



| | |
|---|----|
| 1 - Principe..... | 2 |
| Hachage..... | 2 |
| Collisions..... | 3 |
| 2 - Une fonction d'adressage très recommandable..... | 3 |
| 3 - Les éléments des listes..... | 4 |
| 4 - Un tableau de listes..... | 4 |
| Nombre d'éléments..... | 4 |
| Nature des éléments..... | 4 |
| Création du tableau..... | 6 |
| Utilisation de l'adressage dispersé..... | 6 |
| Destruction du tableau..... | 7 |
| 5 - Gestion des listes..... | 7 |
| Constructeur par défaut..... | 8 |
| Constructeur avec initialisation paramétrée de la clé..... | 8 |
| Recherche d'une donnée..... | 8 |
| Ajout d'une observation..... | 9 |
| Parcours d'une liste..... | 9 |
| Destruction d'une liste..... | 9 |
| 6 - Bon, c'est gentil tout ça, mais ça fait quand même 9 pages. Qu'est-ce que je dois vraiment en retenir ?..... | 10 |

Il arrive fréquemment qu'un programme soit confronté à un problème qui peut être décrit en termes d'associations clé/valeur. Un exemple simple de cette situation est le dénombrement des occurrences des mots d'un texte : il s'agit d'une part d'établir la liste des mots employés par l'auteur (le "lexique" du texte) et, d'autre part, de compter combien de fois chacun d'entre eux apparaît. Dans cette situation, chacun des mots du lexique constitue une clé à laquelle se trouve associée une valeur (le nombre d'occurrences du mot observées dans le texte). Ce problème présente deux caractéristiques remarquables :

- Le nombre de mots différents employés par l'auteur, c'est à dire le nombre de clés, est imprévisible.
- Pour chaque mot du texte, il va falloir déterminer s'il fait ou non déjà partie du lexique en cours d'élaboration, c'est à dire effectuer une recherche. Or le nombre de mots du texte est potentiellement très grand (plusieurs millions).

La première de ces caractéristiques suggère l'utilisation d'une structure de données très dynamique, telle qu'une liste chaînée ou un arbre, par exemple. La seconde caractéristique, en revanche, appelle une organisation des données qui permette aux très nombreuses opérations de recherche de s'effectuer rapidement

Que diriez-vous d'une structure de données dans laquelle les opérations d'insertion seraient aussi simples et rapides que dans une liste chaînée, et qui offrirait en même temps une méthode de recherche encore plus efficace que la recherche dichotomique dans un tableau, sans même exiger en échange un tri préalable des données ?

Cela semble trop beau pour être vrai ? Il est pourtant possible d'obtenir ce genre de performances, et le plus surprenant est que la mise en oeuvre de la technique nécessaire s'avère en fait plus simple que celle d'un arbre ou même d'une recherche dichotomique dans un tableau...

1 - Principe

Les listes chaînées offrent une souplesse inégalée lorsqu'il s'agit d'y ajouter de nouveaux éléments ou d'en supprimer certains. Si l'écriture des quelques fonctions nécessaires à la gestion d'une liste peut difficilement être considérée comme présentant une réelle difficulté, il n'en reste pas moins que, dès que le nombre de données s'accroît, la recherche dans une liste (même triée) devient d'une telle lenteur que cette façon d'organiser les données cesse d'être attractive si de nombreuses recherches doivent être effectuées.

Hachage

Il existe un moyen très simple pour éviter qu'une liste ne s'allonge démesurément : il suffit de répartir les données à stocker entre *plusieurs* listes... Bien entendu, pour que cette fragmentation de la liste¹ présente un avantage réel, il ne faut pas que la recherche consiste à parcourir toutes les listes jusqu'à ce qu'on ait trouvé l'élément recherché. Il faut au contraire que nous disposions d'un moyen infaillible permettant, sans avoir à examiner les listes et leur contenu, de déterminer dans *quelle* liste doit se trouver un élément donné. C'est la mise en place d'un tel moyen, sous la forme d'une **fonction d'adressage** (ou *fonction de hachage*), qui caractérise la technique de l'adressage dispersé.

