Algorithme de compression de Huffman

Par gRRosminet

Table des matières :

Introduction

A l'heure actuelle, si les espaces de stockage ne posent pas de réels problèmes, ce n'est pas le cas des flux de données sur les réseaux. En effet, les réseaux (Lan, Internet …) sont de plus en plus utilisés pour échanger des informations. Cela pose cependant un problème important : celui du temps de transmission. Le débit des réseaux étant limité, on cherche à réduire la taille des données à envoyer afin de rendre la transmission plus rapide et par conséquent plus satisfaisante. Pour cela, il existe deux méthodes.

La première, qui engendre une perte ou une altération de l'information, est utilisée pour tout ce qui concerne l'image et le son. Elle consiste dans un premier temps à enlever toute l'information à laquelle les capteurs humains ne sont pas sensibles (fréquences inaudibles, surfaces insignifiantes …) puis a écrire une nouvelle donnée qui ne pourra plus reprendre son format (qualité) original. Il y a eu suppression de l'information de manière irréversible.

La seconde est dite « lossless », c'est à dire sans perte et elle est appliquée dans tous les domaines ou la perte d'information est inacceptable : exécutables, textes ... Elle consiste à analyser des données et à en déduire une écriture de taille inférieure. L'une de ces méthodes est appelée compression de Huffman (inventée par le mathématicien du même nom) et c'est celle que nous allons étudier ci-après.

Je tiens à préciser que cet article à été écrit grâce à la généreuse participation de GoldenEye qui nous éclaire régulièrement de ses lumières sur notre forum

1) Présentation :

La méthode de compression Huffman consiste à diminuer au maximum le nombre de bits utilisés pour coder un fragment d'information. Prenons l'exemple d'un fichier de texte : Le fragment d'information sera un caractère ou une suite de caractères. Plus le fragment sera grand, plus les possibilités seront grandes et donc la mise en œuvre complexe à exécuter. L'algorithme de Huffman se base sur la fréquence d'apparition d'un fragment pour le coder : plus un fragment est fréquent, moins on utilisera de bits pour le coder. Dans notre exemple de fichier texte, si on considère que notre fragment est la taille d'un caractère, on peut remarquer que les voyelles sont beaucoup plus fréquentes que les consonnes : par exemple la lettre 'e' est largement plus fréquente que la lettre 'x' par conséquent la lettre 'e' sera peut-être codée sur 2 bits alors que la lettre 'x' en prendra 10.

Pour pouvoir compresser puis décompresser l'information, on va donc devoir utiliser une table de fréquences et deux choix s'offrent à nous : calculer une table et l'intégrer au fichier ou utiliser une table générique intégrée dans la fonction de compression. Dans le premier cas, la compression est meilleure puisqu'elle est adaptée au fichier à compresser, mais elle nécessite le calcul d'une table de fréquences et le fichier sera plus important également du fait de la présence de cette table dans le fichier. Dans le second cas, la compression sera plus rapide puisque elle n'aura pas à calculer les fréquences, par contre l'efficacité de la compression sera moindre et le gain obtenu par la première méthode (ratio de compression + taille de la table) peut être supérieur à celui de la deuxième (ratio de compression).

2) Création de la table des fréquences d'apparition des fragments :

Cette table consiste en un comptage empirique des fragments au sein des données à compresser. Reprenons l'exemple d'un texte : nous allons analyser la phrase

« gRRosminet et GoldenEye programment Huffman »

Pour simplifier l'exemple, nous ignorerons la casse :

Lettres	Occurrences	Fréquence
G	3	7,14%
R	3	7,14%
O	3	7,14%
S	1	2,38%
M	4	9,52%
I	1	2,38%
N	4	9,52%

E	6	14,29%
T	3	7,14%
L	1	2,38%
D	1	2,38%
Y	1	2,38%
P	1	2,38%
A	2	4,76%
H	1	2,38%
U	1	2,38%
F	2	4,76%
[ESPACE]	4	9,52%
Total	42	100%

3) Création de l'arbre de Huffman :

L'arbre de Huffman est la structure données qui vas nous permettre de donner un code pour chaque lettre en fonction de sa fréquence.

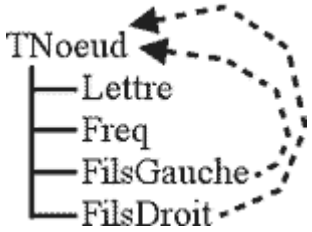
Etape 1

Il faut commencer par trier la liste par ordre croissant de fréquences (vous remarquerez que le tri a été fait sur la fréquence puis sur la lettre ce qui sera important pour permettre la diminution de la taille de l'entete) :

Lettres	Occurrences	Fréquence
D	1	2,38%
H	1	2,38%
I	1	2,38%
L	1	2,38%
P	1	2,38%
S	1	2,38%
U	1	2,38%
Y	1	2,38%
A	2	4,76%
F	2	4,76%
G	3	7,14%
O	3	7,14%
R	3	7,14%

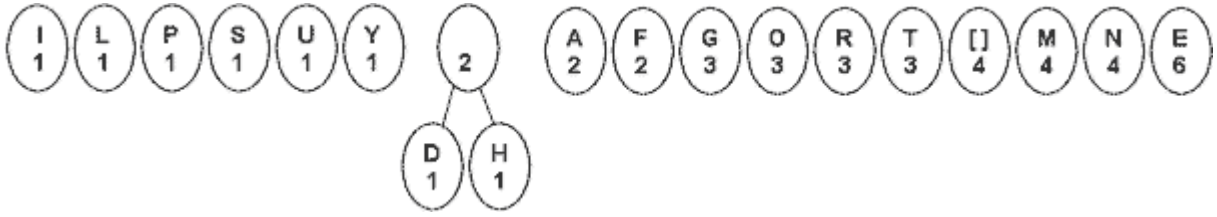
T	3	7,14%
[ESPACE]	4	9,52%
M	4	9,52%
N	4	9,52%
E	6	14,29%

Nous allons maintenant construire un nœud de l'arbre pour chaque fragment et les placer dans une liste ordonnée de noeuds. Un nœud doit avoir une structure telle que ci-contre :

