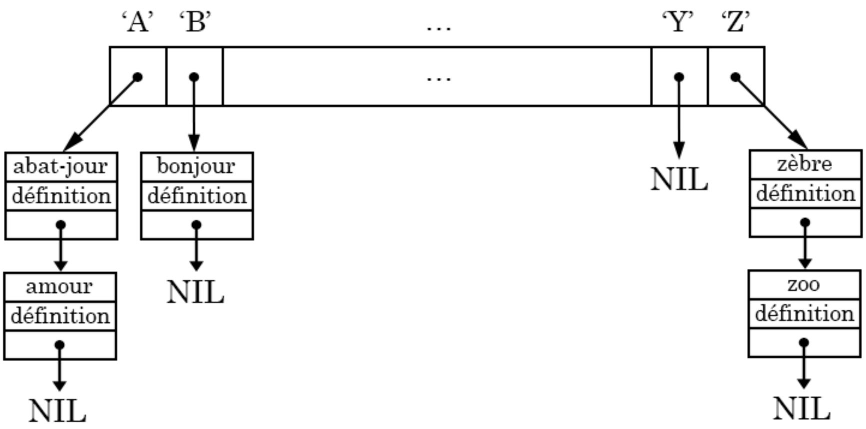

Figure 2.1 – Une implémentation du TAD *Dictionnaire* avec des listes chaînées et un tableau

2.1.2 Les signatures des opérations

On se donne les signatures des opérations du type de données Dictionnaire suivantes :

Fonction creerDictionnaire **() :** Dictionnaire
Fonction estPresent **(**d : Dictionnaire, m : **Chaîne) : Booléen**
Fonction obtenirDefinition **(**d : Dictionnaire, m : **Chaîne) : Chaîne**
Procédure inserer **(E/S** d : Dictionnaire, **E** m, def : **Chaîne)**
⌊**précondition(s)** estUnMot(m)
Procédure supprimer **(E/S** d : Dictionnaire, **E** m : **Chaîne)**
Fonction fusionner **(**d1, d2 : Dictionnaire**) :** Dictionnaire

2.2 Conception détaillée

2.2.1 La création de dictionnaire

Cet algorithme va nous permettre de créer un dictionnaire « vide ». Pour cela, il est nécessaire d'initialiser chacune des listes chaînées à NIL.

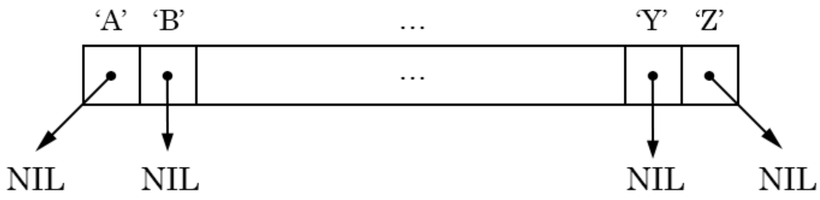

FIGURE 2.2 – Un schéma pour mieux comprendre le fonctionnement de *creerDictionnaire*

Fonction creerDictionnaire **() :** Dictionnaire
Var res : Dictionnaire, i : **Caractère**
Début
Pour i ←'A' **à** 'Z' **faire**
res[i] ← NIL
FinPour
retourner (res)
Fin

2.2.2 La présence d'un mot dans le dictionnaire

On veut tester la présence d'un mot m dans un dictionnaire d. Autrement dit, on va vérifier que le mot se trouve dans la liste chaînée correspondant à sa première lettre. La première chose à faire est donc de se positionner dans la « bonne » liste chaînée.