Si nous utilisons plusieurs listes, il semble assez naturel d'organiser ces listes en un tableau, ce qui permettra d'accéder à une liste donnée à partir de l'index correspondant. Le rôle de la fonction d'adressage est donc très clair : elle doit calculer un index à partir de la clé qui lui est confiée. Etant donnée une clé, la fonction d'adressage désignera donc une liste, et c'est cette seule liste qui aura à être explorée pour y chercher la clé. Si celle-ci n'y figure pas, c'est également dans cette liste qu'un nouvel élément devra être créé pour la recevoir.

Pour que cette méthode soit efficace, il faut bien entendu que le nombre de listes disponibles ne soit pas dérisoire face au nombre de clés différentes, et il faut surtout que la fonction d'adressage produise effectivement le plus de valeurs d'index différentes possibles (en essayant, de plus, d'associer à peu près le même nombre de clés à chaque valeur d'index). Si ces contraintes ne sont pas respectées, le hachage est inefficace : il laisse subsister des "gros morceaux", c'est à dire des cas où l'opération de recherche va être confrontée à l'exploration d'une liste chaînée très longue.

Dans une situation où 1 000 clés doivent être ventilées dans 100 listes, la fonction de hachage idéale constitue 100 listes de 10 éléments, c'est à dire que chaque valeur index est

¹ C'est ce morcellement de la liste qui justifie l'utilisation du terme "hachage", souvent employé informellement pour désigner les techniques d'adressage dispersé.

obtenue à partir d'exactly 10 clés différentes. Aucune recherche ne rencontre alors de liste de plus de 10 éléments. Si la fonction de hachage ne produit que deux index différents, il est évident que nous avons, dans le meilleur des cas, à explorer l'une de deux listes de 500 éléments. Si, de plus, les fréquences des deux valeurs d'index ne sont pas égales, l'une des deux listes sera encore plus longue. Même une fonction de hachage produisant effectivement les 100 valeurs d'index autorisées peut ainsi s'avérer totalement inefficace : si 99 index ne sont produits chacun qu'en réponse à une seule clé, les 901 autres clés produisent toutes le 100^e index, ce qui veut dire que, 9 fois sur 10, la recherche s'effectue sur une liste comportant 901 éléments...

Lorsque la fonction de hachage n'est pas adéquate, les performances de l'adressage dispersé ont tendance à rejoindre celles (désastreuses) d'une simple liste chaînée.

Collisions

Si la fonction de hachage a de bonnes propriétés, en revanche, les performances obtenues resteront correctes même si le nombre de listes utilisées est inférieur au nombre de clés différentes rencontrées. Lorsque la fonction d'adressage produit la même valeur pour différentes clés (ce qui est inévitable s'il y a plus de clés que de valeurs autorisées pour l'index), on dit qu'il y a **collision**. Ce sont les collisions qui obligent à faire grandir les chaînes qu'elles concernent.

En toute rigueur, la méthode décrite ici devrait être baptisée "adressage dispersé avec résolution des collisions par chaînage externe". Il existe en effet d'autres techniques pour traiter les collisions, mais elles ne présentent pas d'avantages évidents dans un contexte C++, qui est très favorable à l'utilisation de pointeurs et à l'allocation dynamique de la mémoire.

2 - Une fonction d'adressage très recommandable

Le choix d'une fonction de hachage dépend, bien entendu, de la nature des clés qui doivent être traitées.

Dans le cas des clés numériques, une méthode très générale est la simple application de l'opérateur **modulo**, qui permet de ramener la valeur de la clé dans l'intervalle admissible pour les index. Bien entendu, cette technique génère des collisions, puisque l'ajout de `tailleTable` à une clé produit une autre clé qui correspondra au même index.

```
1 int calculeIndex(int cle, int tailleTable)
2 {
3     return cle % tailleTable;
4 }
```

Si les clés rencontrées ont des propriétés particulières, connues à l'avance, on sera conduit à adapter la fonction d'adressage de façon à prendre en compte ces propriétés.

Cette approche peut être généralisée au cas où les clés sont des chaînes de caractères d'une longueur quelconque. Rien n'empêche, en effet, de considérer chacun des caractères de la clé comme une valeur utilisable pour un calcul arithmétique tel que celui-ci :

```
1 int calculeIndex(const unsigned char *cle, int tailleTable)
2 {
3     int index = 0;
4     int p;
5     for(p=0 ; cle[p] != 0; ++p)
6         index = (256 * index + cle[p]) % tailleTable;
7     return index ;
8 }
```

Cette fonction est une simple adaptation de celle proposée par Sedgewick, R. **Algorithmes en langage C**. InterEditions. 1991.