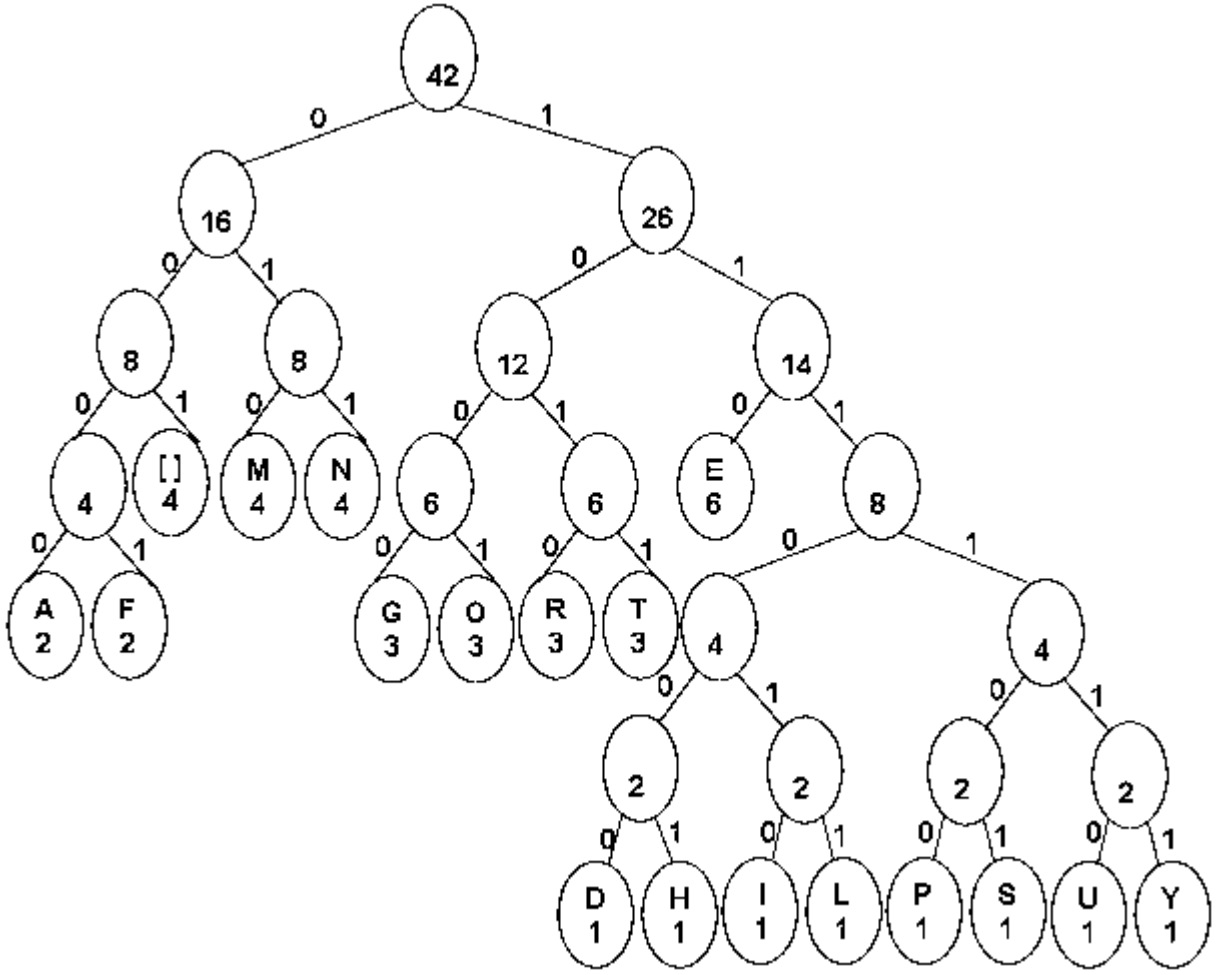


Etape 2

Nous allons maintenant construire l'arbre à partir de la liste ordonnée de nœuds. La construction est très facile : il suffit de prendre les deux nœuds les moins fréquents (D et H) et de les ajouter comme fils d'un nouveau nœud qui aura pour fréquence la somme des deux.



Il suffit de réitérer cette étape jusqu'à ne plus avoir qu'un seul nœud. Après cela, descendre vers la gauche équivaut à un 0, et descendre vers la droite à un 1.



On en déduit le codage suivant :

Lettres	Occurrences	Fréquence	Codage
D	1	6	111000
H	1	6	111001
I	1	6	111010
L	1	6	111011
P	1	6	111100
S	1	6	111101

U	1	6	111110
Y	1	6	111111
A	2	4	0000
F	2	4	0001
G	3	4	1000
O	3	4	1001
R	3	4	1010
T	3	4	1011
[ESPACE]	4	3	001
M	4	3	010
N	4	3	011
E	6	3	110
Total de bits (normal | compressé)	336	166	

La dernière ligne du tableau nous montre qu'une fois compressée, la phrase n'occupe même plus la moitié de l'espace original (hors entête).

4) Compression / Décompression

Compression

Pour la compression des données, on va donc devoir écrir un nouveau fichier contenant deux parties : l'entête et les données compressées. L'entête : il doit contenir le nom original du fichier, la taille originale et une table de correspondance qui permette de reconstituer le fichier original (cf table précédente).

Pour compresser les données, on va donc lire le fichier original fragment par fragment (ici octet par octet) et on écrira le code Huffman correspondant dans le fichier de destination. Attention, les codes de Huffman ne font pas toujours un multiple de 8 bits et par conséquent ils nécessitent d'être mis dans une zone tampon avant d'être écris dans le fichier ! Vous pouvez par exemple utiliser un tableau de 8 caractères qui contienne soit '1' soit '0', mais cela n'est qu'une proposition et l'objectif n'est pas de vous aider à programmer mais de vous faire comprendre le fonctionnement donc pour les plus perdus voyez les applications proposées :-)

Décompression

Pour la décompression, vous lirez l'entête du fichier, puis, pour chaque bit lu, vous descendrez dans votre arbre et à chaque feuille terminale, vous aurrez décodé un caractère original que vous pourrez écrire dans le fichier de restitution :-)

5) Remarques

On définit le ratio de compression :

$$\rho = 1 - \frac{taille\_compressé}{taill\_normale}$$

Ici

$$\rho = 1 - \frac{166}{336} = 50{,}6\%$$

En règle générale, un fichier texte sans pathologie particulière, est compressé par cette méthode suivant un ratio de 30 à 60%.

Sur de très petits fichiers (moins de 1 Ko), l'en-tête est assez volumineux par rapport aux informations compressées. Le fichier réduit a alors une taille supérieure au fichier original…

Sur de gros fichiers, on peut appliquer l'algorithme de Huffman non pas caractère par caractère mais par tranche de 2, 3 ou 4 caractères. Le problème est que la construction de l'arbre prend beaucoup plus de temps (l'arbre est plus gros,il y 65536 doublets possibles de caractères ASCII étendu par exemple). De même, l'en-tête prend une place considérable.

Mise en oeuvre en C++

Algorithme de compression de Huffman en C++

Par gRRosminet

Introduction

Si le principe de la compression de Huffman peut paraitre simple, sa mise en oeuvre n'en n'est pas moins une aventure périlleuse. C'est ce que nous allons voir dans cet aricle. Je tiens a préciser que mon objectif premier à été de réaliser une solution fonctionnelle et qu'elle n'est absolument pas optimisée. Avant de vous plonger dans cet exemple, je vous conseille fortement de jetter un oeil à l'explication de l'algorithme.

