

# Projet de « Programmation (langage C) »

IUP 1<sup>ère</sup> année

Année 2001-2002

## 1 Modalités de réalisation

Deux sujets sont proposés au choix, à réaliser en C, à réaliser par groupes de 3 étudiants au maximum :

- sujet 1 : réalisation d'une bibliothèque de fonctions permettant la manipulation d'images représentées sous forme d'arbres quadrants (section 2) ;
- sujet 2 : réalisation d'un compresseur de fichiers par l'algorithme de Lempel-Ziv-Welch (section 3).

Une soutenance orale dans laquelle chaque membre d'un groupe exposera le travail qu'il a effectué aura lieu **début mars**. Un rapport sera remis au plus tard une semaine avant la soutenance. Il doit exposer brièvement (une dizaine de pages maximum) les structures de données utilisées, les algorithmes choisis et les difficultés rencontrées. Il ne s'agit pas de commenter le source une seconde fois, ni d'expliquer chaque fonction en détail (c'est le rôle, à nouveau, des commentaires du source, en particulier de ceux se trouvant devant les prototypes des fonctions), mais d'en présenter une synthèse. Enfin, le source devra également être fourni par mail avant la soutenance.

Les points suivants devront faire l'objet d'une attention particulière lors de l'écriture du programme : sa correction, sa lisibilité et sa clarté, sa modularité (développement de petites fonctions, répartition du source en plusieurs fichiers regroupant des fonctions de façon logique), sa portabilité, ainsi que le choix des structures de données et si possible l'efficacité des algorithmes. Ce dernier point n'est cependant pas essentiel ici car il fait l'objet d'un cours ultérieur (algorithmique). L'utilisation de `make` et `gdb` sera utile pour accélérer le développement.

## 2 Sujet 1 : manipulation d'images. Arbres quadrants

### 2.1 Objet du projet

Le but de ce projet est d'écrire une bibliothèque de fonctions de manipulation d'arbres quadrants, ainsi qu'une interface d'utilisation de ces fonctions. Ces arbres permettent la représentation et certaines transformations d'images.

### 2.2 Arbres quadrants

On veut manipuler des images, constituées de pixels coloriés disposés sur une surface carrée dont le côté a  $2^n$  pixels, où  $n$  est un entier positif. Dans ce projet, on ne considérera que deux couleurs, blanc ou noir. On représente ces images par un arbre *orienté* ayant les caractéristiques suivantes :

- chaque nœud interne possède quatre fils ;
- l'arbre est complet : toutes les feuilles sont à la même hauteur ;
- chaque feuille est coloriée.

L'arbre est défini récursivement de la façon suivante :

- si le côté du carré est  $1 = 2^0$ , l'arbre consiste en une unique feuille, dont la couleur est celle de l'unique pixel du carré ;
- si le côté du carré est  $2^n$ , la racine de l'arbre a 4 fils. Le premier fils représente la partie Nord-ouest de l'image, le deuxième la partie Nord-est, le troisième la partie Sud-ouest, et enfin le quatrième la partie Sud-est. Chacun des fils représente ainsi un carré de  $2^{n-1} \times 2^{n-1}$  pixels, et l'arbre est de hauteur  $n$ .

Ainsi, chaque feuille représente un pixel dont elle porte la couleur. Dans l'exemple de la figure 1,  $n = 3$ . Les feuilles correspondant à des pixels noirs sont marquées par un carré noir, les autres n'ont pas été dessinées. Les nœuds internes sont marqués par des disques.