3 - Les éléments des listes

Comme nous l'avons vu dans la Leçon 10, il est facile de réaliser des listes chaînées à l'aide d'instances d'une classe : il suffit que celle-ci comporte, en plus des membres destinés à stocker les données, un membre de type "pointeur sur une instance". Dans le cas de l'exemple évoqué plus haut, on pourrait avoir :

```
1 class CElement
2 {
3 protected:
4     CHAINE cle;           //pour stocker le mot
5     CElement * suivant; //pour pouvoir faire la chaîne
6     int valeur;         //pour stocker le nombre d'occurrences du mot
7 };
```

On suppose ici la disponibilité d'un type CHAINE permettant de stocker du texte. Il peut s'agir de la classe CString si vous utilisez les MFC, de la classe string si vous utilisez la librairie standard, ou même d'une classe de votre propre conception. Bien entendu, la classe CElement n'est pas utilisable telle-quelle : nous serons, dans les paragraphes suivants, conduits à la doter de quelques fonctions membre.

4 - Un tableau de listes

La création du tableau de listes soulève deux questions : Quelle doit être sa taille ? Quel est le type de ses éléments ?

Nombre d'éléments

Pour assurer des performances optimales à la fonction de hachage, il a été démontré que

Le nombre de listes disponibles doit être un nombre premier.

Les nombres 997, 4 999, 9 973, 19 997, 29 989, 39 989, 49 999 et 65 521 sont donc quelques-unes des valeurs utilisables. Une liste plus complète est consultable sur <http://www.geocities.com/ResearchTriangle/Thinktank/2434/prime/primenumbers.html>

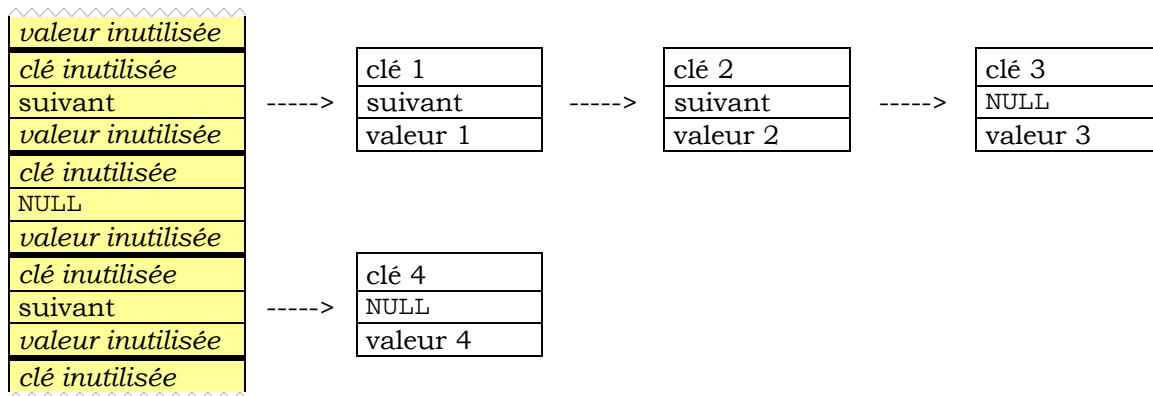
Pour ce qui est de l'ordre de grandeur, diverses considérations interviennent, liées notamment à la quantité de mémoire disponible et à l'importance stratégique des performances obtenues grâce à la dispersion de l'adressage. Une valeur "moyenne" souvent conseillée est le dixième du nombre de clés différentes que l'on s'attend a priori à rencontrer, ce qui donne donc une longueur anticipée d'environ dix éléments pour chaque liste.

Soulignons, une fois encore, que le choix de cette valeur n'est pas absolument critique : les performances se dégradent progressivement, elles ne s'écroulent pas brutalement si le tableau est trop petit.