Je tiens à préciser que la mise en forme automatique du code de cet article est l'oeuvre de Haypo, un de nos valeureux rédacteurs qui nous éclaire régulièrement sur le forum.

Vous pouvez télécharger le code source complet ici

I. Outils nécessaires

Tout d'abord, essayons d'envisager tous ce dont nous aurrons besoin pour ce développement.

a) Statistiques

Nous aurrons besoin d'une liste de fréquence d'apparition des fragments. Dans notre cas, le fragment est un octet et on peut donc faire un tableau de 256 statistiques sans avoir a se soucier de l'espace mémoire que cela représente puisque c'est plutot restreint.

Le format d'une statistique :
- On a besoin de savoir quel est l'élément : son indice dans le tableau l'indiquera, il n'est donc pas nécessaire de le stocker.
- On a besoin de connaitre la fréquence d'apparition du fragment : les fichiers pouvant faire jusqu'à 4Go, il faut dont utiliser un usigned long int pour compter le nombre d'apparition. La fréquence d'apparition n'est pas nécessaire, on ne la stockera donc pas.
- Finalemment, on aurra besoin de stocker le code de Huffman associé au fragment. J'ai fais le choix d'une chaine de caractères qui a l'avantage

de simplifier le débuggage. Mais le désavantage d'etre plutot lent :-(

Pour calculer les tatistiques, nous aurront besoin d'une fonction qui parcours tous le fichier d'entrée et incrémente les compteurs des fragments.

b) arbre de Huffman

Nous aurrons besoin d'une structue d'arbre binaire dont les noeux prennent n charge la comparaison. nous aurrons également besoin d'une liste triée pour ordonner les noeuds et les morceaux d'arbre je propose d'utiliser la file de priorité de la librairie de templates standards du C++. Le probleme est qu'en général, on utilise des liste de pointeurs pour éviter d'avoir des temps de recopie des éléments trop importants. Je propose donc de faire un objet template qui propose les opérateurs de comparaison et un pointeur vers un type d'objet.

Nous aurrons également besoin d'une fonction qui associe les éléments les plus légers 2 à 2 jusqu'a ce que la liste n'en contienne plus qu'un et une qui détermine le code de huffman de chaque élément.

c) compression/décompression

Nous aurrons besoin d'une fonction qui permette d'écrire l'entete de compression associé au fichier et qui effectue la compression. Cette dernière devra s'assurer que les étapes d'analyse et d'écriture de l'entete ont été réalisées correctement avant de réaliser la compression.

Finallement nous aurrons également besoin de fonctions réalisant l'opération inverse : lecture de l'entete de compression, reconstruction de l'arbre puis décompression.

II. Implémentation de l'objet

Pour des raison de clarté du point de vue du compilateur et afin d'éviter tout enchevêtrement possible, tous les développements seront fait dans le namespace Huffman qui contiendra donc tous les outils nécessaires à ce développement et dont l'utilisation d'ans d'autres projets n'est pas envisageable.

a) Définition de la classe

La classe contiendre des informations sur les fichiers mis en jeux, ainsi que des pointeurs vers les éléments de compression qui seront eux alloués dynamiquement. J'ai également inclu des pointeurs de fonctions permettant de faire communiquer l'objet avec l'application principale et que celle ci puisse afficher des information à l'utilisateur final. Toutes les fonctions nécessaires dont nous avons énoncé les propriétés ci-avant ont été prévues et elles sont acompagnées des fonctions nécessaires à la gestion des messages utilisateurs.

J'ajouterai que si vous pretez un peu attention au code, vous remarquerez que le type chois pour le stockage du nombre d'éléments est un

unsigned short int et non pas un unsigned char alors qu'on va compter des éléments de type char. Cela est du au fait qu'on peut n'avoir aucun élément ou n ou 256 et que 256 est strictement égal à 0 par le système d'overflow ce qui pose quelques problèmes par la suite :-(

```
class THuffman
{
   private :
      bool bCompresse;                    // Compression réalisée (O/N)
      std::string sSource;                // Fichier source
      std::string sDestination;           // Fichier de destination
      TNoeudHuffman * ArbreHuffman;       // Arbre de Huffman
      unsigned short int NombreElements;       // Nombre d'éléments différents
      unsigned long int TailleReelle;     // Taille du fichier originale
                                          // lue dans l'entête du fihier
                                          // source.
      unsigned long long int FinEntete;        // Position à laquelle continuer
                                          // la lecture pour décompresser
      TStatistiqueHuffman * Statistiques; // Statistiques du fichier source
      TDegreDialogue DegreVerbalite;      // Degre de parole de la classe
      TFoncMessage FoncAffichageProgression;// Fonction pour affichage de la
                                          // progression des différentes
                                          // fonctionnalités de la classe
      TFoncMessage FoncAffichageEtape;    // Idem pour les etapes
      unsigned int ErrCode;               // Code de la dernière erreur
                                          // survenue
      std::string sErrString;             // Message associé à la dernière
                                          // erreur survenue

   protected :
      virtual bool EcrireEntete();        // Ecrit l'entête dans le fichier
                                          // de destination
      virtual bool LireEntete();          // Lit l'entête à partir du
                                          // fichier source
      bool ConstruireArbreFreq();         // Construit l'arbre de Huffman à
                                          // partir des frequences
                                          // ==> déduire le code
      bool ConstruireArbreCode();         // Construit l'arbre de Huffman à
                                          // partir des codes
                                          // ==> décompression

   public :
      // Constructeur / Destructeur
```

```
        explicit THuffman(const std::string &sSrc = std::string(""),
                          const std::string &sDst = std::string(""),
                          TDegreDialogue Dialogue = ddBavard,
                          const TFoncMessage Progres = NULL,
                          const TFoncMessage Etape = NULL);
        virtual ~THuffman();

        virtual bool Analyser();              // Analyse le fichier source et
                                              // alimente les statistiques
        bool Compresser();                    // Compresser le fichier source
        bool Decompresser();                  // Décompresser le fichier source

        unsigned long int TailleInitiale();
        unsigned long int TailleCompresse();  // Disponible uniquement après
                                              // l'analyse
        unsigned long int TailleFinale();     // Disponible uniquement après
                                              // la compression

        void SetSource(const std::string &sSrc);
        void SetDestination(const std::string &sDst);
        void SetDegreDialogue(const TDegreDialogue v);
        void SetFonctionProgression(TFoncMessage Progres);
        void SetFonctionEtape(TFoncMessage Etape);

        std::string GetSource() const;
        std::string GetDestination() const;
        TDegreDialogue GetDegreDialogue() const;

        unsigned int GetDerniereErreur(std::string &sMessage) const;

        // Fonction par défaut pour l'affichage des messages.
        static void AfficherMessage(const std::string &sMessage);
    };
```

b) Définition des outils connexes

Comme vous avez pu le voir, j'utilise des types qui ne sont pas standards, je les ai définis un peut plus haut dans mon entête.