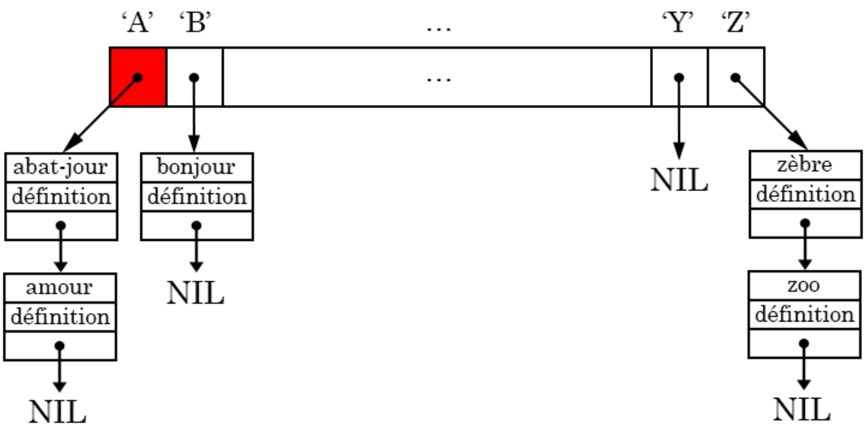

FIGURE 2.3 – Dans cet exemple, le mot recherché commence par la lettre A

Pour cela, on transforme la première lettre du mot (c'est-à-dire m[1]) en majuscule pour avoir l'indice correspondant. On se positionne donc dans la case $d[maj(m[1])]$ du tableau. À ce stade, on positionne un curseur sur la liste pour pouvoir la parcourir.

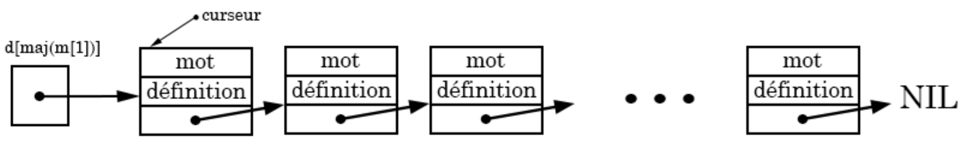

FIGURE 2.4 – On s'apprête à parcourir la liste chaînée

Pour optimiser un peu la recherche du mot m, on utilise le fait que la liste que l'on va parcourir est ordonnée. Autrement dit, on se positionnera sur le premier mot supérieur ou égal à m plutôt que de parcourir toute la liste jusqu'à ce qu'on le trouve. Si ce mot est égal à m, alors on aura trouvé m.

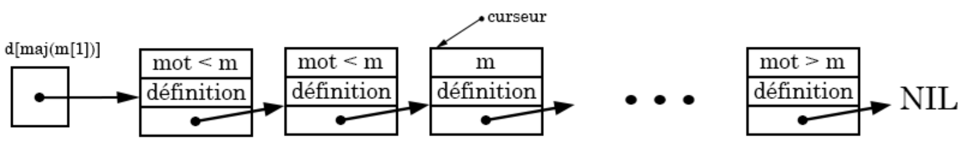

FIGURE 2.5 – Dans cet exemple on a trouvé m

Sinon, m n'est pas dans la liste car tous les mots suivants seront supérieurs strictement à m (du fait que la liste soit ordonnée).

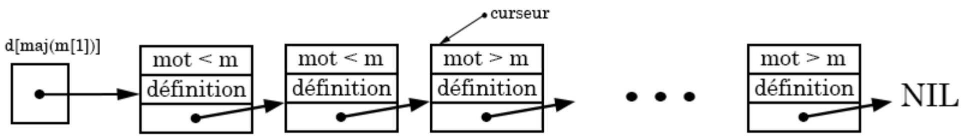

FIGURE 2.6 – Dans cet exemple, m n'est pas dans la liste

Bien sûr, il existera un cas où le parcours sera entièrement réalisé : celui où tous les mots se trouvant dans la liste sont inférieurs strictement à m.

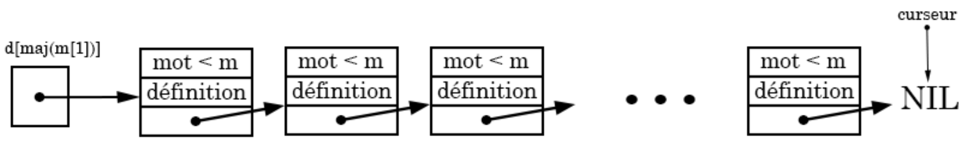

FIGURE 2.7 – Le pire des cas

Résumons tout cela. Tant que le que le curseur n'est pas arrivé au bout ou que le mot du curseur est inférieur strictement au mot recherché, on avance. Si le curseur est arrivé au bout, alors le mot n'est pas présent. Si le curseur n'est pas arrivé à la fin de la liste, alors :

— S'il se trouve sur l'élément recherché, alors l'élément en question est bien présent.
— Par contre, si le mot sur lequel il est positionné n'est pas le mot recherché, alors le mot ne se trouve pas dans la liste (car celle-ci est ordonnée et les éléments suivants sont donc strictement supérieurs).