Nature des éléments

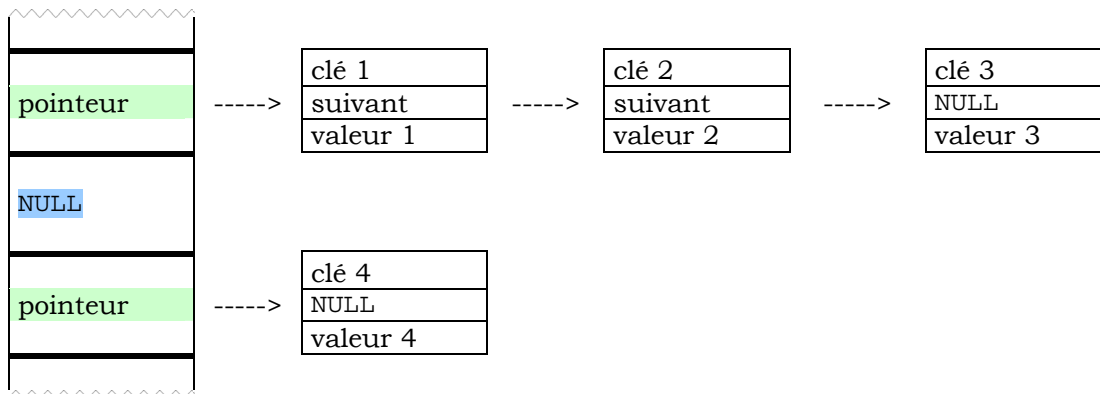
Chacun des éléments du tableau est le point d'entrée d'une liste chaînée. Comme nous l'avons vu au cours de la Leçon 10, le point d'entrée d'une liste peut être soit une instance de la classe CElement, soit un simple pointeur sur un CElement.

Nous avons jusqu'à présent choisi de représenter les listes vides en "sacrifiant" une instance de la classe CElement. Cette instance sert de point d'entrée dans la liste et, lorsque son membre suivant est NULL, elle concrétise la notion (un peu paradoxale) de "liste vide". Cette façon de procéder simplifie légèrement l'écriture des fonctions de gestion de la liste, mais elle conduit à créer une instance de la classe CElement qui ne contiendra jamais de données. Lorsqu'un programme n'utilise que quelques listes, le gaspillage de mémoire occasionné par ce sacrifice ne mérite même pas qu'on signale qu'il est dérisoire. Un programme qui utilise l'adressage dispersé risque, pour sa part, d'être conduit à mettre en place plusieurs dizaines de milliers de listes. Le **gaspillage** occasionné par la création d'un **tableau** de dizaines de milliers de CElement qui ne contiendront jamais de données peut donc cesser d'être négligeable.



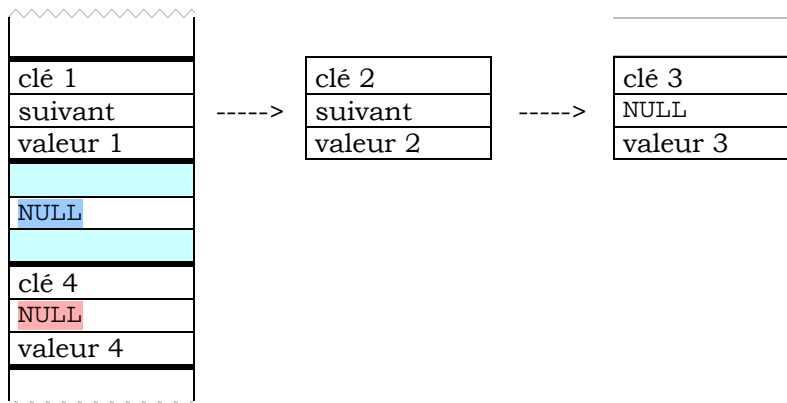
Stockage imaginaire des informations concernant quatre couples clé/valeur dans une structure à adressage dispersé où les listes vides sont représentées par des CElement dont le membre suivant est NULL

Créer un tableau de "pointeur sur CElement" réduirait considérablement ce gaspillage, puisqu'il ne serait plus nécessaire de réserver de la place pour des membres "clé" et "valeur" dans le tableau.