- Commençont par TDegreDialogue : c'est un type énuméré utilisé pour la gestion des message d'information de l'utilisateur.

```
typedef enum {ddMuet = 0, ddProgression = 1, ddEtape = 2, ddBavard = 3}
        TDegreDialogue;
```

- TFoncMessage : C'est le type des fonctions auxquelles mon objet enverra les messages sur la progression des opérations.

```
typedef void (*TFoncMessage)(const std::string &) ;
```

- TStatistiqueHuffman : c'est la structure que j'ai utilisée pour compter le nombre d'apparitions de chaque élément. La méthode CreerNoeudHuffman() permet de générer un noeud (feuille) à utiliser dans l'arbre de huffman. J'ai fais le choix de stocker le code de huffman dans la statistique pour des raisons de facilité d'accès lors de la compression.

```
struct TStatistiqueHuffman
{
   unsigned long int Frequence;
   std::string sCode;

   TStatistiqueHuffman() : Frequence(0), sCode("") {}
   TNoeudHuffman * CreerNoeudHuffman(unsigned char Valeur);
};
```

- TNoeudHuffman : cette structure permet de construire l'arbre et contient un pointeur vers la string du code de huffman dans le cas d'une feuille. J'aurrai pu utiliser le principe de l'héritage et faire deux structures : noeud et feuille, mais je suis trop fainéant pour ca ;-) Ceci dit, vous pouvez toujours le faire pour votre propre dévellopement. Cette structure propose une méthode pour déterminer le code de huffman des feuilles ainsi qu'une autre pour recréeer une feuille dans l'arbre a partir d'un code de huffman. J'ai bien entendu intégré quelques opérateurs nécessaires au fonctionnement de la file de priorité que nous utiliserons pour la construction de l'arbre. J'ai également déclaré un type pointeur sur ce type de structure.

```
struct TNoeudHuffman
{
   unsigned char Valeur;                    // Fragment
   unsigned int Frequence;                  // Frequence des sous-arbres
   std::string * lpsCode;                   // Pointeur sur la chaine qui
                                            // contient le code de Huffman
   TNoeudHuffman * FilsG;
   TNoeudHuffman * FilsD;

   inline TNoeudHuffman(unsigned char Val = 0, unsigned int Freq = 0,
                        std::string * lpsCodeStr = NULL,
                        TNoeudHuffman * Fg = NULL, TNoeudHuffman * Fd = NULL)
             : Valeur(Val), Frequence(Freq), lpsCode(lpsCodeStr), FilsG(Fg),
               FilsD(Fd){}

   inline ~TNoeudHuffman()
             {
                if (FilsG) delete FilsG;
                if (FilsD) delete FilsD;
             }
   // Fonction récursive pour déterminer le code de Huffman des feuilles
   void DeterminerCodeHuffman(const std::string &sCode) const;

   // Fonction récursive pour créer les noeuds correspondants à un code
   void CreerFeuille(const std::string &sCode, unsigned char valeur);

   friend bool operator<(const TNoeudHuffman &A, const TNoeudHuffman &B);
   friend bool operator>(const TNoeudHuffman &A, const TNoeudHuffman &B);
   friend bool operator==(const TNoeudHuffman &A, const TNoeudHuffman &B);
};
typedef TNoeudHuffman * PNoeudHuffman;
```

Finalement, je vais vous présenter l'outil dont je vous parlais pour pouvoir utiliser les objets de types divers avec la file de priorité. Il est extrement simple et propose divers opérateurs qui sont mappés sur ceux de l'objet sur lequel il pointe.

```
template<class T>
```

```
class PComparateur
{
   private :
      T * pointeur;

   public :
      inline explicit PComparateur(T *p = NULL) : pointeur(p) {}
      inline explicit PComparateur(T &p) : pointeur(&p) {}

      inline T * get() const { return pointeur; }
      inline void set(T *p) { pointeur = p; }
      inline void set(T &p) { pointeur = &p; }

      friend bool operator<(const PComparateur<T> &A, const PComparateur<T> &B)
         {
            return *(A.pointeur) < *(B.pointeur);
         }
      friend bool operator>(const PComparateur<T> &A, const PComparateur<T> &B)
         {
            return *(A.pointeur) > *(B.pointeur);
         }
      friend bool operator<=(const PComparateur<T> &A, const PComparateur<T> &B)
         {
            return *(A.pointeur) <= *(B.pointeur);
         }
      friend bool operator>=(const PComparateur<T> &A, const PComparateur<T> &B)
         {
            return *(A.pointeur) >= *(B.pointeur);
         }
      friend bool operator==(const PComparateur<T> &A, const PComparateur<T> &B)
         {
            return *(A.pointeur) == *(B.pointeur);
         }

      friend std::ostream &operator<<(std::ostream &out, const PComparateur<T> &B)
         {
            return out << *(B.pointeur);
         }

      friend std::istream &operator>>(std::istream &in, const PComparateur<T> &B)
         {
            return in >> *(B.pointeur);
         }
```

```
};
```

c) codage des méthodes

Dans cette partie, je vais simplement parcourir le fichier de code et vous expliquer les méthodes unes a unes.

- DeterminerCodeHuffman : L'utilisation d'une chaine de caractères rend la détermination du code de huffman très simple par l'utilisation d'une fonction récursive : tout ce qu'il y a à faire est un parcours en profondeur de l'arbre en concaténant un caractère '0' ou '1' au code du noeud actuel suivant si on prend le fils gauche ou le fils droite.

```
void TNoeudHuffman::DeterminerCodeHuffman(const std::string &sCode) const
{
   // seules les feuilles terminales devraient avoir lpsCode non NULL
   if (lpsCode)
      *lpsCode = sCode;
   else
   {
      if (FilsG)
         FilsG->DeterminerCodeHuffman(sCode + "0");
      if (FilsD)
         FilsD->DeterminerCodeHuffman(sCode + "1");
   }
}
```

- CreerFeuille : C'est exactement l'inverse de la fonction précédente : on va "manger" un caractère a chaque étape et créer les noeuds du chemin au fur et à mesure si nécessaire, jusqu'a ce qu'on arrive à la feuille.

```
void TNoeudHuffman::CreerFeuille(const std::string &sCode,
                                 unsigned char valeur)
{
   TNoeudHuffman * * p;
```

```
      // Détermination du pointeur qui va être modifié
      if (sCode[0] == '0')
         p = &FilsG;
      else
         p = &FilsD;

      if (sCode.length() > 1)
      {
         // Création d'un noeud intermédiaire
         if (!(*p))
            *p = new TNoeudHuffman();
         (*p)->CreerFeuille(sCode.substr(1,sCode.length()),valeur);
      }
      else
      {
         // Création de la feuille (*p est obligatoirement null puisqu'une
         // feuille ne peut pas être un noeud intermédiaire)
         *p = new TNoeudHuffman(valeur);
         p = p;
      }

   }
```

Je fais l'impasse sur les opérateurs, le constructeur et le destructeur qui ne présentent pas d'itérêt particulier.

