PROGRAMMATION C

TP 3 – 16 octobre 2025 – durée 4h

Codage de Huffman

Présentation

Le codage de Huffman est un algorithme de **compression**. Il lit un fichier d'entrée, à partir duquel il calcule un *code binaire* pour chaque caractère présent dans le fichier. Autrement dit, il associe une *suite de bits* à chaque caractère du fichier. Ensuite, il écrit la suite des *codes binaires* de chaque caractère du fichier, pour obtenir un *fichier compressé* (de taille plus petite que le fichier original), et *décodable* (c'est-à-dire, qui permet de reconstruire l'original).

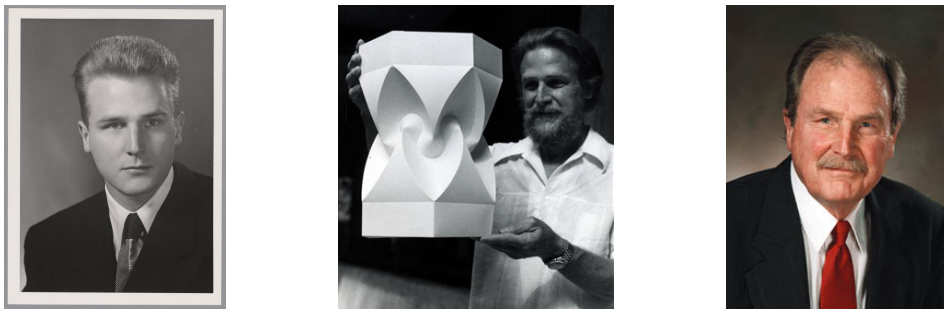


FIGURE 1 : David Albert Huffman, 9/8/1925–7/10/1999, à différentes périodes de sa vie

Une première particularité de ce codage est que la *suite de bits associée à chaque caractère dépend du fichier* d'entrée : si on compresse deux fichiers différents, un même caractère présent dans les deux fichiers n'aura pas forcément le même codage dans chacun des fichiers compressés.

Une autre particularité du codage de Huffman est que les codes utilisés sont de ***longueur variable*** : *plus un symbole est fréquent* dans le fichier source, plus le code qui lui est associé sera *court*.

Cet algorithme, à la fois naturel et très joli, a eu de multiples utilisations et extensions.

Objectif du TP

L'objectif de ce TP est d'écrire un programme servant à compresser et à décompresser des fichiers, en utilisant l'algorithme de Huffman. Ce TP est divisé en quatre parties réparties sur deux semaines.

- **Partie 1** : Implémentation des **tas binaires**. L'algorithme de compression utilise une *file de priorité* implémentée par un *tas binaire* pour associer un code à chaque symbole. La définition des tas binaires est détaillée ci-dessous.
- **Partie 2** : Construction, sauvegarde et lecture d'un **arbre de Huffman**. Cet objet est utilisé par l'algorithme pour associer un code à chaque symbole. Dans cette partie, nous implémenterons les arbres de Huffman. En particulier, c'est ici que nous utiliserons les *tas binaires* : la procédure qui construit un arbre de Huffman à partir d'un fichier à compresser utilise une file de priorité.
- **Partie 3** : Écriture de l'algorithme de **compression**.
- **Partie 4** : Écriture de l'algorithme de **décompression**.

Préliminaires

Commencez par télécharger l'archive `tp-huffman.tgz` et désarchivez-la.

```
$ tar xfvz tp-huffman.tgz
# ou, si le fichier a été téléchargé comme tp-huffman.tar :
$ tar xfv tp-huffman.tar
```

L'archive contient les fichiers sources que vous allez devoir modifier :

- `main.c` est le fichier contenant la fonction `main()` du programme. Il vous servira à écrire les tests de vos fonctions. Il contient déjà plusieurs tests. Vous pouvez les commenter, ajouter les vôtres, etc.
- `binheap.h` et `binheap.c` sont les fichiers dédiés à l'implémentation des tas binaires. Le type et les prototypes sont déjà écrits dans `binheap.h`. Vous devez écrire `binheap.c`.
- `hufftree.h` et `hufftree.c` sont les fichiers dédiés aux arbres de Huffman. Le type et les prototypes sont déjà écrits dans `hufftree.h`. Vous devez écrire `hufftree.c`.
- `huffcomp.h` et `huffcomp.c` sont les fichiers dédiés à l'algorithme de compression. Le type et les prototypes sont déjà écrits dans `huffcomp.h`. Vous devez écrire `huffcomp.c`.
- `huffdecomp.h` et `huffdecomp.c` sont les fichiers dédiés à l'algorithme de décompression. Le type et les prototypes sont déjà écrits dans `huffdecomp.h`. Vous devez écrire `huffdecomp.c`.

L'archive `tp-huffman.tgz` contient également plusieurs fichiers sources **déjà écrits** que vous pouvez utiliser :

- `dequeue.h` et `dequeue.o` contient une implémentation de « tuyaux étendus » (en utilisant des tableaux). Elle est déjà compilée (le second fichier est « *.o* », pas un « *.c* »). Vous pouvez l'utiliser pour écrire les fonctions pour lesquelles les tuyaux sont utiles. Bien sûr, vous pouvez également utiliser votre propre implémentation de files ou de piles. Deux versions compilées sont fournies :
  - `dequeue_arm64.o`, version compilée pour MacOS.
  - `dequeue_x86-64.o`, version compilée pour Linux.