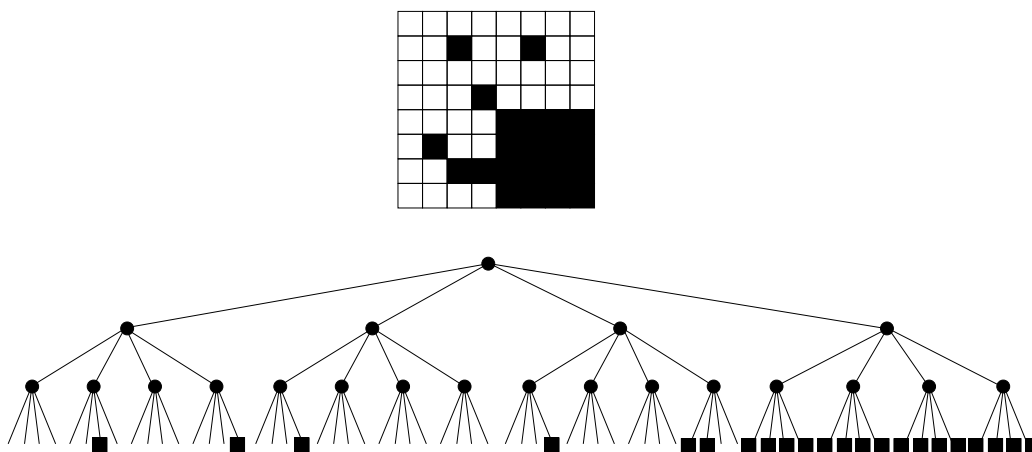


FIG. 1 – Un arbre quadrant et son image associée

À des fins de vérification, les figures pourront aussi être lues et écrites dans des fichiers de caractères, un octet particulier représentant les pixels blancs (espace par exemple), un autre les pixels noirs. Si on reprend l'exemple de la figure 1, en représentant un pixel blanc par le caractère espace (␣ sur la figure ci-dessous) et un pixel noir par \*, on obtiendrait (↵ représentant le caractère *newline*):

```

␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ↵
␣ ␣ * ␣ ␣ * ␣ ␣ ↵
␣ ␣ ␣ ␣ ␣ ␣ ␣ ␣ ↵
␣ ␣ ␣ * ␣ ␣ ␣ ␣ ↵
␣ ␣ ␣ ␣ * * * * ↵
␣ * ␣ ␣ * * * * ↵
␣ ␣ * * * * * * ↵
␣ ␣ ␣ ␣ * * * * ↵

```

FIG. 2 – Représentation d'une image dans un fichier

### 2.3 Travail demandé (première partie)

- A. On demande de définir :
  - A.1. un type `quad` permettant de représenter les arbres quadrants ;
  - A.2. une fonction `int ecrire_quad(FILE *f, quad *t)` ; sauvegardant dans le fichier ouvert donné par `f` l'image représentée par `t` (sous le format donné précédemment) ;
  - A.3. une fonction `quad *lire_quad(FILE *f)` ; lisant une image d'un fichier représentant une image, et retournant un pointeur sur l'arbre correspondant ;
  - A.4. une fonction `quad *intersection(quad *t1, quad *t2)` ; calculant l'intersection des images représentées par les arbres pointés par `t1` et `t2`. Les pixels noirs de la nouvelle image sont ceux qui étaient noirs dans les deux images de départ ;
  - A.5. une fonction `quad *union(quad *t1, quad *t2)` ; calculant l'union des images représentées par les arbres pointés par `t1` et `t2`. Les pixels noirs de la nouvelle image sont ceux qui étaient noirs dans l'une ou l'autre des images de départ ;
  - A.6. des fonctions `quad *symetrie_verticale(quad *t)` ; et `quad *symetrie_horizontale(quad *t)` ; permettant d'effectuer des symétries par rapport aux axes médians ;
  - A.7. une fonction `quad *rotation(quad *t)` ; effectuant la rotation d'angle  $\frac{\pi}{2}$  et de centre le milieu de l'image représentée par `t` ;
  - A.8. une fonction `quad *inversion_video(quad *t)` ; effectuant l'échange des couleurs blanc-noir dans l'image représentée par `t` ;