- Analyser : Cette fonction est extrêmement simple : elle alloue dynamiquement la mémoire nécessaire aux statistiques puis compte le nombre d'apparitions de chaque caractère du fichier.

```
   bool THuffman::Analyser()
   {
      std::ifstream fSrc;
      char Valeur;
      char BufferProgression[10];
      unsigned long int Taille = TailleInitiale();
      unsigned long int Position = 0;

      if (DegreVerbalite & ddEtape)
         FoncAffichageEtape("Analyse du fichier source");
```

```
// Si le nom du fichier n'est pas défini ce n'est pas nécessaire d'essayer
// de l'ouvrir
if (sSource.length())
   fSrc.open(sSource.c_str(), std::ios::in | std::ios::binary);
else
{
   ErrCode = ERRCODE_FICHIERINDEFINI;
   sErrString = "Veuillez préciser le nom du fichier source";
   return false;
}

if (fSrc.is_open())
{
   if (DegreVerbalite & ddProgression)
      FoncAffichageProgression("0%");
   // Allocation du tableau de statistiques
   if (Statistiques)
      delete [] Statistiques;
   Statistiques = new TStatistiqueHuffman[256];

   // Parcours du fichier caractère par caractère
   // Les optimisations d'accès au fichier sont gérées par l'OS
   while (!fSrc.eof())
   {
      fSrc.read((char *)(&Valeur),sizeof(char));
      Statistiques[(unsigned char)(Valeur)].Frequence++;
      sprintf(BufferProgression,"%d%%",int(float(Position)/Taille*100));
      if (DegreVerbalite & ddProgression)
         FoncAffichageProgression(BufferProgression);
      Position++;
   }

   // Fermeture du fichier
   fSrc.close();
   return true;
   ErrCode = ERRCODE_PASDERREUR;
}
else
{
   ErrCode = ERRCODE_OUVERTURELECTURE;
   sErrString = "Impossible d'ouvrir le fichier d'entrée en lecture";
   return false;
```

```
      }
   }
```

- Compresser : Cette fonction s'arrange pour que toutes les étapes nécessaires à la compression soient effectuées (analyse, arbre, entête) puis lit le fichier source et concatène le code correspondant à chacun des caractères dans une chaine de buffer. Dès que la chaine de buffer est suffisament remplie (32 bits), on utilise alors un bitset de 32 bits pour convertir la sous-chaine des 32 premiers caractères en un entier long non-signé qui est alors écrit dans le fichier de destination. Lorsque tout le fichier source a été lu, le reste du buffer est rempli de 0 pour arriver à 4 octets puis écrit sur le disque. La compression est finie :)

```
bool THuffman::Compresser()
{
   typedef std::bitset<32> TDataBuffer;
   std::ifstream fSrc;
   std::ofstream fDst;
   char Valeur;
   char BufferProg[10];
   unsigned long int Position = 0;
   std::string BufferOut;
   TDataBuffer Data;
   unsigned long int uData;

   TailleReelle = TailleInitiale();

   if (!Statistiques)
      if (!Analyser())
         return false;

   if (!ArbreHuffman)
      if (!ConstruireArbreFreq())
         return false;

   if (!EcrireEntete())
      return false;

   // Compression du fichier source

   if (DegreVerbalite & ddEtape)
      FoncAffichageEtape("Compression en cours");
```

```
if (sSource.length())
{
   // Ouverture du fichier source
   fSrc.open(sSource.c_str(), std::ios::in | std::ios::binary);

   // Ouverture du fichier de destination
   if (fSrc.is_open())
   {
      if (sDestination.length())
        fDst.open(sDestination.c_str(), std::ios::app | std::ios::binary);
      else
      {
         ErrCode = ERRCODE_FICHIERINDEFINI;
        sErrString = "Veuillez préciser le nom du fichier de destination";
         return false;
      }
   }
   else
   {
         ErrCode = ERRCODE_OUVERTURELECTURE;
        sErrString = "Impossible d'ouvrir le fichier d'entrée en lecture";
         return false;
   }
}
else
{
   ErrCode = ERRCODE_FICHIERINDEFINI;
   sErrString = "Veuillez préciser le nom du fichier source";
   return false;
}

if (fDst.is_open())
{
   if (DegreVerbalite & ddProgression)
      FoncAffichageProgression("0%");
   // Parcours du fichier caractère par caractère
   // Les optimisations d'accès au fichier sont gérées par l'OS
   fSrc.read((char *)(&Valeur),sizeof(char));
   while (!fSrc.eof())
   {
      // insertion du code correspondant dans le buffer
      BufferOut += Statistiques[(unsigned char)(Valeur)].sCode;
      if (BufferOut.length() > 31)
      {
```

```
            // On dispose d'assez de données pour écrire
            // Conversion du buffer de caractères '1' et '0' en entier long
            Data = static_cast<TDataBuffer>(BufferOut.substr(0,32));
            uData = Data.to_ulong();
            // écriture des données
            fDst.write((char *)(&uData),sizeof(unsigned long int));
            if (!fDst.good())
            {
               ErrCode = ERRCODE_ERREURECRITURE;
               sErrString = "Erreur lors de l'écriture des données";
               return false;
            }
            // libération du buffer
            BufferOut.erase(0,32);
         }
         sprintf(BufferProg,"%d%%",int(float(Position) / TailleReelle * 100));
         if (DegreVerbalite & ddProgression)
            FoncAffichageProgression(BufferProg);
         Position++;
         fSrc.read((char *)(&Valeur),sizeof(char));
      }

      // vidage du reste du buffer
      Data.reset();
      BufferOut.resize(32,'0');
     Data = static_cast<TDataBuffer>(BufferOut);
      uData = Data.to_ulong();
      fDst.write((char *)(&uData),sizeof(unsigned long int));

      // Fermeture des fichier
      fSrc.close();
      fDst.close();
   }
   else
   {
      ErrCode = ERRCODE_OUVERTUREECRITURE;
      sErrString = "Impossible d'ouvrir le fichier de sortie en écriture";
      return false;
   }

   ErrCode = ERRCODE_PASDERREUR;
   return true;
```

```
}
```

