2. Problème d'algorithmique et programmation

L'ensemble du problème est consacré à l'algorithme de Huffman qui permet de coder un texte caractère par caractère à l'aide d'une chaîne binaire en minimisant la longueur totale de la chaîne obtenue ; cet algorithme permet de faire de la compression de données. Les parties 1 et 2 du problème sont des parties préparatoires, le codage d'un texte sera abordé dans la troisième partie.

Préliminaire concernant la programmation : il faudra écrire des fonctions ou des procédures à l'aide d'un langage de programmation qui pourra être soit **Caml**, soit **Pascal**, tout autre langage étant exclu. **Indiquer en début de problème le langage de programmation choisi ; il est interdit de modifier ce choix au cours de l'épreuve**. Certaines questions du problème sont formulées différemment selon le langage de programmation ; cela est indiqué chaque fois que cela est nécessaire. Par ailleurs, pour écrire une fonction ou une procédure en langage de programmation, le candidat pourra définir des fonctions ou des procédures auxiliaires qu'il explicitera ou faire appel à d'autres fonctions ou procédures définies dans les questions précédentes.

Dans l'énoncé du problème, un même identificateur écrit dans deux polices de caractères différentes désigne la même entité, mais du point de vue mathématique pour la police écrite en italique (par exemple : *nb_arbres*) et du point de vue informatique pour celle écrite en romain (par exemple : `nb_arbres`). Pour écrire une valeur de type caractère, on représente celle-ci entre apostrophes (par exemple, 'a' pour le caractère a).

Dans tout le problème, on utilise des arbres binaires. Pour un arbre, les termes de **nœud** et de **sommet** sont synonymes ; c'est le terme de nœud qui est retenu dans ce problème. Un nœud qui n'a pas de fils est appelé **feuille** alors qu'un nœud qui a au moins un fils est appelé un **nœud interne**.

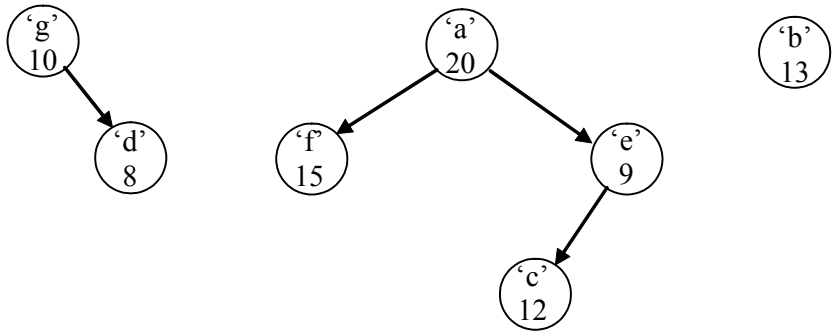
Chaque nœud *n* des arbres binaires de ce problème contient, outre les indications concernant ses éventuels fils gauche et droit (voir plus bas), un caractère appelé **lettre du nœud** et noté *lettre(n*) et un entier strictement positif appelé **poids du nœud** et noté *poids*(*n*). Ces arbres binaires sont appelés **H_arbres**.

Une **forêt** est dans ce sujet une collection de H_arbres.

Une forêt est représentée en mémoire par :

- le nombre de H_arbres de la forêt, noté *nb_arbres* ;
- le nombre total de nœuds, noté *nb_nœuds* ;
- un tableau (ou vecteur) de nœuds appelé *table* ; dans tout le problème, ce tableau est supposé suffisamment grand pour contenir les nœuds de tous les H-arbres de la forêt considérée ; les nœuds sont rangés dans le tableau entre les indices 0 et *nb_nœuds* – 1 ; ATTENTION : les nœuds qui sont les racines des H_arbres de la forêt se trouvent nécessairement au début du tableau *table*, c'est-à-dire entre les indices 0 et *nb_arbres* – 1. Pour un nœud donné, les fils gauche et droit sont indiqués par leurs indices dans le tableau *table* des nœuds ; lorsqu'un fils gauche ou droit n'existe pas, cela est indiqué par une valeur d'indice égale à –1. Le fils gauche d'un nœud *n* sera noté *fg*(*n*) et le fils droit sera noté *fd*(*n*).

Exemple introductif (notée *F_ex*) :



Pour la forêt *F_ex*, on a : *nb_arbres* = 3, *nb_nœuds* = 7 ; plusieurs tables peuvent correspondre à *F_ex*, une possibilité est :

table :

Indice du nœud	0	1	2	3	4	5	6
lettre	'g'	'a'	'b'	'e'	'f'	'c'	'd'
poids	10	20	13	9	15	12	8
fg	–1	4	–1	5	–1	–1	–1
fd	6	3	–1	–1	–1	–1	–1

Dans cette table, on voit que le nœud d'indice 3 contient la lettre 'e' et le poids 9, que son fils gauche se trouve à l'indice 5 de la table et qu'il n'a pas de fils droit.