  Pour utiliser la version MacOS `dequeue_arm64.o`, sous le shell :

  ```
  $ rm -f dequeue.o; ln -s dequeue_arm64.o dequeue.o
  ```

  et, de façon similaire, pour utiliser la version Linux `dequeue_x86-64.o` :

  ```
  $ rm -f dequeue.o; ln -s dequeue_x86-64.o dequeue.o
  ```
- `testprint.h` et `testprint.c` : ces fichiers contiennent des fonctions que vous pouvez utiliser pour afficher et tester les objets considérés dans ce TP (comme les tas binaires, les arbres de Huffman, …).
- `alloc.h`, `alloc.c` et `error.h`. Ces fichiers donnent accès aux macros pour l'allocation mémoire. Il n'est pas obligatoire de les utiliser. Attention, cependant à inclure `alloc.c` dans vos compilations : certaines des fonctions déjà écrites l'utilisent.
- `Makefile`, pour compiler avec `make`.

```
$ make                  # erreur sur warning, pas de vérification de variables inutilisées
$ make no-error-on-warn # pas d'erreur sur warning, mais vérification de variables inutilisées
$ make clean            # nettoyage, à ne faire que si on est maniaque
```

Une fois la compilation réussie, l'exécutable construit peut être lancé avec l'option `--help`.
Il permet de tester les fonctions spécifiques au tas, aux arbres, à la compression et la décompression.

⚠ Pensez à tester votre code *régulièrement*

Vous devrez écrire des tests. Pour cela, vous disposez de plusieurs fonctions d'affichage déjà écrites dans le fichier `testprint.c` qui sont à utiliser régulièrement. En particulier, les deux fonctions suivantes vous seront utiles pour les deux premières parties :

```
void print_binheap(binheap *);     // Affichage d'un tas binaire
void print_hufftree(huffnode *);   // Affichage d'un arbre de Huffman
```

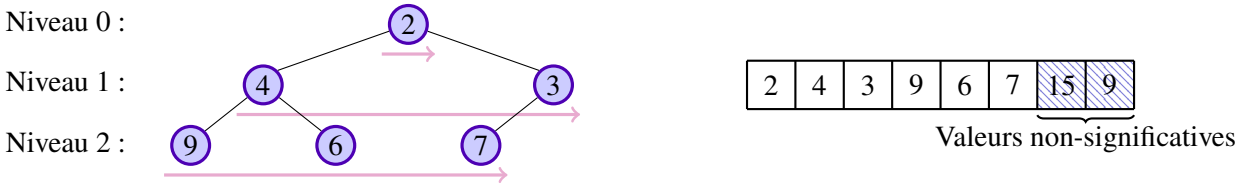
1 Partie 1 : Tas binaires

On souhaite implémenter les tas binaires *min*, comme ceux vus en algorithmique. Les fichiers `binheap.h` et `binheap.c` sont dédiés à cette implémentation. Comme d'habitude, `binheap.h` est déjà écrit et contient le type et les prototypes des fonctions que vous devez programmer dans `binheap.c`.

Commençons par décrire la structure de données que nous allons utiliser. Il est standard de représenter un tas binaire en utilisant un tableau qui va contenir toutes les valeurs se trouvant dans le tas *rangées dans un ordre bien précis*. Plus précisément, on représente un tas de taille k avec une structure de données qui contient les trois objets suivants :

1. Un tableau dont la taille allouée est une puissance de deux supérieure ou égale à k. Les valeurs du tas y sont stockées « niveau par niveau » : on écrit d'abord la valeur de la racine (le niveau 0), puis les valeurs de ses deux enfants (le niveau 1), puis les quatre valeurs de leurs enfants (le niveau 2), …
2. La taille c de ce tableau,
3. La taille k du tas, c'est-à-dire le nombre de valeurs significatives dans le tableau. On a donc $k \leqslant c$. Ces valeurs significatives sont stockés consécutivement en début de tableau.

On décrit cette représentation graphiquement dans la Figure 2 ci-dessous.



Tas binaire à représenter — Représentation par un tableau de taille 8.

FIGURE 2 : Représentation d'un tas binaire (d'entiers) par un tableau

De plus, les tas que nous utiliserons plus tard pour l'algorithme de Huffman ne contiendront pas des entiers mais des objets plus compliqués (spécifiquement, des nœuds d'un arbre de Huffman : nous verrons ce que c'est plus loin). Pour cette raison, nous allons devoir écrire une implémentation « générique » des tas binaires : le type des valeurs que peuvent contenir nos tas n'est pas fixé. Cela a deux conséquences :

1. Nous allons *utiliser des pointeurs génériques* : le tableau contiendra en fait des éléments de type `void *`, qui sont des pointeurs sur des objets dont le type n'est pas précisé. Ils pointeront vers les valeurs du tas à représenter (allouées en mémoire autre part).
2. Puisque le type des valeurs n'est pas fixé, la *fonction de comparaison* qui ordonne ces valeurs devra être un paramètre du tas. En C, il est possible de définir un tel paramètre en utilisant un *pointeur de fonction*. Un des champ de notre type sera donc un pointeur sur une fonction qui associe un Booléen à deux éléments de type `void *` (qui indique si le premier est considéré comme « plus petit » que le deuxième).