Stockage imaginaire des informations concernant quatre couples clé/valeur dans une structure à adressage dispersé où les listes vides sont représentées par des pointeurs sur CElement dont la valeur est NULL

Si cette façon de procéder est très efficace pour représenter les **listes vides**, il faut quand même rappeler que, par hypothèse, notre fonction d'adressage va faire en sorte qu'il y ait le moins possible de listes vides. Or, lorsque la liste n'est pas vide, le **pointeur stocké dans le tableau** apparaît non seulement comme une perte de place, mais également comme une indirection supplémentaire qui ralonge artificiellement la liste et ralentit donc l'exploration de celle-ci. En définitive, il semble clair qu'aucun des deux moyens envisagés lors de la Leçon 10 ne représente de façon optimale une liste vide dont l'entrée est un élément d'un tableau de hachage. La meilleure solution est, dans ce cas, de constituer un tableau de CElement *pouvant être utilisés* pour stocker des données :



Stockage imaginaire des informations concernant quatre couples clé/valeur dans une structure à adressage dispersé où les listes vides sont représentées par des CElement dont la clé ou la valeur ont une **valeur particulière**

Cette façon de procéder soulève cependant une légère difficulté : comme on le voit bien sur la représentation graphique ci-dessus, la présence d'une valeur NULL dans le membre suivant d'un élément du tableau ne signifie plus que la liste correspondante est **vide**. En effet, cette

valeur `NULL` apparaît également lorsque la liste est réduite à un seul élément. Il est donc nécessaire de convenir d'un autre moyen pour indiquer sans ambiguïté qu'une liste est vide. Dans notre exemple, il est facile d'utiliser à cet effet une valeur particulière pour le membre "clé" (la chaîne vide, par exemple) ou pour le membre valeur (-1, par exemple), puisque de telles valeurs ne peuvent en aucun cas être rencontrées dans le cas d'un couple clé/valeur réel.

Dans un cas où aucune valeur "naturellement impossible" ne pourrait être utilisée pour signaler les listes vides, il faudrait se résoudre à ajouter un membre à la classe dont les éléments du tableau sont des instances, quitte à devoir utiliser des classes différentes pour les éléments du tableau et pour les éléments "hors tableau" des listes chaînées.

Création du tableau

Etant donné sa taille, le tableau de listes sera normalement un faux tableau :

```
1 CElement * table = new CElement [TAILLE_TABLE];
2 if (table == NULL)
    //prendre ici les mesures nécessaires
```

Utilisation de l'adressage dispersé

Dans un cas tel que celui que nous avons envisagé à titre d'exemple, l'insertion d'information dans la structure de données à adressage dispersé reposera sur une séquence du genre :

```
1 index = calculeIndex(leMot, TAILLE_TABLE);
2 table[index].ajouteOccurrence(leMot);
```

On pourrait penser que le calcul de l'index a vocation à être pris en charge par une classe "enveloppant" la gestion de l'adressage dispersé. C'est certainement une option possible, mais, dans bien des cas, il est nécessaire de procéder à une analyse caractère par caractère des données, de façon à repérer où commencent et où finissent les mots. Le calcul de l'index n'est pas très difficile à incorporer dans ce traitement, ce qui évite d'avoir à parcourir une seconde fois le texte (potentiellement très long, rappelons-le) caractère par caractère.

Une fois les informations recueillies dans la structure à adressage dispersé, leur exploitation exige évidemment de calculer l'index correspondant à la liste où les données relatives au mot concerné sont (éventuellement) stockées :

```
1 index = calculeIndex(leMot, TAILLE_TABLE);
2 nb = table[index].nombreOccurrences(leMot);
```

La classe `CElement` devra donc comporter de fonctions membre `ajouteOccurrence()` et `nombreOccurrences()` capables de répondre à cet usage.

Ces deux fonctions ne sont, nous allons le voir, que de simples gestionnaires de liste chaînées, tout à fait analogues à ceux présentés dans la Leçon 10. Le fait qu'elles seront exécutées sur des listes faisant partie d'une structure à adressage dispersé ne les concerne aucunement, et ne complique donc en rien leur mise au point.