- A.9. une fonction `quad *zoom(quad *t, int quart)`; effectuant l'agrandissement d'un quart de la figure. On peut demander l'agrandissement de l'un des quatre quarts de plan correspondant aux quatre premiers fils. L'argument `quart` sert à préciser quel quart de l'image on agrandit. On introduira les constantes `NO`, `NE`, `SO`, `SE` pour spécifier l'une de ces parties d'image.
- A.10. une fonction `quad *composante_connexe(quad *t, quad *pixel)`; effectuant la recherche de la composante connexe d'un pixel donné. La *composante connexe* d'un pixel  $p$  est l'ensemble de tous les pixels de même couleur que  $p$  qui sont atteignables à partir de  $p$  en faisant uniquement des pas Nord, Sud, Ouest, ou Est, et en ne traversant que des pixels de même couleur que  $p$ . Par exemple, sur l'exemple de la figure 1, il y a cinq composantes noires. Les pixels de l'image résultante se trouvant dans la composante connexe seront de même couleur que le pixel de départ, les autres seront de couleur opposée.
- A.11. une fonction `quad *composante_connexe2(quad *t, quad *pixel)`; effectuant un deuxième type de recherche de composante connexe, les pas autorisés étant cette fois Nord, Sud, Ouest, Est, Nord-ouest, Nord-est, Sud-ouest, et Sud-est. Sur l'exemple de la figure 1, il n'y a que trois telles composantes noires.
- B. Écrire une interface utilisateur pour permettre un test (convivial) des fonctions ci-dessus. Cette interface doit permettre de fixer la taille des images par l'utilisateur au début de la session de travail. Il ne doit pas être nécessaire de recompiler le projet pour changer cette taille.
- C. Écrire un en-tête `quad.h` donnant accès aux prototypes, constantes et variables fournies par la bibliothèque ;
- D. Compiler la bibliothèque dynamique sous le nom `libquadtrees.so`. Avec `gcc` sur `nivose`, on pourra utiliser une ligne de commande du type (info `gcc` pour plus de détails!):
- ```
$ gcc -Wall -pedantic <fichiers source> -o libquadtrees.so -shared -fpic
```
- Compiler enfin l'interface qui fait appel à cette bibliothèque de la façon suivante :
- ```
$ gcc -Wall -pedantic -L<repertoire bibliothèque> interface.c -o interface -lquadtrees
```
- À l'exécution l'interface, la variable `LD_LIBRARY_PATH` spécifiera le répertoire contenant la bibliothèque.

## 2.4 Une représentation plus compacte. Travail demandé (deuxième partie)

La représentation interne des arbres est parfois inefficace en termes d'espace. Dans l'exemple de la figure 1, de nombreux sous-arbres portent tous la même information. Par exemple, les quatre premières feuilles sont toutes blanches. Dans ce cas, on peut décider de faire porter l'information directement au niveau du nœud qui supporte ces feuilles. De même, le sous-arbre représentant tout le plan Sud-est est de couleur uniforme. On le remplace donc par une feuille. On obtient un arbre qui représente encore la même image, mais qui possède moins de nœuds (toujours avec la même convention de représentation des zones blanches) :

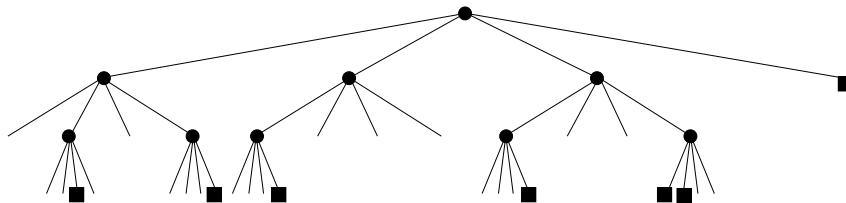


FIG. 3 – L'arbre de la figure 1 « allégé »

Formellement, on remplace tout sous-arbre maximal de couleur uniforme par une feuille de cette couleur. On demande de réécrire la bibliothèque pour qu'elle puisse manipuler de tels arbres.