On va donc utiliser le type suivant en C (celui-ci est déjà écrit dans `binheap.h`).

```c
typedef struct{
  void **array;                 // Tableau stockant des pointeurs vers chaque valeur du tas.
  int size_array;               // Taille totale du tableau.
  int size_heap;                // Taille du tas (nombre de cases utilisées dans le tableau).
  bool (*fc) (void *, void *);  // Pointeur sur la fonction de comparaison utilisée par le tas.
} binheap;
```

Comme d'habitude, on commence par écrire quelques fonctions auxiliaires qui seront utiles pour implémenter les primitives des files de priorité. Tout d'abord, nous aurons besoin de fonctions pour « naviguer » à l'intérieur d'un tas binaire.

```c
// Retourne l'indice de l'enfant gauche du noeud à l'indice i.
int left_binheap(int i);

// Retourne l'indice de l'enfant droit du noeud à l'indice i.
int right_binheap(int i);

// Retourne l'indice du parent du noeud à l'indice i.
int parent_binheap(int i);

// Teste si l'indice i correspond à un noeud valide dans le tas p.
bool isvalid_binheap(binheap *p, int i);
```

Naturellement, nous aurons aussi besoin de fonctions auxiliaires pour modifier la taille du tableau utilisé dans la représentation quand c'est nécessaire.

```c
// Double la taille du tableau utilisé dans la représentation.
static void grow_binheap(binheap *);

// Divise par deux la taille du tableau utilisé dans la représentation.
static void shrink_binheap(binheap *);
```

Enfin, la liste des fonctions primitives à implémenter pour les tas binaires est la suivante :

```c
// Création d'un tas vide
// Le paramètre est un pointeur de fonction donnant la fonction de comparaison à utiliser.
binheap *create_binheap(bool (*) (void *, void *));

// Suppression
void delete_binheap(binheap *);

// Test du vide
bool isempty_binheap(binheap *);

// Taille
int getsize_binheap(binheap *);

// Insertion d'une valeur
void push_binheap(binheap *, void *);

// Récupération du minimum sans le retirer
void *peekmin_binheap(binheap *);

// Récupération du minimum en le retirant
void *popmin_binheap(binheap *);
```

⚠ Pensez à tester votre code *régulièrement*

Vous devez écrire des tests pour vérifier votre implémentation. En particulier, vous disposez de fonctions déjà écrites et pensées pour les tests. Leurs prototypes sont décrits dans `testprint.h` et les fonctions elle-mêmes sont écrites dans `testprint.c` :

```
// Génération d'un tas binaire à partir d'un tableau d'entiers.
// Le tas contiendra des pointeurs vers les cases du tableau
binheap *array_to_binheap(int *array, int size);

// Affichage d'un tas binaire.
// Prévu pour un tas dont les éléments sont de type int *.
void print_binheap(binheap *);
```

Plus de détails sur l'utilisation de ces fonctions sont disponibles dans `testprint.h`.

Enfin, une bonne façon de tester votre implémentation des tas est d'écrire une fonction qui trie un tableau en utilisant un tas. La fonction commence par insérer tous les éléments du tableau dans un tas et les ressort ensuite dans l'ordre. Cette fonction est déjà écrite dans `main.c`.

```
// Tri d'un tableau par un tas binaire
void heap_sort(int *array, int size);
```

2 Partie 2 : Arbres de Huffman

Nous allons maintenant considérer le codage de Huffman lui-même. Comme expliqué au début de la feuille, l'algorithme de compression prend en entrée un fichier source et fonctionne en deux étapes :

1. Il associe un *code en binaire* à chaque symbole présent dans le fichier source. Par « symbole », on veut dire chaque valeur que peut prendre un octet dans le fichier[1]. Pour chaque symbole, son code dépend de sa fréquence dans le fichier : plus il est fréquent, plus le code sera court.
2. Le fichier compressé est ensuite généré à partir du fichier source en remplaçant chaque symbole par le code qui lui a été associé à la première étape. L'algorithme enregistre également des informations supplémentaires nécessaires pour la décompression (par exemple, on doit enregistrer l'association entre symboles et codes).

Dans cette partie, on va surtout s'intéresser à la première étape. La compression en elle-même sera le sujet de la troisième partie. On va commencer par illustrer l'algorithme qui construit les codes par un exemple.

Regardons la phrase ‘‘`exemple de codage de huffman`’’. On veut associer un *code en binaire* à tous les symboles présents dans cette phrase. Pour cela, l'algorithme commence par compter le nombre d'occurrences de chaque symbole présent dans cette phrase pour le stocker dans un tableau de fréquences. En pratique, l'algorithme qu'on va écrire manipulera un tableau de fréquences dont la taille est *256* (une case par valeur possible d'un octet). Pour l'exemple, on ne va représenter que les symboles qui apparaissent effectivement dans la phrase à compresser (ceux dont le nombre d'occurrences est strictement supérieur à zéro). Pour la phrase ‘‘`exemple de codage de huffman`’’ on obtient le tableau de fréquences décrit dans la Figure 3.