- Decompresser : Cette fonction s'arrange pour que toutes les étapes nécessaires à la décompression soient effectuées (lecture de l'entête, arbre) puis lit le fichier source 4 octets par 4 octets. Chaque groupe de 4 octets est alors transformé en bitset qu'on utilise alors pour parcourrir l'arbre de Huffman.

```
bool THuffman::Decompresser()
{
   typedef std::bitset<32> TDataBuffer;
   std::ifstream fSrc;
   std::ofstream fDst;
   char Valeur;
   char BufferProg[10];
   unsigned long int Position = 0;
   TDataBuffer Data;
   unsigned long int uData;
   TNoeudHuffman * Pos;
   int i;


   if (!Statistiques)
      if (!LireEntete())
         return false;

   if (!ArbreHuffman)
      if (!ConstruireArbreCode())
         return false;

   // Décompression du fichier source
   if (DegreVerbalite & ddEtape)
      FoncAffichageEtape("Décompression en cours");
   if (sSource.length())
   {
      // Ouverture du fichier source
      fSrc.open(sSource.c_str(), std::ios::in | std::ios::binary);

      // Ouverture du fichier de destination
      if (fSrc.is_open())
```

```
   {
      if (sDestination.length())
        fDst.open(sDestination.c_str(), std::ios::out | std::ios::binary);
      else
      {
         ErrCode = ERRCODE_FICHIERINDEFINI;
        sErrString = "Veuillez préciser le nom du fichier de destination";
         return false;
      }
   }
   else
   {
         ErrCode = ERRCODE_OUVERTURELECTURE;
        sErrString = "Impossible d'ouvrir le fichier d'entrée en lecture";
         return false;
   }
}
else
{
   ErrCode = ERRCODE_FICHIERINDEFINI;
   sErrString = "Veuillez préciser le nom du fichier source";
   return false;
}

if (fDst.is_open())
{
   fSrc.seekg(FinEntete, std::ios::beg);
   if (DegreVerbalite & ddProgression)
      FoncAffichageProgression("0%");
   Pos = ArbreHuffman;
   // Parcours du fichier 4 octets par 4 octets
   // Les optimisations d'accès au fichier sont gérées par l'OS
   fSrc.read((char *)(&uData),sizeof(unsigned long int));
   while (!fSrc.eof())
   {

      // Conversion du buffer en bitset
      Data = static_cast<TDataBuffer>(uData);

      // Parcours du buffer pour décoder les caractères
      for (i = 31 ; (i >= 0) && (Position < TailleReelle) ; i--)
      {
         // Parcours de l'arbre
         if (Data.test(i))
```

```
            Pos = Pos->FilsD;
         else
            Pos = Pos->FilsG;

         if (!Pos)
         {
            ErrCode = ERRCODE_INATTENDUE;
            sErrString = "Impossible de lire dans l'arbre de Huffman";
            return false;
         }

         if (!(Pos->FilsD || Pos->FilsG))
         {
            // Feuille terminale : écriture de l'octet
            fDst.write((char *)(&(Pos->Valeur)),1);
            if (!fDst.good())
            {
               ErrCode = ERRCODE_ERREURECRITURE;
               sErrString = "Erreur lors de l'écriture des données";
               return false;
            }
            // retour à la racine de l'arbre
            Pos = ArbreHuffman;
            Position++;
         }
      }

      sprintf(BufferProg,"%d%%",int(float(Position) / TailleReelle * 100));
      if (DegreVerbalite & ddProgression)
         FoncAffichageProgression(BufferProg);

      fSrc.read((char *)(&uData),sizeof(unsigned long int));
   }

   // Fermeture des fichier
   fSrc.close();
   fDst.close();
}
else
{
   ErrCode = ERRCODE_OUVERTUREECRITURE;
   sErrString = "Impossible d'ouvrir le fichier de sortie en écriture";
   return false;
}
```

```
      ErrCode = ERRCODE_PASDERREUR;
      return true;
   }
```

Je fais encore l'impasse sur les fonctions de calcul de taille des fichiers qui ne sont que de simples opérations mathématiques à la portée de tous. Je saute également les fonctions d'accès aux aux membres de la classe.