Il arrive fréquemment que le besoin d'accès aux données ne se limite pas à la recherche d'une entrée particulière, mais implique le parcours exhaustif de la structure. C'est, par exemple, le cas lorsqu'on souhaite enregistrer dans un fichier les données contenues dans une structure à adressage dispersé. Un parcours complet de la structure de données implique évidemment un parcours du tableau et, pour chaque élément de celui-ci, un parcours de la liste dont il contient éventuellement le premier élément. Il suppose, par ailleurs, que la classe `CElement` propose une fonction permettant d'accéder à la clé et à la valeur d'une instance donnée, ainsi que d'un moyen d'accéder à l'élément suivant un élément donné. L'ensemble de ces fonctionnalités peut être fourni par une même fonction membre, dont le fragment de code suivant suppose qu'elle se nomme `contenu()`, qu'elle renvoie l'adresse de l'élément suivant celui au titre duquel elle est invoquée, et qu'elle dispose de deux paramètres de type référence dans lesquels elle va respectivement placer la `clé` et la `valeur` de l'élément au titre duquel elle est invoquée.

Ce fragment de code suppose également que `fichier` est une variable de type `ofstream` en état de recevoir des données via l'opérateur d'insertion.

```
1 int i;
2 int nb;
3 CHAINE forme;
4 CElement * aAfficher;
5 for (i=0 ; i < TAILLE_TABLE ; ++i)
6 {
7     aAfficher = m_table + i; //autrement dit : aAfficher = & (m_table[i]);
8     while (aAfficher != NULL)
9     {
10        aAfficher = aAfficher->contenu(forme, nb);
11        if (forme != "")
12            fichier << forme << "\t" << nb << endl;
13    }
14 }
```

Sans présenter une difficulté insurmontable du point de vue programmatore, le parcours exhaustif reste toutefois le talon d'Achille des structures de données à adressage dispersé, surtout lorsque les collisions sont résolues par chaînage externe. L'allocation dynamique des éléments peut, en effet, les conduire à être très éparpillés en mémoire, ce qui peut se traduire par une dégradation des performances lors de traitements exigeant de nombreux accès à toutes les données. Dans de telles circonstances, un simple tableau peut parfois s'avérer plus efficace.

Destruction du tableau

Une fois devenue inutile, la structure de données à adressage dispersé sera détruite, en n'oubliant pas de commencer par détruire toutes les listes chaînées qui ont leur origine dans le tableau. Cette description suppose que la classe CElement soit également dotée d'une fonction `detruiListe()`, capable de libérer la mémoire occupée par les éléments de la liste qui ne figurent pas dans le tableau.

```
1 int n;
2 for(n = 0 ; n < TAILLE_TABLE ; ++n)
3     table[n].detruiListe();
4 delete[] table;
```

Nous savons, que, lors de la **destruction du tableau**, la libération de la mémoire est précédée d'un appel au destructeur de la classe CElement au titre de chacun des éléments du tableau. Il pourrait être tentant de définir explicitement ce destructeur pour qu'il s'occupe automatiquement de la destruction des CEelement constituant la liste à laquelle le CEelement faisant partie du tableau permet d'accéder. Ceci permettrait effectivement de détruire l'ensemble de la structure de données à adressage dispersé en **une seule instruction**. En revanche, cette façon de procéder présenterait un gros inconvénient : il deviendrait impossible de détruire un élément d'une liste sans détruire tous ceux qui le suivent. Or, en situation réelle, il arrive que la destruction d'un élément intervienne dans un contexte autre que la destruction complète de la liste. Nous renoncerons donc à la tentation de définir explicitement le destructeur de CEelement.

5 - Gestion des listes

Comme nous l'avons vu précédemment, le fait que nous soyons confrontés à plusieurs dizaines de milliers de listes ne rend pas le problème de leur gestion différent de celui traité dans la Leçon 10. Toutefois, les conventions utilisées pour représenter les listes vides sont ici un peu différentes, et notre maîtrise du langage nous autorise maintenant une approche un peu moins rudimentaire de ces opérations. Nous allons donc ébaucher la mise au point d'une classe capable de traiter le problème du dénombrement des occurrences des mots d'un texte.

Par rapport aux fonctions analogues présentées au cours de la Leçon 10, plusieurs des fonctions envisagées ici présentent la particularité d'utiliser un appel récursif. Les listes rencontrées dans le cadre de la mise en place d'un adressage dispersé sont, par hypothèse, brèves, et l'allègement de l'écriture obtenu grâce à la récursion ne sera donc pas payé trop cher en termes d'efficacité du programme.