On en déduit donc l'algorithme suivant :

Fonction estPresent (d : Dictionnaire, m : **Chaîne) : Booléen**
  Var curseur : ListeChaineeMD
Début
  curseur ← d[maj(m[1])]
  Tant que ((curseur ≠ NIL) et (curseur^.mot < m)) **faire**
    curseur ← curseur^.suivant
  FinTantQue
  retourner ((curseur ≠ NIL) et (curseur^.mot = m))
Fin

2.2.3 Récupérer une définition

On cherche à récupérer la définition d'un mot m qui est dans un dictionnaire d. Pour cela, on se place encore dans la bonne case du tableau, et on parcoure la liste chaînée pour le retrouver.

Il s'agit quasiment du même algorithme que le précédent sauf que, une fois positionné sur le bon élément de la liste, on retourne la définition. Si le mot n'a pas été trouvé, on retourne une chaîne nulle.

Fonction obtenirDefinition (d : Dictionnaire, m : **Chaîne) : Chaîne**
  Var curseur : ListeChaineeMD, res : **Chaîne**
Début
  res ← ""
  curseur ← d[maj(m[1])]
  Tant que ((curseur ≠ NIL) et (curseur^.mot < m)) **faire**
    curseur ← curseur^.suivant
  FinTantQue
  Si ((curseur ≠ NIL) et (curseur^.mot = m)) **alors**
    res ← curseur^.definition
  FinSi
  retourner (res)
Fin

2.2.4 Lois de composition externes

On a deux lois de composition externes sur un dictionnaire : l'insertion d'un mot ou la suppression d'un mot.

Insertion d'un mot avec sa définition

On cherche ici à insérer un mot m avec sa définition def dans un dictionnaire d. On récupère, là encore, la liste chaînée à laquelle m appartient dans le dictionnaire d et on veut l'insérer au bon endroit dans cette liste. On commence par positionner un curseur sur cette liste.

On distingue tout d'abord deux cas particuliers. Le cas où la liste dans laquelle on va insérer m est vide ou encore le cas où le premier mot de la liste est supérieur strictement à m. Cela signifie que nous allons devoir insérer m et sa définition en tête de liste.

Pour cela, on commence par allouer un espace mémoire de la taille de *Noeud* que l'on fait pointer sur une variable *tmp*, dans lequel on copie m et def. Ensuite, on fait pointer la case du dictionnaire $d[maj(m[1])]$ sur *tmp* et enfin on fait pointer le champ suivant de *tmp* sur le curseur.

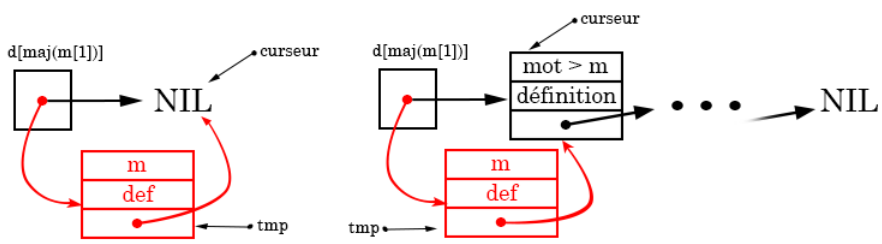

FIGURE 2.8 – Les deux cas particuliers pour l'insertion en tête de liste

On distingue aussi le cas d'insertion en fin de liste, c'est à dire le cas où tous les mots de la liste sont inférieurs strictement à m. On positionne le curseur juste avant d'arriver en fin de liste (c'est-à-dire de telle manière que le champ suivant du curseur pointe sur NIL). Dès lors, on alloue encore un espace mémoire *tmp*, on fait pointer le champ suivant de *tmp* sur le champ suivant du curseur (c'est-à-dire liste sur NIL), puis l'on fait pointer le champ suivant du curseur sur *tmp*.