Texte à coder : ‘‘`exemple de codage de huffman`’’

Tableau des fréquences :

e	x	m	p	l	_	d	c	o	a	g	h	u	f	n
6	1	2	1	1	4	3	1	1	2	1	1	1	2	1

FIGURE 3 : Construction du tableau de fréquences à partir d'un texte à coder

1. Dans les exemples, on considère des fichiers texte. Cela nous permet de voir les symboles présents comme des caractères ASCII plutôt que des valeurs entre 0 et 255

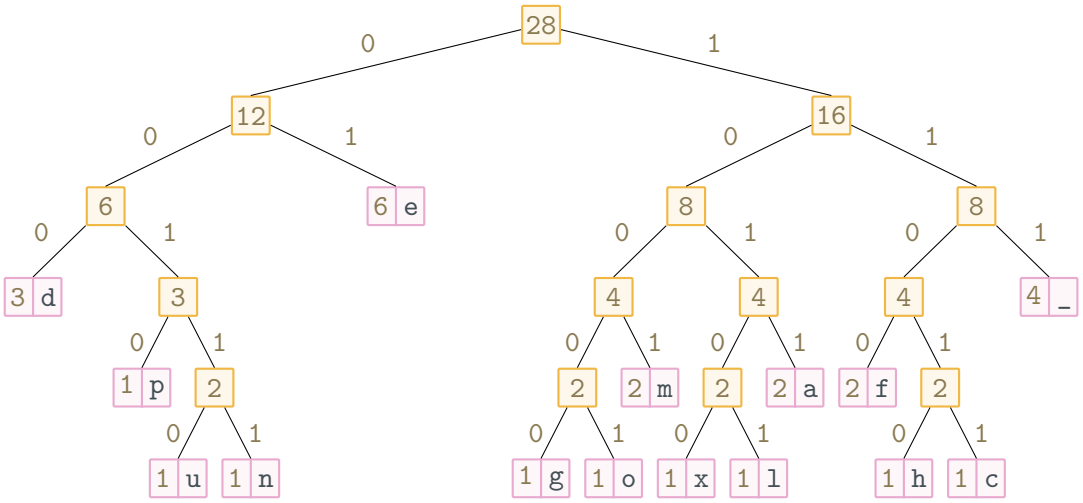
Une fois le tableau de fréquences construit, on utilise un algorithme basé sur les files de priorité (celui-ci est décrit plus bas) pour construire un *arbre de Huffman*. C'est un arbre *complet* (tous les nœuds ont soit deux enfants, soit zéro enfant) qui satisfait les propriétés suivantes :

- Les feuilles correspondent aux symboles présents dans le fichier à compresser. Pour chaque symbole, l'arbre doit contenir une unique *feuille* qui porte *deux valeurs* : le symbole lui-même et la fréquence de celui-ci dans le fichier.
- En revanche, les *nœuds internes* ne portent qu'*une seule valeur* : une fréquence. Elle doit être égale à la somme des fréquences de ses deux enfants. Les valeurs des nœuds internes sont donc déterminées par les fréquences des feuilles.

L'arbre de Huffman est conçu pour enregistrer les codes associés aux symboles. Plus précisément, une fois que cet arbre est construit, on obtient le code en binaire d'un symbole en regardant le chemin qui mène de la racine de l'arbre à l'unique feuille contenant ce symbole. En effet, puisque nous regardons un arbre binaire, un tel chemin consiste à descendre dans l'arbre en prenant à chaque nœud interne, soit son enfant gauche, soit son enfant droit. On peut donc associer naturellement un code en binaire à un tel chemin :

- Prendre un enfant gauche correspond à un `0`.
- Prendre un enfant droit correspond à un `1`.

On représente l'arbre de Huffman généré à partir de la phrase ‘‘`exemple de codage de huffman`’’ dans la Figure 4 ci-dessous.



Arbre de Huffman

d	000
p	0010
u	00110
n	00111
e	01
g	10000
o	10001
m	1001
x	10100
l	10101
a	1011
f	1100
h	11010
c	11011
_	111

Codes associés

FIGURE 4 : Arbre de Huffman construit à partir du tableau de fréquences de la Figure 3

2.1 Implémentation des arbres de Huffman en C

Nous devons d'abord choisir comment nous allons représenter les arbres de Huffman en C. On va utiliser une approche similaire à l'implémentation standard des listes chaînées et représenter les nœuds de nos arbres individuellement. Un nœud donné contiendra les informations suivantes :

1. Sa fréquence.
2. Le symbole auquel il correspond (cette information ne sera significative que si le nœud est une feuille).
3. Un pointeur sur son enfant gauche (égal à `NULL` si il n'a pas d'enfant gauche).
4. Un pointeur sur son enfant droit (égal à `NULL` si il n'a pas d'enfant droit).

En C, nous utiliserons donc le type suivant (celui-ci est déjà écrit dans `hufftree.h`) :

```c
struct huffnode{
  // Fréquence.
  int freq;

  // Octet codé par le noeud (significatif seulement si le noeud est une feuille).
  int byte;

  // Enfant gauche (NULL si pas d'enfant gauche).
  struct huffnode *leftchild;

  // Enfant droit (NULL si pas d'enfant droit).
  struct huffnode *rightchild;
};
```