Constructeur par défaut

La création du tableau de hachage implique la création de nombreuses instances de `CElement`. Plutôt que d'avoir à prévoir une boucle pour préparer ces instances au recueil de données², on peut faire en sorte que chaque nouvelle instance naisse avec des valeurs convenables dans ses variables membre. Pour le pointeur suivant, cette valeur est évidemment `NULL`, de façon à indiquer que la nouvelle instance ne désigne pour l'instant aucun successeur. Nous avons déjà souligné le fait cette valeur `NULL` peut plus être interprété comme signifiant que la liste est vide. Il nous faut donc convenir d'un autre moyen pour distinguer les éléments correspondant à une liste vide des éléments correspondant à une liste réduite à ce seul et unique élément. La meilleure solution semble ici de faire en sorte que le membre "clé" ne contienne **aucun texte** lorsque la liste est vide.

Un texte vide dans la clé n'est en effet pas envisageable autrement que dans le cas d'un `CElement` non utilisé : dénombrer les occurrences d'une absence de texte n'a aucun sens.

Puisque la symbolisation de la liste vide est assurée à l'aide du membre clé, la valeur initiale du membre valeur peut être choisie sans avoir à tenir compte de ce problème. Bien que la valeur 0 puisse sembler a priori la plus logique, la réflexion montre en fait qu'il existe une meilleure option. En effet, tant que la clé est vide, la valeur contenue dans le `CElement` est sans importance. Si elle est à 0, ceci exige que, au moment où le `CElement` est affecté au dénombrement des occurrences d'un mot, deux opérations soient effectuées : une copie du mot en question dans le membre `cle`, et le passage du membre `valeur` de 0 à 1. Cette seconde opération peut être rendue inutile en initialisant directement `valeur` à **1**.

```
1 CElement::CElement() : cle(""), suivant(NULL), valeur(1)
2 {
3 //les initialisations sont faites, le bloc de code reste vide
}
```

Il est possible que, dans certains cas, on désire créer des `CElements` affectés à une clé, mais indiquant un nombre d'occurrences nul (par exemple pour gagner du temps en créant à l'avance les `CElements` correspondants à des mots dont on est quasiment certains qu'ils vont apparaître dans le texte traité). Le fait que le constructeur par défaut initialise le nombre d'occurrences à 1 n'est évidemment pas un obstacle à cette approche : on peut procéder par affectation ou, mieux encore, définir un constructeur admettant comme paramètre un entier qui servira à initialiser la variable membre `valeur`.

Constructeur avec initialisation paramétrée de la clé

Les instances de `CElement` qui font partie du tableau de hachage ne seront pas les seules créées : lorsqu'une collision l'exigera, il faudra ajouter des `CElements` aux listes. Dans ce contexte, le membre `cle` doit être **initialisé** avec le mot dont l'apparition dans le texte a créé la collision.

```
1 CElement::CElement(CHAINES mot) : cle(mot), suivant(NULL), valeur(1)
2 {
3 //les initialisations sont faites, le bloc de code reste vide
}
```

Recherche d'une donnée

La recherche de l'élément correspondant à un mot donné est une opération assez simple : il suffit de vérifier si la clé de l'instance au titre de laquelle la fonction est exécutée est le mot recherché. Si c'est le cas, le nombre d'occurrences à renvoyer est contenu dans `valeur`. Dans le cas contraire, ce nombre est celui que renverra la fonction si elle est invoquée au titre de l'élément suivant (sauf, bien entendu, s'il n'y a plus d'éléments : dans ce cas, le nombre d'occurrences du mot recherché est nul, puisqu'on a épuisé la liste dans lequel il doit se trouver sans l'avoir jamais rencontré).

² Notez, cependant, que si la même structure de données doit servir à plusieurs traitements successifs (l'analyse de plusieurs textes, par exemple), il faudra de toutes façons écrire quelque part une boucle de ce genre...

```

1 int CElement::nombreOccurrences(const CHAINE & cible)
2 {
3     if (cle == cible)
4         return valeur;
5     if (suivant != NULL)
6         return suivant->nombreOccurrences (cible); //on refile le bébé au suivant
7     return 0; //le mot n'est pas présent
8 }