## 2.5 Extensions possibles

Selon le temps et la motivation restante, on peut ajouter des fonctionnalités (prises en compte dans la notation uniquement si le travail précédent est correctement réalisé). Par exemple : traitement de plusieurs couleurs ; zoom sur une partie d'image autre que l'un des quarts de plan `NO`, `NE`, `SO` ou `SE` ; ajout d'une interface graphique...

## 3 Sujet 2 : compression en codage de Lempel-Ziv-Welch (LZW)

### 3.1 Objet du projet

Le but de ce projet est d'écrire un programme permettant de compresser des fichiers afin qu'ils occupent moins d'espace sur périphérique de stockage. On demande également d'écrire un décompresseur pouvant restaurer le fichier original. Les algorithmes de compression sont nombreux. Celui proposé ici est l'algorithme de Lempel-Ziv-Welch.

### 3.2 Principe de l'algorithme LZW : la compression

L'algorithme LZW lit le fichier à compresser linéairement, et construit un dictionnaire pendant la lecture. Ce dictionnaire contient des fragments de texte, appelés *phrases*, constitués de caractères consécutifs du fichier. Chaque phrase est associée dans le dictionnaire à un *code* entier. Initialement, le dictionnaire ne contient que les 256 phrases composées d'un unique caractère (on supposera que le type `char` est représenté sur un octet), et le code de chacune d'elles est la valeur de l'`unsigned char` correspondant. Au fur et à mesure de la compression, le dictionnaire s'enrichit de nouvelles phrases avec pour chacune un code entier.

L'algorithme maintient une *fenêtre* de texte comprenant des caractères consécutifs du fichier original. Les caractères apparaissant avant cette fenêtre sont considérés comme déjà traités. La fenêtre se déplace vers la droite dans le fichier original. Initialement, elle ne contient que le premier caractère du fichier. Puis :

1. tant que la fenêtre est dans le dictionnaire, on ajoute le caractère suivant en fin de fenêtre ;
2. lorsqu'on arrête d'ajouter des caractères en fin de fenêtre, on est donc dans un des deux cas suivants :
  - o soit le dernier caractère de la fenêtre est le dernier caractère du fichier et la fenêtre est une phrase du dictionnaire. Dans ce cas, on produit le code de cette phrase dans le fichier compressé, et la compression est terminée.
  - o soit la fenêtre n'est pas une phrase du dictionnaire. Elle est donc de la forme  $\langle s \rangle \langle c \rangle$  où  $\langle s \rangle$  est une phrase du dictionnaire et où  $\langle c \rangle$  est son dernier caractère. Dans ce cas,
    - (a) on ajoute  $\langle s \rangle \langle c \rangle$  dans le dictionnaire avec le 1<sup>er</sup> code disponible ;
    - (b) on produit le code de  $\langle s \rangle$  en sortie dans le fichier compressé ;
    - (c) on décale la fenêtre de telle sorte que  $\langle c \rangle$  devienne son seul caractère, et on reprend en 1.

La figure 4 montre un exemple de fonctionnement de l'algorithme sur le texte `ananas`. Au départ (1), le premier caractère se trouve dans la fenêtre. Comme la chaîne `a` est dans le dictionnaire, la fenêtre grandit d'un caractère (2). La fenêtre `an` n'est pas dans le dictionnaire, donc on l'ajoute `an` au dictionnaire en lui attribuant le premier code non utilisé (ici, 256), et on émet le code de `a` dans le fichier compressé. Le début de la fenêtre se déplace ensuite sur le dernier caractère de la fenêtre précédente (3). Comme `a` est dans le dictionnaire, la fenêtre grandit d'un caractère (4). La fenêtre `na` est ajoutée au dictionnaire (code 257) et on émet le code de `n`. Le début de la fenêtre se déplace ensuite sur le dernier caractère de la fenêtre précédente (5). Comme `a` est dans le dictionnaire, la fenêtre grandit (6). Comme `an` est encore dans le dictionnaire, la fenêtre grandit à nouveau (7). La chaîne `ana` est ajoutée au dictionnaire (code 258), on émet le code de `an`, soit 256 ; on repositionne enfin le début de la fenêtre sur sa dernière lettre (8). Comme `a` est dans le dictionnaire, la fenêtre grandit (9). On ajoute `as` au dictionnaire, on émet le code de `a`, et on déplace la fenêtre sur le caractère `s`. On a atteint la fin de fichier, on émet le code de `s`.