Pour commencer, vous allez devoir écrire quelques fonctions basiques qui vont vous permettre de manipuler les arbres de Huffman dans l'algorithme principal. Celles-ci sont à écrire dans `hufftree.c`.

```c
// Création d'une feuille.
huffnode *create_huffleaf(int byte,int freq);

// Fusion de deux arbres non-vides en utilisant un nouveau noeud racine.
// La fréquence du nouveau noeud devra être la somme de celles de ces deux enfants
// Le champ byte du nouveau noeud n'est pas significatif (ce n'est pas une feuille)
huffnode *merge_hufftree(huffnode *left, huffnode *right);

// Teste si un arbre est réduit à une seule feuille.
bool isleaf_huffnode(huffnode *p);

// Retourne la valeur de l'octet codé dans un noeud.
int getbyte_huffnode(huffnode *p); // Ne doit être utilisé que sur les feuilles

// Retourne les enfants d'un noeud.
huffnode *getleft_huffnode(huffnode *p);
huffnode *getright_huffnode(huffnode *p);

// Libération complète d'un arbre de Huffman.
void free_hufftree(huffnode *);
```

⚠ Pensez à tester votre code *régulièrement*

Comme toujours, vous devez écrire des tests. Vous disposez d'une fonction déjà écrite dans `testprint.h` et `testprint.c` pour afficher un arbre de Huffman (les fréquences sont omises) :

```c
void print_hufftree(huffnode *);
```

2.2 Génération d'un arbre de Huffman à partir d'un fichier

Nous sommes prêts à programmer l'algorithme qui construit l'arbre de Huffman à partir du fichier à compresser. Celui-ci utilise une file de priorité dont les éléments sont des arbres de Huffman. Les arbres seront ordonnés selon la fréquence de leur racine : étant donnés deux arbres, le « plus petit » sera celui dont la racine a la plus petite fréquence.

Étant donné un fichier d'entrée « `input` » à compresser, l'algorithme utilisé pour calculer l'arbre de Huffman associé procède en trois étapes consécutives :

1. **Calcul du tableau de fréquences** : On construit un tableau de taille 256 à partir du fichier `input`. Pour chaque indice i tel que $0 \leqslant i \leqslant 255$, la i-ème case doit contenir la fréquence du « symbole » i dans `input` : c'est-à-dire le nombre d'octets qui ont la valeur i.

2. **Initialisation de la file de priorité** : Pour chaque symbole i dont la fréquence est non nulle, on crée un arbre de Huffman réduit à une unique feuille contenant ce symbole i et sa fréquence. On insère tous ces arbres de Huffman dans une file de priorité initialement vide.

   À la fin de cette étape, la file de priorité contient donc autant d'arbres qu'il y a de symboles distincts dans le fichier. De plus, tous ces arbres sont réduits à une unique feuille.
3. **Construction de l'arbre** : Tant que la file de priorité contient **au moins deux arbres** on effectue les opérations suivantes :
   - On prend dans la file de priorité les deux arbres t_1 et t_2 de plus petites fréquences.
   - On fusionne t_1 et t_2 en utilisant un nouveau nœud racine dont les enfants sont t_1 et t_2 (par définition des arbres de Huffman, la fréquence de cette racine est la somme de celles de t_1 et t_2).
   - On ajoute le nouvel arbre qu'on vient de construire dans la file de priorité.

Après la dernière étape, la file de priorité contient un seul arbre : c'est l'arbre de Huffman voulu.

Vous allez maintenant devoir implémenter cet algorithme en C. Plus précisément, les fonctions à écrire sont les suivantes :

```c
// Fonction de comparaison entre deux arbres de Huffman, à utiliser pour le tas binaire.
// Le plus petit de deux arbres est celui dont la racine a la plus petite fréquence.
bool compare_hufftree(void *, void *);

// Création de l'arbre de Huffman à partir du fichier à compresser.
// Le flux input doit être ouvert en lecture.
huffnode *datafile_to_hufftree(FILE *input);
```

2.3 Sauvegarde et lecture d'un arbre de Huffman dans un fichier

Puisque l'arbre de Huffman utilisé détermine l'association entre symboles et codes, nous en aurons besoin aussi bien pour la compression que pour la décompression. Il faudra donc qu'on *sauvegarde* l'arbre de Huffman à l'intérieur du fichier compressé lors de la compression et qu'on *lise* cet arbre sauvegardé lors de la décompression.

Pour sauvegarder un arbre de Huffman à l'intérieur d'un fichier, on va y écrire la liste de ses nœuds dans l'ordre préfixe. Le *parcours préfixe* d'un arbre se décrit de manière récursive :

1. On commence par écrire la racine.
2. On utilise la récursion pour écrire les nœuds contenus dans l'enfant gauche.
3. On utilise la récursion pour écrire les nœuds contenus dans l'enfant droit.

Le parcours préfixe peut aussi se décrire graphiquement. On présente un exemple dans la Figure 5.

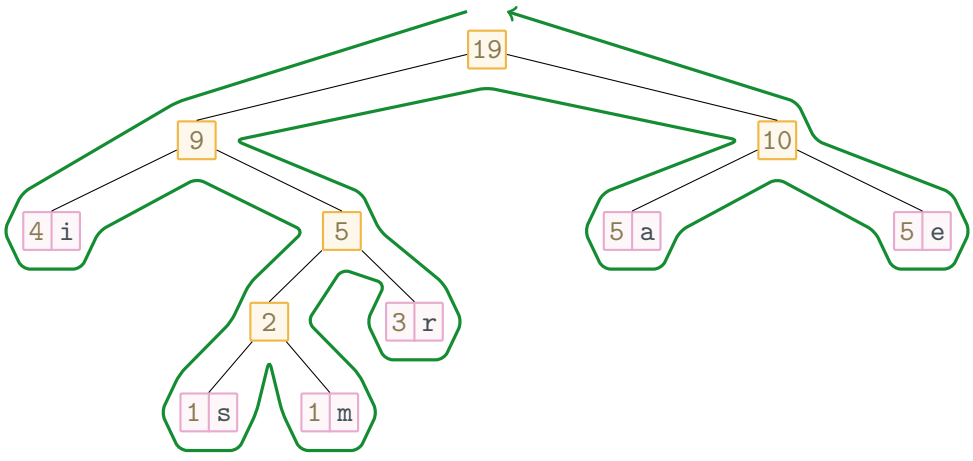


FIGURE 5 : Parcours préfixe : les nœuds sont écrits la première fois que la ligne passe à côté d'eux.