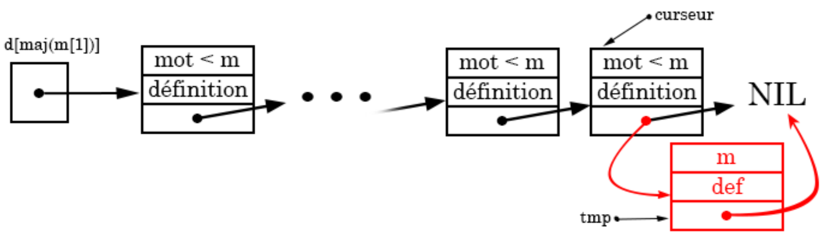

FIGURE 2.9 – Le cas particulier d'insertion en fin de liste

Pour le cas général, on va positionner le curseur sur un élément inférieur strictement à m mais aussi juste avant un élément supérieur strictement à m (on veut « intercaler » m de cette manière pour garder une liste ordonnée). Il s'agit alors de la même insertion que l'insertion en fin de liste.

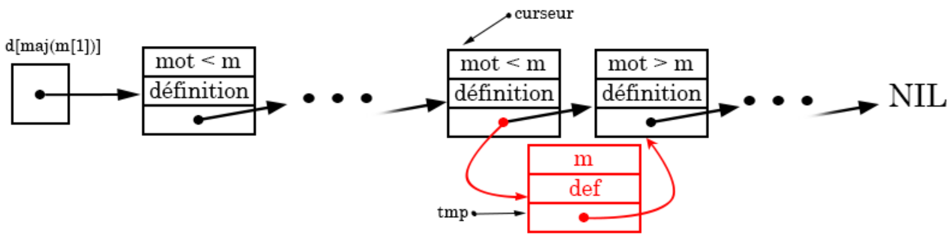

FIGURE 2.10 – Le cas général

Il ne faut pas oublier le cas particulier où le mot se trouve déjà dans la liste chaînée. Il nous faudra donc éviter de l'insérer une deuxième fois dans la liste en s'arrêtant dès qu'on le rencontre.

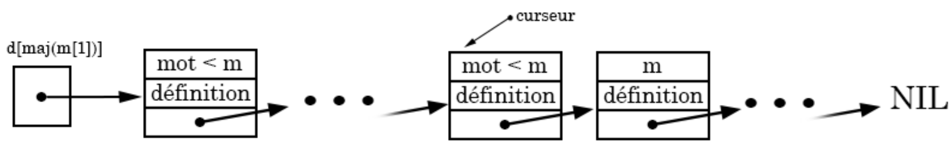

FIGURE 2.11 – On ne fait rien lorsque le mot est déjà dans la liste

Résumons-nous. On positionne un curseur. Si le curseur pointe sur NIL ou si le mot du curseur est supérieur strictement à m, alors on insère en tête. Sinon, si le curseur n'est pas positionné sur m, tant que le champ suivant du curseur ne pointe pas sur NIL et que le mot du champ suivant du curseur est inférieur strictement à m, on avance. Si l'on a arrêté d'avancer parce que le champ suivant du curseur pointait sur NIL, alors on insère en fin de liste. Sinon c'est que l'on s'est arrété parce que le mot du champ suivant du curseur était supérieur ou égal à m. S'il n'est pas égal à m, on peut insérer (et c'est la même insertion qu'en fin de liste).