```

Ajout d'une observation

La fonction qui enregistre l'occurrence d'un mot est un peu plus complexe que la fonction de recherche d'un nombre d'occurrences, car quatre cas doivent y être pris en compte. Si le mot correspond à la clé de l'instance au titre de laquelle la fonction est exécutée, il suffit d'incrémenter valeur. Si ce n'est pas le cas, deux hypothèses sont possibles : soit l'instance au titre de laquelle la fonction est exécutée est disponible (et il suffit alors de l'utiliser pour dénombrer les occurrences du mot³), soit ce n'est pas le cas. Cette seconde hypothèse se divise elle-même en deux : soit l'instance au titre de laquelle la fonction est exécutée est la dernière de la liste, et il faut ajouter un élément à celle-ci pour dénombrer les occurrences du mot, soit ce n'est pas le cas. Dans cette dernière hypothèse, il suffit de se déclarer incompetent et de soumettre le problème à l'élément suivant...

```

1 void CElement::ajouteOccurrence(const CHAINE & mot)
2 {
3     if (cle == mot)
4         ++valeur; //c'est une occurrence du mot qu'on compte, tout va bien
5     else
6         if (cle == "") //on ne compte encore rien, adoptons ce mot
7             cle = mot; //valeur contient déjà 1, grâce au constructeur par défaut
8         else //on traite déjà un mot différent de celui qu'il faut recenser
9             if (suivant == NULL) //on est en fin de liste
10                {
11                    suivant = new CElement(mot); //on crée un élément chargé de ce mot
12                    if(suivant == NULL)
13                        //prendre ici les mesures nécessaires...
14                }
15            else //on n'est pas en fin de liste
16                suivant->ajouteOccurrence(mot); //on refile le bébé à l'élément suivant
17 }

```

Parcours d'une liste

La fonction contenu(), dont nous avons vu qu'elle était nécessaire au parcours exhaustif de la structure de données à adressage dispersé, peut être définie ainsi :

```

1 CElement * CElement::contenu(CHAINE & forme, int & nb)
2 {
3     forme = cle;
4     nb = valeur;
5     return suivant;
6 }

```

Destruction d'une liste

La destruction des listes doit prendre en compte une de leurs particularités : le premier élément de chacune d'entre-elles fait partie d'un tableau (sans doute faux) et sera détruit lors de destruction de celui-ci. La fonction de destruction des listes doit donc s'abstenir de détruire ce premier élément.

³ Les seules instances de CElement qui peuvent exister sans pour autant être déjà utilisées sont celles qui font partie du tableau. Ce cas n'est donc rencontré que lorsqu'une liste reçoit son premier élément, ce qui veut dire qu'il est inutile alors de chercher à explorer la liste pour vérifier si un de ses éléments n'est pas déjà consacré au recensement des occurrences du mot.

Cette "protection" du premier élément de chaque liste peut être obtenue de deux façons : soit en utilisant une fonction de destruction qui laisse intacte l'instance au titre de laquelle elle est invoquée, soit en invoquant la fonction au titre du second élément de chaque liste. Cette seconde approche exige toutefois de vérifier que ce second élément existe avant d'essayer de l'utiliser pour appeler la fonction, et il est toujours préférable de déplacer le plus de complexité possible dans les fonctions membre, au profit de la simplicité du code qui utilise les instances. La fonction suivante correspond donc à l'adoption de la première méthode :

```
1 void CElement::detruitListe()  
2 {  
3     if (suivant == NULL)  
4         return;  
5     suivant->detruitListe();  
6     delete suivant;  
7 }
```

6 - Bon, c'est gentil tout ça, mais ça fait quand même 9 pages. Qu'est-ce que je dois vraiment en retenir ?

- 1) Il est parfois possible d'obtenir à la fois la souplesse de gestion d'une liste chaînée et des performances en recherche supérieures à celles offertes par un tableau trié.
- 2) La technique d'adressage dispersé repose sur deux éléments fondamentaux : une fonction de hachage et une technique de traitement des collisions.
- 3) Une fonction de hachage effectue un calcul dont les données sont la clé associée à l'élément qui doit être stocké dans la structure à adressage dispersé, et dont le résultat est la position où l'élément en question doit être stocké.
- 4) On dit qu'il y a collision lorsque deux éléments prétendent occuper la même place dans la structure de données (c'est à dire lorsque la fonction de hachage produit le même résultat pour deux clés différentes).
- 5) La création de listes chaînées est un moyen simple et efficace pour gérer les collisions.