	Fenêtre	Sortie	Ajout dict.	
			phrase	code
(1)	a n a n a s			
(2)	a n a n a s	code(a)	an	256
(3)	a n a n a s			
(4)	a n a n a s	code(n)	na	257
(5)	a n a n a s			
(6)	a n a n a s			
(7)	a n a n a s	256	ana	258
(8)	a n a n a s			
(9)	a n a n a s	code(a)	as	259
(10)	a n a n a s	code(s)		

FIG. 4 – Fonctionnement de l'algorithme

### 3.3 Décompression

L'intérêt de l'algorithme est que l'on n'a pas besoin du dictionnaire pour la décompression : la correspondance entre les codes et les chaînes peut être reconstruite par le décompresseur. On lit pour cela la suite de codes du fichier compressé et on mémorise en permanence les deux derniers codes lus, dans des variables appelées ci-dessous `cc` (pour `code courant`) et `cp` (pour `code précédent`).

- 1) Initialement, le dictionnaire contient les codes des chaînes d'un seul caractère.
- 2) Soit `cc` le premier code du fichier compressé. C'est celui d'une telle chaîne : on produit en sortie le caractère correspondant.
- 3) On sauvegarde `cc` dans `cp`, et on lit dans `cc` le code suivant dans le fichier. Deux cas sont possibles :
  - si le code de `cc` est déjà dans le dictionnaire, on produit en sortie la phrase correspondante. Soit  $\langle c \rangle$  son premier caractère et  $\langle s \rangle$  la chaîne de code `cp` : on ajoute la chaîne  $\langle s \rangle \langle c \rangle$  au dictionnaire ;
  - si le code de `cc` n'est pas encore dans le dictionnaire, soit  $\langle s \rangle$  la chaîne de code `cp` et  $\langle c \rangle$  son premier caractère. On produit  $\langle s \rangle \langle c \rangle$  en sortie et on l'ajoute au dictionnaire en correspondance avec `cc`.

L'étape 3) est répétée jusqu'à la fin du fichier compressé.

### 3.4 Structures de données, détails d'implémentation

Pour une compression efficace, il faut pouvoir déterminer rapidement si une phrase se trouve dans le dictionnaire et si oui, retrouver son code. On peut pour cela utiliser une structure d'arbre dont les nœuds peuvent avoir un nombre variable de fils. Chaque nœud de l'arbre sauf la racine correspond à une phrase codée. Une arête entre deux nœuds est étiquetée par un caractère.

Un nœud de l'arbre correspond à la phrase obtenue en lisant les étiquettes sur les arêtes du chemin de la racine à ce nœud. Le code de la phrase est mémorisé dans le nœud correspondant. Un nœud peut donc avoir autant de fils qu'il y a de lettres dans l'alphabet.

À titre d'exemple, l'arbre construit après lecture de `ananas` est celui de la figure 5 (en codant les phrases d'un seul caractère par le code ASCII du caractère). Ainsi, pour trouver le code d'une phrase, on parcourt l'arbre à partir de sa racine en suivant les arêtes étiquetées par les caractères de la phrase. Si une chaîne  $\langle s \rangle$  amène à un nœud  $n$  de l'arbre, et si pour un caractère  $\langle c \rangle$  aucune arête étiquetée par  $\langle c \rangle$  ne sort du nœud  $n$ , la phrase  $\langle s \rangle \langle c \rangle$  n'est pas encore dans le dictionnaire. Pour l'y insérer, on crée un nouveau nœud  $n'$  et une arête étiquetée  $\langle c \rangle$  reliant  $n'$  à  $n$ . Le nouveau nœud porte le code choisi pour  $\langle s \rangle \langle c \rangle$ .