D'où l'algorithme :

Procédure inserer **(E/S** d : Dictionnaire, **E** m, def : **Chaîne)**
    ⌊**précondition(s)** estUnMot(m)
    Var tmp, curseur : ListeChaineeMD
Début
    curseur ← d[maj(m[1])]
    Si ((curseur = NIL) ou (curseur^.mot > m)) **alors**
        tmp ← **allouer**(Noeud)
        tmp^.mot ← m
        tmp^.definition ← def
        d[maj(m[1])] ← tmp
        tmp^.suivant ← curseur
    Sinon
        Si (curseur^.mot ≠ m) **alors**
            Tant que ((curseur^.suivant ≠ NIL) et (curseur^.suivant^.mot < m)) **faire**
                curseur ← curseur^.suivant
            FinTantQue
            Si ((curseur^.suivant = NIL) ou (curseur^.suivant^.mot ≠ m)) **alors**
                tmp ← **allouer**(Noeud)
                tmp^.mot ← m
                tmp^.definition ← def
                tmp^.suivant ← curseur^.suivant
                curseur^.suivant ← tmp
            FinSi
        FinSi
    FinSi
Fin

Suppression d'un mot et sa définition

On cherche maintenant à supprimer un mot m dans un dictionnaire d. Là encore, on commence par positionner un curseur sur la liste qui est susceptible de contenir m. On va parcourir cette liste de la même manière que pour l'insertion, et on libèrera l'espace mémoire pris par le mot et sa définition une fois celui-ci trouvé.

Tout d'abord, si la liste est vide (c'est-à-dire si le curseur est égal à NIL), on ne fait évidemment rien. Si la liste commence par m, on procédera en faisant pointer la case du dictionnaire $d[maj(m[1])]$ sur l'élément suivant du curseur, puis on libérera le curseur (suppression en tête).

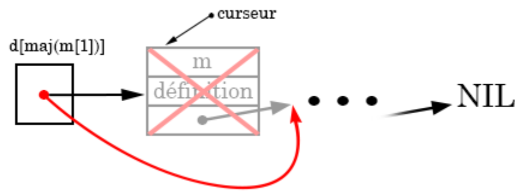

FIGURE 2.12 – Le cas particulier de suppression en tête

Concernant le cas général, il s'agit du même principe que pour l'insertion : on avance le curseur jusqu'à ce que son champ suivant pointe sur le *Noeud* contenant m. On utilise alors une variable *tmp* que l'on fait pointer sur le champ suivant du curseur. Dès lors, il suffit de faire pointer le curseur sur le champ suivant du champ suivant du curseur (c'est-à-dire sur le champ suivant de *tmp*), puis de libérer l'espace mémoire *tmp*.

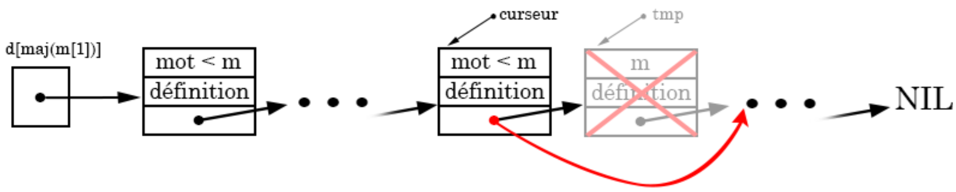

FIGURE 2.13 – Le cas général

Il faudra aussi faire attention à un autre cas particulier : celui où le mot m n'est pas présent dans la liste. Il sera donc nécessaire de tester cette éventualité.

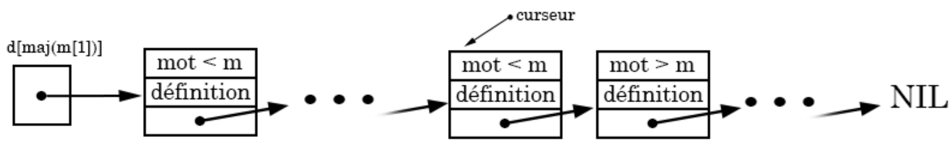

FIGURE 2.14 – Lorsque m n'est pas dans la liste, on ne fait rien

Résumons cela. On positionne un curseur. Tout d'abord, si le curseur pointe sur NIL, on ne fait rien. Maintenant, si le mot du curseur est m, on supprime la tête de liste. Sinon, tant que le champ suivant du curseur ne pointe pas sur NIL et que le mot du champ suivant du curseur est inférieur strictement à m, on avance. Si l'on s'est arrêté parce que le mot du champ suivant du curseur est égal à m, alors on supprime le *Noeud* contenant m.