De plus, les fréquences enregistrées dans les nœuds sont inutiles pour la décompression (elles servent uniquement pour la construction de l'arbre de Huffman que nous avons programmée plus haut). Nous pouvons donc les omettre dans notre sauvegarde :

- Pour écrire un nœud interne, on écrira seulement le caractère « `N` » (valeur 78 en ASCII).
- Pour écrire une feuille, on écrira le caractère « `L` » (valeur 76 en ASCII) suivi du symbole de cette feuille.

Par exemple, pour sauvegarder l'arbre décrit dans la Figure 5, on écrira la séquence de caractères suivante dans le fichier de destination :

```
NNLiNNLsLmLrNLaLe
```

Vous allez maintenant devoir implémenter les algorithmes de sauvegarde et de lecture en C. Dans les deux cas, il est fortement recommandé (mais pas obligatoire) d'écrire des procédures *récursives*.

```c
// Écriture de l'arbre de Huffman dans le futur fichier compressé.
// Le flux output doit être ouvert en écriture.
// On stocke l'arbre par une représentation préfixe.
void save_hufftree(huffnode *p, FILE *output);

// Lecture de l'arbre de Huffman dans un fichier.
// Le flux in doit être ouvert en lecture.
// La tête de lecture doit être positionnée au début du codage de l'arbre.
// Puisque la sauvegarde ne stocke pas les fréquences, elles seront toutes
// égales à zéro dans l'arbre construit.
huffnode *read_hufftree(FILE *in);
```

3 Partie 3 : Compression d'un fichier

Nous en avons déjà écrit une partie de la procédure de compression dans la Partie 2 : nous avons désormais des fonctions pour construire de l'arbre de Huffman à partir du fichier source et le sauvegarder. Il nous reste à écrire le programme qui génère le codage du fichier source en remplaçant chaque symbole par le code que lui associe l'arbre. Cette procédure ne travaille pas directement avec un arbre de Huffman : étant donné un symbole, il est compliqué de récupérer le code associé à partir de cet arbre[2]. Une première étape va donc être de construire à partir de l'arbre un objet pensé pour cette opération : un *dictionnaire de Huffman*.

3.1 Construction du dictionnaire de Huffman

Un *dictionnaire de Huffman* est un tableau de taille 256 : chaque case correspond à un symbole (c'est-à-dire à une valeur que peut prendre un octet). Pour tout indice i tel que $0 \leqslant i \leqslant 255$, on aura deux cas :

- Si le symbole de valeur i n'a pas de code associé (ce qui se produit quand ce symbole n'apparait pas dans le fichier à compresser), la i-ème case du dictionnaire contiendra `NULL`.
- Si le symbole de valeur i est associé à un code, la i-ème case du dictionnaire contiendra ce code sous la forme d'un tableau de `0` et de `1` (on utilisera le type `unsigned char` pour les éléments de ce tableau).

Nous allons donc utiliser les types suivants dans notre implémentation en C :

2. En revanche, l'opération inverse est simple : étant donné un code, on peut facilement récupérer le symbole associé en lisant l'arbre. Ce sera utile plus tard pour la décompression.

```c
// Type pour représenter un seul code
typedef struct {
  // Le nombre de bits dans le code (un code n'aura jamais plus de 256 bits).
  unsigned char size;

  // Le tableau qui stocke le code : les valeurs sont toutes 0 ou 1.
  unsigned char *code;
} huffcode;

// Le dictionnaire contiendra tous les codes.
// Il sera de type huffcode **, et pointera vers 256 cases.
// Chacun de ses éléments sera donc de type huffcode *.
// Pour les valeurs i ayant un code, la ième case contient ce code
// Pour les valeurs i n'ayant pas de code (de fréquence nulle), la ième case
// contient NULL.
```

Nous allons maintenant devoir programmer une fonction qui construit le dictionnaire de Huffman à partir de l'arbre Huffman (lui-même calculé à partir du fichier source à compresser).

```c
// Construction du dictionnaire à partir de l'arbre de Huffman.
huffcode **create_huffdict(huffnode *);

// Suppression du dictionnaire.
void delete_huffdict(huffcode **);
```

Indication

On pourra utiliser une fonction auxiliaire **récursive** qui manipule un tuyau :

```c
void fill_huffdict(huffnode *tree, dequeue *deq, huffcode **dict);
```

La récursivité permet de descendre dans l'arbre et le tuyau sert à mémoriser la séquence de bits qu'on a rencontrée aux appels récursifs précédents. Ainsi, quand on arrive sur une feuille, le tuyau contient le code du symbole correspondant à cette feuille (qu'on peut donc écrire dans le dictionnaire).