- EcrireEntete : Ici, on fournit les informations sur le fichier original et la table de décompresion. pour la table de décompression, je fournis : 1 octet pour désigner l'élément, 1 octet pour le nobre de bits et 1 octet ou 1 short pour le code en lui même. Bien entendu, je ne donne que les caractères utiles.

```
   bool THuffman::EcrireEntete()
   {
      std::ofstream fDst;
      unsigned long int ULongValue;
      unsigned short int UShortValue;
      unsigned char UCharValue;
      std::string sNomFichier;
      unsigned long int DernierSeparateur;
      char BufferProgression[10];


      if (DegreVerbalite & ddEtape)
         FoncAffichageEtape("Ecriture de l'entête");

      if (sSource.length())
      {
         if (sDestination.length())
            fDst.open(sDestination.c_str(), std::ios::out | std::ios::binary);
         else
         {
            ErrCode = ERRCODE_FICHIERINDEFINI;
            sErrString = "Veuillez préciser le nom du fichier de destination";
            return false;
         }
      }
      else
```

```
{
   ErrCode = ERRCODE_FICHIERINDEFINI;
   sErrString = "Veuillez préciser le nom du fichier source";
   return false;
}

if (fDst.is_open())
{
   if (DegreVerbalite & ddProgression)
      FoncAffichageProgression("0%");
   // Ecriture de la taille du fichier en octets
   ULongValue = TailleInitiale();
   fDst.write((char *)(&ULongValue),sizeof(unsigned long int));
   if (!fDst.good())
   {
      ErrCode = ERRCODE_ERREURECRITURE;
      sErrString = "Erreur lors de l'écriture de l'entête";
      return false;
   }

   // Ecriture du nom du fichier original sans le chemin
   DernierSeparateur = sSource.rfind(SEPARATEUR_REPERTOIRE);
   if (DernierSeparateur > sSource.length())
      // Il n'y a pas le chemin
      sNomFichier = sSource;
   else
      sNomFichier = sSource.substr(DernierSeparateur + 1);

   if (sSource.empty())
   {
      ErrCode = ERRCODE_FICHIERINDEFINI;
      sErrString = "Veuillez préciser le fichier d'entrée";
      return false;
   }


   // Ecriture de la taille du nom du fichier original
   UCharValue = sNomFichier.length();
   fDst.write((char *)(&UCharValue),sizeof(char));
   if (!fDst.good())
   {
      ErrCode = ERRCODE_ERREURECRITURE;
      sErrString = "Erreur lors de l'écriture de l'entête";
      return false;
```

```
}

fDst.write(sNomFichier.c_str(),UCharValue);
if (!fDst.good())
{
   ErrCode = ERRCODE_ERREURECRITURE;
   sErrString = "Erreur lors de l'écriture de l'entête";
   return false;
}

// Ecriture du nombre d'éléments différents
fDst.write((char *)(&NombreElements),sizeof(unsigned short int));
if (!fDst.good())
{
   ErrCode = ERRCODE_ERREURECRITURE;
   sErrString = "Erreur lors de l'écriture de l'entête";
   return false;
}

if (DegreVerbalite & ddProgression)
   FoncAffichageProgression("1%");
for (int i = 0 ; i < 256 ; i++)
{
   // Si le caractère a une fréquence nulle on passe au suivant
   if (!Statistiques[i].Frequence)
      continue;

   // Ecriture du code du caractère
   UCharValue = i;
   fDst.write((char *)(&UCharValue),sizeof(char));
   if (!fDst.good())
   {
      ErrCode = ERRCODE_ERREURECRITURE;
      sErrString = "Erreur lors de l'écriture de l'entête";
      return false;
   }

   // Ecriture du nombre de bits du caractère
   UCharValue = Statistiques[i].sCode.length();
   fDst.write((char *)(&UCharValue),sizeof(char));
   if (!fDst.good())
   {
      ErrCode = ERRCODE_ERREURECRITURE;
      sErrString = "Erreur lors de l'écriture de l'entête";
```

```
            return false;
         }

         if (UCharValue <= 8)
         {
            // Ecriture du code dans un espace de 8 bits
            UCharValue = 0;
            for (int j = 0 ; j < Statistiques[i].sCode.length() ; j++)
               if (Statistiques[i].sCode[j] == '1')
                  UCharValue |= (0x01 << (7 - j));
            fDst.write((char *)(&UCharValue),sizeof(char));
            if (!fDst.good())
            {
               ErrCode = ERRCODE_ERREURECRITURE;
               sErrString = "Erreur lors de l'écriture de l'entête";
               return false;
            }
         }
         else
         {
            // Ecriture du code dans un espace de 16 bits
            UShortValue = 0;
            for (int j = 0 ; j < Statistiques[i].sCode.length() ; j++)
               if (Statistiques[i].sCode[j] == '1')
                  UShortValue |= (0x01 << (15 - j));
            fDst.write((char *)(&UShortValue),sizeof(unsigned short int));
            if (!fDst.good())
            {
               ErrCode = ERRCODE_ERREURECRITURE;
               sErrString = "Erreur lors de l'écriture de l'entête";
               return false;
            }
         }
         sprintf(BufferProgression,"%d%%",(((i*99)/256) + 1));
         if (DegreVerbalite & ddProgression)
            FoncAffichageProgression(BufferProgression);
      }

      if (DegreVerbalite & ddProgression)
         FoncAffichageProgression("100%");
      fDst.close();
      ErrCode = ERRCODE_PASDERREUR;
      return true;
   }
```

```
      else
      {
         ErrCode = ERRCODE_OUVERTUREECRITURE;
         sErrString = "Impossible d'ouvrir le fichier de sortie en écriture";
         return false;
      }
   }
```

- LireEntete : Ici, on fait l'inverse : on relit les informations générales puis on relit la table de décompression à l'aide de masques de bits.

```
   bool THuffman::LireEntete()
   {
      std::ifstream fSrc;
      unsigned short int UShortValue;
      unsigned char UCharValue;
      char * buffer;
      int Elt;
      int NbBits;
      char BufferProgression[10];


      if (DegreVerbalite & ddEtape)
         FoncAffichageEtape("Lecture de l'entête");

      if (sSource.length())
         fSrc.open(sSource.c_str(), std::ios::in | std::ios::binary);
      else
      {
         ErrCode = ERRCODE_FICHIERINDEFINI;
         sErrString = "Veuillez préciser le nom du fichier source";
         return false;
      }

      if (fSrc.is_open())
      {
         // réinitialisation des statistiques
         if (Statistiques)
            delete [] Statistiques;
```

```
Statistiques = new TStatistiqueHuffman[256];

// Lecture de la taille du fichier en octets
fSrc.read((char *)(&TailleReelle),sizeof(unsigned long int));
if (!fSrc.good())
{
   ErrCode = ERRCODE_ERREURLECTURE;
   sErrString = "Erreur lors de la lecture de l'entête";
   return false;
}

// Lecture de la taille du nom du fichier original
fSrc.read((char *)(&UCharValue),sizeof(char));
if (!fSrc.good())
{
   ErrCode = ERRCODE_ERREURLECTURE;
   sErrString = "Erreur lors de la lecture de l'entête";
   return false;
}

// Création du buffer pour la lecture du nom du fichier
buffer = new char[UCharValue+1];

// Lecture du nom du fichier original
fSrc.read(buffer,UCharValue);
if (!fSrc.good())
{
   ErrCode = ERRCODE_ERREURLECTURE;
   sErrString = "Erreur lors de la lecture de l'entête";
   return false;
}

// Zero terminal
buffer[UCharValue] = 0;
if (sDestination.empty())
   sDestination = buffer;

// Lecture du nombre d'éléments différents
fSrc.read((char *)(&NombreElements),sizeof(unsigned short int));
if (!fSrc.good())
{
   ErrCode = ERRCODE_ERREURLECTURE;
   sErrString = "Erreur lors de la lecture de l'entête";
   return false;
```

```
}

for (int i = 0 ; i < NombreElements ; i++)
{
   // Lecture du code du caractère
   fSrc.read((char *)(&UCharValue),sizeof(char));
   if (!fSrc.good())
   {
      ErrCode = ERRCODE_ERREURLECTURE;
      sErrString = "Erreur lors de la lecture de l'entête";
      return false;
   }
   Elt = UCharValue;

   // Lecture du nombre de bits du caractère
   fSrc.read((char *)(&UCharValue),sizeof(char));
   if (!fSrc.good())
   {
      ErrCode = ERRCODE_ERREURLECTURE;
      sErrString = "Erreur lors de la lecture de l'entête";
      return false;
   }
   NbBits = UCharValue;

   if (NbBits <= 8)
   {
      // Lecture du code dans un espace de 8 bits
      fSrc.read((char *)(&UCharValue),sizeof(char));
      if (!fSrc.good())
      {
         ErrCode = ERRCODE_ERREURLECTURE;
         sErrString = "Erreur lors de la lecture de l'entête";
         return false;
      }
      for (int j = 0 ; j < NbBits ; j++)
         Statistiques[Elt].sCode += (UCharValue & (0x01 << (7 - j))) ?
                                                        "1" : "0";
   }
   else
   {
      // Lecture du code dans un espace de 16 bits
      fSrc.read((char *)(&UShortValue),sizeof(unsigned short int));
      if (!fSrc.good())
      {
```

```
                ErrCode = ERRCODE_ERREURLECTURE;
                sErrString = "Erreur lors de la lecture de l'entête";
                return false;
             }
             for (int j = 0 ; j < NbBits ; j++)
                Statistiques[Elt].sCode += (UShortValue & (0x01 << (15 - j)))
                                                        ? "1" : "0";
          }

          sprintf(BufferProgression,"%d%%",(((Elt*99)/256) + 1));
          if (DegreVerbalite & ddProgression)
             FoncAffichageProgression(BufferProgression);
       }

       if (DegreVerbalite & ddProgression)
          FoncAffichageProgression("100%");
       FinEntete = fSrc.tellg();
       fSrc.seekg(0, std::ios::beg);
       FinEntete -= fSrc.tellg();
       fSrc.close();
       ErrCode = ERRCODE_PASDERREUR;
       return true;
    }
    else
    {
       ErrCode = ERRCODE_OUVERTURELECTURE;
       sErrString = "Impossible d'ouvrir le fichier d'entrée en lecture";
       return false;
    }
 }
```