On peut donc en déduire l'algorithme :

Procédure supprimer (**E/S** d : Dictionnaire, **E** m : **Chaîne)**

Var tmp, curseur : ListeChaineeMD
Début
curseur ← d[maj(m[1])]
Si (curseur ≠ NIL) **alors**
Si (curseur^.mot = m) **alors**
d[maj(m[1])] ← curseur^.suivant
libérer(curseur)
Sinon
Tant que ((curseur^.suivant ≠ NIL) et (curseur^.suivant^.mot < m)) **faire**
curseur ← curseur^.suivant
FinTantQue
Si ((curseur^.suivant ≠ NIL) et (curseur^.suivant^.mot = m)) **alors**
tmp ← curseur^.suivant
curseur^.suivant ← tmp^.suivant
libérer(tmp)
FinSi
FinSi
FinSi
Fin

2.2.5 La loi de composition interne fusionner

On cherche ici à fusionner deux dictionnaires « compatibles » en prenant soin de ne pas laisser de doublons. La première approche (la plus évidente), consiste à insérer tous les éléments des deux dictionnaires dans un seul dictionnaire. On se donne donc une variable res de type Dictionnaire. Pour chaque liste des deux dictionnaires, on insère tous les éléments de ces listes dans res.

Il nous faut ici créer une procédure d'insertion d'une ListeChaineeMD dans un dictionnaire :

Procédure ajouterListeMD (**E/S** d : Dictionnaire, **E** l : ListeChaineeMD)
Début
Tant que (l ≠ NIL) **faire**
inserer(d, l^.mot, l^.definition)
l ← l^.suivant
FinTantQue
Fin

La fonction fusionner devient alors :

Fonction fusionner (d1, d2 : Dictionnaire) **:** Dictionnaire
Var res : Dictionnaire, i : **Caractère**
Début
res ← creerDictionnaire()
Pour i ←'A' **à** 'Z' **faire**
ajouterListeMD(res, d1[i])
ajouterListeMD(res, d2[i])
FinPour
retourner (res)
Fin

On remarque ici que la complexité de la procédure *ajouterListeMD* est très élevée : en effet, à chaque itération de la boucle *tant que* on utilise l'opération *inserer* qui est en O(n). Autrement dit, comme la boucle *tant que* parcoure toute la liste chaînée, on est globalement en O(n^2), ce qui est relativement mauvais. On se propose donc de travailler directement sur les listes sans passer par l'opération *inserer*, pour retrouver une complexité en $O(n)$.

Nous allons créer une fonction de fusion de deux ListeChaineeMD que nous appliquerons aux listes des deux dictionnaires à fusionner. On se donne donc deux ListeChaineeMD $l1$ et $l2$ et une variable res pour stocker le résultat. On place des curseurs : $curseur1$ sur $l1$, $curseur2$ sur $l2$ et $curseur$ sur res. On allouera au fur et à mesure un espace mémoire *tmp* dans lequel on chargera le couple mot/définition en cours de traitement, que l'on ira chaîner sur le $curseur$ de res que l'on fera ensuite avancer.

On commence par remarquer un premier cas particulier : celui où $l1$ et $l2$ sont égales à NIL. Le résultat devra donc être NIL.

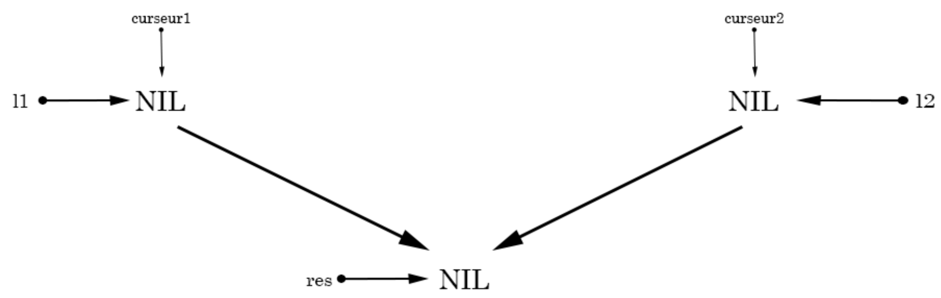

Figure 2.15 – Un premier cas particulier

Ensuite, il faut aussi traiter le cas où seulement l'une des deux listes est remplie. Si $l1$ est remplie et pas $l2$ par exemple, il suffit d'allouer un par un les éléments de $l1$ dans res, en utilisant le $curseur1$ pour se balader sur la liste chaînée.