⚠ Pensez à tester votre code *régulièrement*

Encore une fois, vous devez écrire des tests pour tester vos fonctions. Ici aussi, vous disposez d'une fonction déjà écrite dans les fichiers `testprint.h` et `testprint.c` qui vous permet d'afficher un dictionnaire :

```c
void print_huffdict(huffcode **);
```

Vous pouvez l'utiliser pour comparer le dictionnaire que votre fonction construit avec l'arbre de Huffman d'origine.

3.2 Compression

Nous sommes maintenant prêts à programmer la fonction de compression. On choisit de structurer les fichiers compressés en trois segments consécutifs qui contiennent des informations distinctes :

- **Segment 1** : Les 4 premiers octets du fichier enregistrent la taille du fichier original.
- **Segment 2** : La sauvegarde de l'arbre de Huffman qui a été utilisé pour la compression.
- **Segment 3** : Le codage du fichier original obtenu en remplaçant chaque symbole par son code.

Pour le **Segment 1**, notre compresseur devra simplement calculer la taille du fichier source et l'écrire avec la fonction suivante (que vous devez programmer) :

```c
// Écriture d'un unsigned int sur 4 octets dans un fichier.
void write_size(FILE *output, unsigned int size)
```

Pour le **Segment 2**, nous avons déjà écrit la fonction `void save_hufftree(huffnode *, FILE*)` qui sauvegarde un arbre de Huffman en Partie 2. On pourra donc l'utiliser dans notre compresseur.

En revanche, il y a une difficulté concernant l'écriture du **Segment 3** : celui-ci ne respecte pas la structuration classique d'un fichier en octets. En effet, le code de Huffman associé à un symbole est une séquence de bits *de longueur arbitraire*. Ainsi, si le code d'un symbole est composé de k bits, on souhaite utiliser exactement k bits pour l'écrire dans le fichier compressé. Par exemple, reprenons la phrase ''`exemple de codage de huffman`''. Comme nous l'avons vu dans la Figure 4, les codes des symboles `e`, `x`, `m`, `p` et `l` sont `01`, `10100`, `1001`, `0010` et `10101` respectivement. Pour cette raison, le codage du premier mot ''`exemple`'' utilisera exactement 24 bits (soit 3 octets) comme décrit dans la Figure 6 ci-dessous.

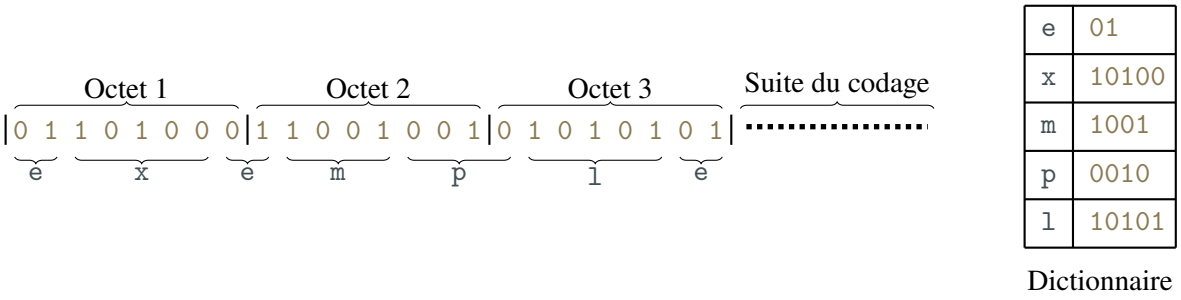


FIGURE 6 : Codage du début de la phrase ''`exemple de codage de huffman`''

Cette situation pose un problème technique. En C, le grain le plus fin pour écrire dans un fichier est l'octet : on ne peut pas écrire « bit par bit ». Comme le montre la Figure 6, il n'est donc pas possible d'écrire les codes des symboles du fichier source directement et un par un dans le fichier compressé. On va résoudre ce problème en utilisant un tuyau « *tampon* » dont les valeurs sont des bits (c'est-à-dire `0` ou `1`) :

- Au lieu d'écrire les codes directement dans le fichier compressé, on insère d'abord leurs bits dans ce tuyau tampon. À chaque fois qu'un nouveau symbole est lu dans fichier source, tous les bits du code qui sont associés à ce symbole par le dictionnaire sont insérés à droite du tuyau tampon.
- Dès que le tuyau tampon contient au moins 8 bits (c'est-à-dire un octet complet), on prend les 8 premiers bits à sa gauche et on écrit l'octet correspondant dans le fichier compressé.

Attention, il peut arriver qu'à la fin de cette procédure, le tuyau tampon contienne encore entre 1 et 7 bits. Dans ce cas, il faudra insérer suffisamment de zéros à droite du tuyau tampon pour faire un octet complet et inscrire ce dernier octet dans le fichier compressé. C'est pour cette raison qu'il est nécessaire d'écrire la taille du fichier source au début du fichier compressé : quand nous écrirons le décompresseur, il faudra pouvoir différencier ces zéros supplémentaires d'un « vrai » code de Huffman.