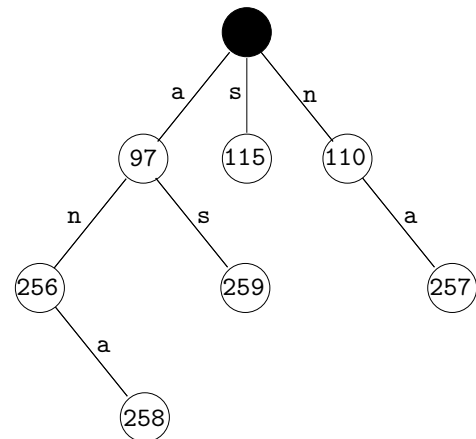


FIG. 5 – Arbre de recherche des codes des phrases

Les entiers utilisés pour le codage des phrases seront plus petits que `USHRT_MAX`, supposé valoir  $2^{16} - 1 = 65535$ . On pourra donc coder 65535 phrases au maximum. Une fonction sera chargée de retourner la longueur nécessaire pour représenter les codes courants, en nombre de bits. Dans une première approche, elle retournera un nombre constant, choisi entre 9 et 15 par le compresseur (une fois pour toute par compression). L'utilisateur du compresseur pourra choisir ce nombre en lançant la compression.

On pourra ensuite adopter l'approche plus économique suivante : lors de la compression ou de la décompression, le nombre  $k$  de phrases dans le dictionnaire varie. Pour économiser sur la taille des codes des phrases, la phrase courante est codée par un code  $\lceil \log k \rceil$  bits.

Enfin, lorsque le dictionnaire atteint sa taille maximale, on pourra choisir

- soit de ne plus insérer de nouveaux codes : le dictionnaire n'évoluera plus,
- soit de recommencer un nouveau dictionnaire. L'ancien dictionnaire sera dans ce cas oublié.

Le choix entre l'une ou l'autre de ces alternatives sera à nouveau possible au lancement de la compression, au travers d'une option.

### 3.5 Travail demandé

A. La première partie du travail consiste à écrire une bibliothèque permettant des entrées-sorties bit à bit.

A.1. Définir un type `bit` permettant de représenter un bit.

A.2. Définir un type `fic_bin` ainsi que des fonctions

```
- fic_bin *open_fic_bin(char *name, const char *mode);,  
- int close_fic_bin(fic_bin *f);,  
- int write_fic_bin(fic_bin *f, bit b);,  
- int read_fic_bin(fic_bin *f, bit *b);
```

permettant d'effectuer des entrées-sorties dans un fichier bit à bit. La structure `fic_bin` représente un fichier ouvert et permet en plus de mémoriser les bits dont on a demandé la lecture ou l'écriture, mais en nombre inférieur à 8, donc ne formant pas encore un octet complet. Le mode d'ouverture est donné par le second argument de `open_fic_bin`.

A.3. Compiler cette bibliothèque dynamique sous le nom `libficbin.so` (cf. section 2.3 partie D pour la compilation et l'utilisation de la bibliothèque).

B. Implémenter l'algorithme de calcul de l'arbre, et écrire les fonctions de compression et de décompression utilisant les fonctions de `libficbin.so`. Il est nécessaire d'écrire d'autres fonctions intermédiaires : écriture d'un code dans un fichier binaire, manipulation de la fenêtre, etc. Il faut veiller à une répartition correcte du code dans plusieurs fichiers, selon les tâches réalisées.

C. Écrire le code de deux commandes permettant de compresser et de décompresser les fichiers, de synopsis :

```
compresser [-num] [-k] fichier_source fichier_dest  
decompresser fichier
```

Les exécutables `compresser` et `decompresser` devront être identiques. L'option `-num` permet de donner au programme le nombre de bits sur lequel on désire coder chaque phrase, dans le cas d'un code de taille fixe (`num` est alors un entier compris entre 9 et 15). L'option `-k` demande de garder le dictionnaire existant s'il est plein au lieu d'en reconstruire un nouveau.