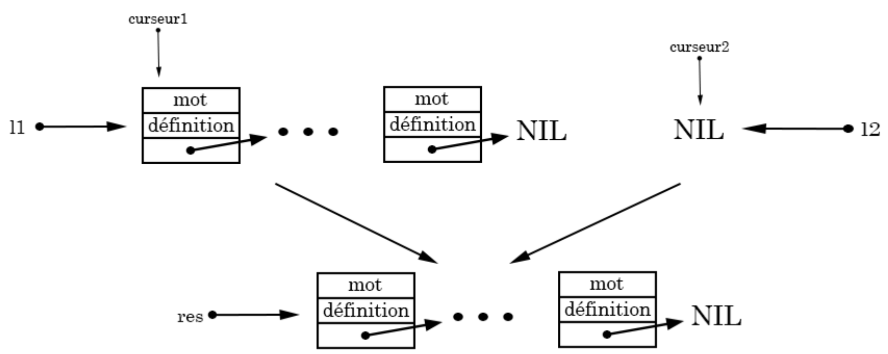

Figure 2.16 – Un second cas particulier

Si les deux listes sont remplies, alors il faudra comparer leurs éléments au fur et à mesure pour avoir une liste res ordonnée. Si l'élément de $l1$ est inférieur à celui de $l2$, il faudra le mettre avant dans res puis avancer le $curseur1$. Si le mot rencontré après dans $l1$ est supérieur à celui de $l2$, alors il faudra copier dans res le mot de $l2$ et faire avancer le $curseur2$... Ainsi de suite jusqu'à ce que l'on arrive à un des deux cas terminaux précédents (une des deux listes (ou les deux) est égale à NIL).

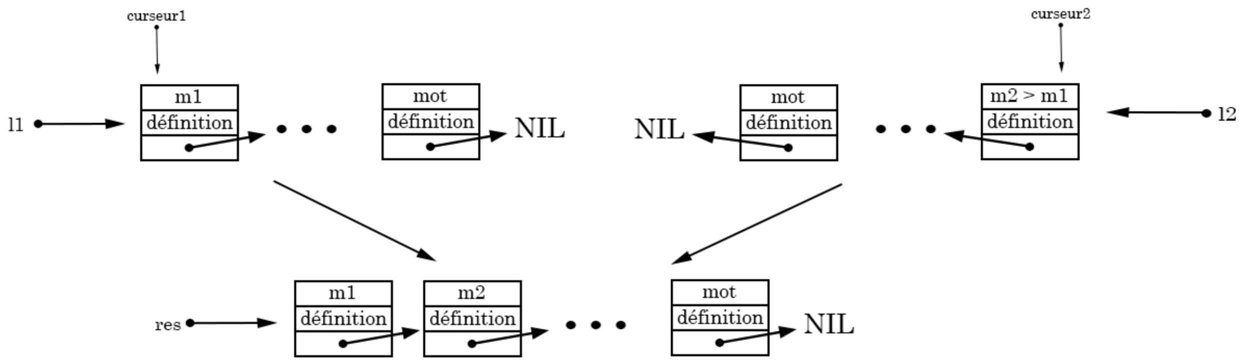

FIGURE 2.17 – Le cas général

Attention, il nous faudra aussi prendre en compte le cas où les deux éléments comparés sont égaux : pour ne pas le copier deux fois à la suite, on déplacera $curseur1$ et $curseur2$ en même temps.