On commence par écrire deux fonctions auxiliaires qui nous seront utiles pour manipuler notre tuyau tampon dans la fonction principale.

```c
// Insertion de tous les bits d'un code à droite d'un tuyau tampon.
// Le tuyau est supposé contenir une séquence de 0 et de 1.
void insert_code_in_buffer(dequeue *, huffcode *);

// Calcule l'octet correspondant aux 8 premiers bits à gauche d'un tuyau tampon.
// Le tuyau est supposé contenir une séquence de bits (0 ou 1) de longueur au moins 8.
// Les 8 bits lus doivent être supprimés du tuyau.
int buffer_to_byte(dequeue *);
```

Nous pouvons maintenant programmer la fonction de compression. Celle-ci doit d'abord calculer la taille du fichier source, l'arbre de Huffman et le dictionnaire de Huffman associés. Ces trois objets doivent ensuite être utilisés pour écrire les trois segments décrits ci-dessus dans le fichier compressé.

```c
// Fonction de compression du fichier input dans output.
//
// input doit être ouvert en lecture.
// output doit être ouvert en écriture.
// Le fichier output devra contenir dans l'ordre :
// 1) La taille de input codée sur 4 octets.
// 2) La sauvegarde de l'arbre de Huffman obtenu à partir de input.
// 3) Le codage de input.
void compress_file(FILE *input, FILE *output);
```

4 Partie 4 : Algorithme de décompression

Comme nous l'avons vus dans la Partie 3, les fichiers compressés sont structurés en trois segments consécutifs. Pour décompresser, nous allons donc devoir lire ces trois segments, les interpréter et utiliser l'information qu'ils contiennent pour décoder le fichier.

Par définition, le **Segment 1** occupe exactement les quatre premiers octets du fichier compressé. Il code la taille du fichier source original et s'interprète donc comme un entier non signé. On va commencer par écrire une fonction qui lit ces quatre octets et retourne l'entier non signé qu'ils codent.

```c
// Lecture de la taille du fichier décompressé dans le fichier compressé.
//
// Le flux doit être ouvert en lecture.
// La taille est codée sur les 4 premiers octets.
unsigned int read_size(FILE *in);
```

Le **Segment 2** contient la sauvegarde de l'arbre de Huffman qui a été utilisé pour compresser le fichier source original. Le nombre d'octets qu'ils occupe n'est pas fixé. Nous avons déjà écrit une fonction `huffnode *read_hufftree (FILE *)` qui permet de lire un arbre de Huffman et donc de décoder ce segment dans la Partie 2. Nous allons donc pouvoir utiliser cette fonction dans notre décompresseur.

Le **Segment 3** contient le codage du fichier source original. C'est cette partie que nous allons devoir décoder pour restaurer le fichier source. Pour cela nous allons utiliser les informations fournies par les deux premiers segments. Commençons par expliquer comment décoder le premier symbole. Cette opération consiste à lire un à un les bits à partir du début du **Segment 3** tout en parcourant l'arbre de Huffman à partir de la racine. Les bits qu'on lit déterminent le chemin pris dans l'arbre :

- Si on lit un `0`, on descend dans l'arbre en prenant l'enfant gauche.
- Si on lit un `1`, on descend dans l'arbre en prenant l'enfant droit.

Éventuellement, le parcours de l'arbre arrive dans une feuille. Par définition de l'arbre de Huffman, cela signifie qu'on a fini de lire le premier code et que le symbole auquel il correspond est celui enregistré dans la feuille qu'on a atteint. On peut donc maintenant écrire ce symbole dans le fichier décompressé : nous avons décodé le premier symbole. Il suffit ensuite de répéter cette opération autant de fois que nécessaire afin de *décoder tous les symboles*. On notera qu'on sait à l'avance le nombre de symboles à décoder (et donc le nombre de répétitions de cette opération) puisque nous connaissons la taille du fichier source original.

Il reste néanmoins un problème technique similaire à celui que nous avons rencontré pour la compression. En C, il n'est pas possible de lire un fichier bit par bit. Il est *obligatoire de lire des octets entiers*. On va à nouveau se servir d'un tuyau « *tampon* » dont les valeurs sont des bits (c'est-à-dire `0` ou `1`) pour résoudre ce problème. Comme on ne peut lire que des octets entiers, les bits du **Segment 3** sont insérés dans ce tuyau par groupes de huit. On peut ensuite les ressortir un par un pour le décodage. Le principe est le suivant :

- Tant que le tuyau tampon est non-vide, on lit les codes bit par bit à l'intérieur de celui-ci.
- Dès que le tuyau tampon est vide, on lit un nouvel octet dans le **Segment 3** et on insère ses 8 bits dans le tuyau tampon. On repasse ensuite à l'étape précédente.

On va d'abord écrire une fonction auxiliaire qui lit un seul code de Huffman et retourne le symbole correspondant en utilisant ce principe.

```c
// Lecture d'un code de Huffman à partir du tampon et du fichier compressé
//
// Le fichier doit être ouvert en lecture.
// Lit un code entier et retourne la valeur qui lui est associée par l'arbre de Huffman.
// Tant que le buffer est non-vide, c'est dans celui-ci qu'on lit.
// Si le buffer est vide, on lit un octet du fichier pour insérer ses 8 bits dans le buffer
// Le buffer est potentiellement non-vide à la fin de la lecture d'un code.
// Les bits qui restent dans le buffer font partie du prochain code à lire.
int read_huffcode(FILE *in, dequeue *buffer, huffnode *thetree);
```

Nous sommes maintenant prêts à écrire la fonction complète de décompression.

```c
// Procédure de décompression.
//
// in doit être ouvert en lecture.
// out doit être ouvert en écriture.
// Décompresse le fichier in et écrit la version décompressée dans out.
void decompress_file(FILE *in, FILE *out);
```