- ConstruireArbreFreq : On veut construire l'arbre de huffman a partir de l'analyse effectuée sur le fichier source. Tout d'abord, on rempli la file à priorité de la librairie standard avec des noeuds créés à partir des statistiques. Ensuite, il suffit d'assembler les éléments les moins prioritaires deux par deux et de réintroduire le nouveau noeud dans la file de priorité et ainsi de suite jusqu'à ce qu'il n'y ai plus qu'un seul élément dans la file : la racine de l'arbre.

```
bool THuffman::ConstruireArbreFreq()
{
```

```
typedef PComparateur<TNoeudHuffman> PCompHuffman;
PCompHuffman NoeudA;
PCompHuffman NoeudB;
// Création d'une file à priorité classée en ordre inverse
std::priority_queue<PCompHuffman,
                    std::deque<PCompHuffman>,
                    std::greater<PCompHuffman> > FilePrioNoeuds;

if (!Statistiques)
{
   ErrCode = ERRCODE_PASANALYSE;
   sErrString = "L'analyse n'a pas été effectuée";
   return false;
}

// Création de la liste initiale
for (int i = 0 ; i < 256 ; i++)
   if (Statistiques[i].Frequence)
   {
      // On ne considère que les fragments qui apparaissent
  FilePrioNoeuds.push(PCompHuffman(Statistiques[i].CreerNoeudHuffman(i)));
   }

if (FilePrioNoeuds.empty())
{
   ErrCode = ERRCODE_FICHIERVIDE;
   sErrString = "Fichier vide";
   return false;
}

NombreElements = FilePrioNoeuds.size();

// Génération de l'arbre de Huffman
while (FilePrioNoeuds.size() > 1)
{
   NoeudA = FilePrioNoeuds.top();
   FilePrioNoeuds.pop();
   NoeudB = FilePrioNoeuds.top();
   FilePrioNoeuds.pop();

   FilePrioNoeuds.push(PCompHuffman(new TNoeudHuffman(0,
                                                      NoeudA.get()->Frequence
                                                      + NoeudB.get()->Frequence,
                                                      NULL,
```

```
                                                    NoeudA.get(),
                                                    NoeudB.get()));
      }

      // Détermination de la racine et vidage de la file à priorité
      ArbreHuffman = FilePrioNoeuds.top().get();
      FilePrioNoeuds.pop();

      // Détermination des codes de Huffman
      ArbreHuffman->DeterminerCodeHuffman("");

      ErrCode = ERRCODE_PASDERREUR;
      return true;

   }
```

- ConstruireArbreCode : On veut construire l'arbre de Huffman à partir des codes de huffman lus dans l'entête du fichier compressé. Il suffit d'appeller la fonction CreerFeuille pour chaque statistique a partir de la racine et le travail se fait tout seul :-)

```
bool THuffman::ConstruireArbreCode()
{
   char BufferProg[10];
   if (DegreVerbalite & ddEtape)
      FoncAffichageEtape("Création de l'arbre de décodage");
   if (DegreVerbalite & ddProgression)
      FoncAffichageProgression("0%");
   // Création de la racine
   ArbreHuffman = new TNoeudHuffman();
   for (int i = 0 ; i <  256 ; i++)
   {
      if (Statistiques[i].sCode.length())
      {
         ArbreHuffman->CreerFeuille(Statistiques[i].sCode, i);
      }
      sprintf(BufferProg, "%d%%", (i * 100)/256);
      if (DegreVerbalite & ddProgression)
         FoncAffichageProgression(BufferProg);
   }
   return true;
```

```
    }
```

III. Remarques diverses en conclusion

a) Compilateur

Ca y est, tout à été vu, il ne reste plus qu'à compiler. Pensez à utiliser les fonctions d'optimisation du compilateur car aucun de mes algorithmes n'a été optimisé, et l'utilisation des chaines de caractères est particulièrement lente. Par ailleurs, il se peut que le code ne soit pas fonctionnel avec tous les compilateurs : j'ai eu quelques problèmes avec le compilateur intégré de Kylix lors du développement et j'ai du passer au compilateur GCC v3.2 (fournit en standard dans Linux Mandrake 9.0). Je n'ai pas refait de tests depuis que j'ai fini mon développement, mais il se peut que cela vienne d'une erreur personnelle, tout comme il se peut qu'un partie de l'implémentation de la STL contienne un bug sur ce compilateur. Je vous invite donc à me communiquer le nom de tout compilateur ne produisant pas un code fonctionnel.

b) Optimisation

L'efficacité de ce code est grandement dépendant de l'implémentation de la librairie standard. Aussi, il pourrait être intéressant de faire des tests de temps d'exécution sur une file a priorité basée sur une list plutot que sur une deque, même si cela ne représente qu'une minuscule partie du temps d'exécution total (quelques milisecondes).
Comme je l'ai déjà dit, l'utilisation des chaines de caractères pour représenter les bits est excellente au niveau pédagogique car très visuelle, mais elle est désastreuse au niveau des performances. Je vous encourage à essayer de développer une version utilisant directement le mode natif des bits dans des entiers, même si cela peut se révéler être une accrobatie extrêmement périlleuse.
Pour améliorer encore les temps d'exécution, limitez la verbosité du programme au maximum. Vous pouvez également essayer de faire une version qui n'analyse qu'une partie du fichier à compresser (dans le cas des gros fichiers) mais il faut faire attention aux caractères non rencontrés pendant l'analyse : il faut décider de ce qu'on fait d'eux car ils aurront tout de meme besoin d'un code !