Fonction fusionnerListeMD (l1, l2 : ListeChaineeMD) **:** ListeChaineeMD
    Var res, tmp, curseur : ListeChaineeMD
Début
    res ← NIL
    curseur1 ← l1
    curseur2 ← l2
    Tant que (curseur1 ≠ NIL) et (curseur2 ≠ NIL) **faire**
        tmp ← **allouer**(Noeud)
        Si (curseur1^.mot ≤ curseur2^.mot) **alors**
            Si (curseur1^.mot = curseur2^.mot) **alors**
                curseur2 ← curseur2^.suivant
            FinSi
            tmp^.mot ← curseur1^.mot
            tmp^.definition ← curseur1^.definition
            curseur1 ← curseur1^.suivant
        Sinon
            tmp^.mot ← curseur2^.mot
            tmp^.definition ← curseur2^.definition
            curseur2 ← curseur2^.suivant
        FinSi
        Si (res = NIL) **alors**
            res ← tmp

```
            curseur ← tmp
        Sinon
            curseur^.suivant ← tmp
            curseur ← tmp
        FinSi
    FinTantQue
    Si (curseur1 = NIL) alors
        Tant que (curseur2 ≠ NIL) faire
            tmp ← allouer(Noeud)
            tmp^.mot ← curseur2^.mot
            tmp^.definition ← curseur2^.definition
            Si (res = NIL) alors
                res ← tmp
                curseur ← tmp
            Sinon
                curseur^.suivant ← tmp
                curseur ← tmp
            FinSi
        FinTantQue
    Sinon
        Tant que (curseur1 ≠ NIL) faire
            tmp ← allouer(Noeud)
            tmp^.mot ← curseur1^.mot
            tmp^.definition ← curseur1^.definition
            Si (res = NIL) alors
                res ← tmp
                curseur ← tmp
            Sinon
                curseur^.suivant ← tmp
                curseur ← tmp
            FinSi
        FinTantQue
    FinSi
    Si (res ≠ NIL) alors
        curseur^.suivant ← NIL
    FinSi
    retourner (res)
Fin

Fonction fusionner (d1, d2 : Dictionnaire) : Dictionnaire
    Var   res : Dictionnaire, i : Caractère
Début
    res ← creerDictionnaire()
    Pour i ←'A' à 'Z' faire
        res[i] ← fusionnerListeMD(d1[i], d2[i])
    FinPour
    retourner (res)
Fin
```

2.3 Complexité

Avec les listes, on a les complexités suivantes :

creerDictionnaire	estPresent	obtenirDefinition	inserer	supprimer	fusionner
$\Omega(1)$	$\Omega(1)$	$\Omega(1)$	$\Omega(1)$	$\Omega(1)$	$\Omega(1)$
O(1)	O(n)	O(n)	O(n)	O(n)	O(n)

FIGURE 2.18 – Complexité des opérations

On ne parle pas de la complexité moyenne Θ (trop compliquée à calculer). Toutefois, on peut noter que l'on s'est arrangé pour que celle-ci soit relativement faible (en utilisant notamment le fait que la liste soit ordonnée).

2.4 Bilan

Globalement, cette structure de donnée fonctionne parfaitement et reste relativement simple à manipuler. Toutefois, la complexité des opérations se révèle assez mauvaise (dans le pire des cas). Sur un appareil limité en mémoire vive et en « puissance brute » (on pourrait imaginer un petit appareil éléctronique faisant office de dictionnaire), on atteindrait rapidement les limites de cette structure de donnée.

Conclusion

Ce projet, d'une durée relativement limitée, nous aura néanmoins enseigné de nombreuses choses :

- Une meilleur compréhension de la notion de type abstrait de données.
- Une maîtrise plus poussée des listes chaînées, une plus grande maitrise du cours.
- De nouvelles connaissances suites aux recherches que nous avons réalisées dans le but de trouver des structures de données plus efficaces.
- Une opportunité pour nous perfectionner en langage C en implémentant les algorithmes de ce projet pour s'assurer de leur bon fonctionnement.

On aura ainsi pu mieux saisir l'importance (et même la nécessité) de la conception préliminaire et détaillée avant le dévellopement de n'importe quelle structure de données informatique.

Bibliographie

[1] *Nathalie Chaignaud*, *Algorithmes et structures de données*, Institut National des Sciences Appliquées de Rouen, département GM, 2013.

[2] *Nicolas Delestre*, *Algorithmique et base de la programmation*, Institut National des Sciences Appliquées de Rouen, département ASI, 2013.