Indications pour la programmation

Caml : On définit les identificateurs et les types suivants

```
type Noeud = {
      lettre      : char;
      poids       : int;
      fg          : int;
      fd          : int;
};;

let noeud_vide =
      {lettre = `\000`; poids = 0; fg = -1; fd = -1};;

type Foret = {
            mutable nb_arbres : int;
            mutable nb_noeuds : int;
            table             : Noeud vect;
           };;

let MAX = 100;;
```

`\000` représente le caractère de valeur nulle (caractère qui est différent du caractère `0`).

La valeur `MAX` donne un majorant du nombre total de nœuds des forêts considérées ; c'est la dimension donnée au champ `table` d'une valeur de type `Foret`.

Les types `Noeud` et `Foret` sont des types pour des enregistrements (un tel type est aussi appelé type produit). Un enregistrement contient des champs (aussi appelés composantes ou étiquettes). Une valeur de type `Noeud` contient les champs `lettre`, `poids`, `fg` et `fd` ; une valeur de type `Foret` contient les champs `nb_arbres`, `nb_noeuds` et `table`.

La forêt *F_ex* de l'exemple introductif peut être définie par :

```
let F_ex =
   let table_F_ex = make_vect MAX noeud_vide in
      table_F_ex.(0) <- {lettre = `g`; poids = 10; fg = -1; fd = 6};
      table_F_ex.(1) <- {lettre = `a`; poids = 20; fg = 4; fd = 3};
      table_F_ex.(2) <- {lettre = `b`; poids = 13; fg = -1; fd = -1};
      table_F_ex.(3) <- {lettre = `e`; poids = 9; fg = 5; fd = -1};
      table_F_ex.(4) <- {lettre = `f`; poids = 15; fg = -1; fd = -1};
      table_F_ex.(5) <- {lettre = `c`; poids = 12; fg = -1; fd = -1};
      table_F_ex.(6) <- {lettre = `d`; poids = 8; fg = -1; fd = -1};
      {nb_arbres = 3; nb_noeuds = 7; table = table_F_ex};;
```

Un champ d'un enregistrement peut être ou non mutable ; si un champ est mutable, sa valeur peut être modifiée ; pour qu'un champ d'un enregistrement soit mutable, il faut l'indiquer au moment de la déclaration du type enregistrement ; c'est ce qui est fait ici pour les champs `nb_arbres` et `nb_noeuds` d'un enregistrement de type `Foret`. On peut accéder à la valeur d'un champ quelconque de type enregistrement en faisant suivre le nom de cette valeur d'un point puis du nom du champ considéré ; par exemple, pour la forêt définie ci-dessus, `F_ex.nb_arbres` vaut 3 et `F_ex.table.(0).poids` vaut 10. ATTENTION : la modification d'un champ mutable se fait à l'aide du signe `<-` ; on pourra par exemple écrire : `F_ex.nb_arbres <- 2;` pour indiquer que cette forêt contient dorénavant deux H_arbres.

Fin des indications pour Caml

Pascal : Dans tout le problème, on supposera qu'on écrit les différentes fonctions ou procédures dans un fichier contenant les définitions suivantes :

```
const MAX = 100;

type Noeud = RECORD
            lettre : Char;
            poids  : Integer;
            fg     : Integer;
            fd     : Integer;
end;
```

```
type T_Noeuds = array[0..MAX - 1] of Noeud;

type Foret = RECORD
            nb_arbres : Integer;
            nb_noeuds : Integer;
            table   : T_Noeuds;
end;
```

Les types `Noeud` et `Foret` sont des types pour des enregistrements (`RECORD`). Un enregistrement contient des champs (quelquefois aussi appelés des membres) ; par exemple, une variable de type `Noeud` contient les champs `lettre`, `poids`, `fg` et `fd` ; on peut accéder à un champ d'une variable de type enregistrement en faisant suivre le nom de cette variable d'un point puis du nom du champ considéré, comme dans la définition de `F_ex` ci-dessous. Les variables de type enregistrement se manipulent comme toute autre variable : on peut définir des variables de type enregistrement, on peut affecter à une variable de type enregistrement la valeur d'une autre variable du même type, les variables de type enregistrement peuvent servir de paramètres à des fonctions ou procédures et peuvent être renvoyées par des fonctions ; en revanche, il est interdit de les comparer directement.

Ainsi, la forêt *F_ex* de l'exemple introductif est définie par :

```
F_ex.nb_arbres := 3;
F_ex.nb_noeuds := 7;
F_ex.table[0].lettre := 'g'; F_ex.table[0].poids := 10;
F_ex.table[0].fg := -1; F_ex.table[0].fd := 6;
F_ex.table[1].lettre := 'a'; F_ex.table[1].poids := 20;
F_ex.table[1].fg := 4; F_ex.table[1].fd := 3;
F_ex.table[2].lettre := 'b'; F_ex.table[2].poids := 13;
F_ex.table[2].fg := -1; F_ex.table[2].fd := -1;
F_ex.table[3].lettre := 'e'; F_ex.table[3].poids := 9;
F_ex.table[3].fg := 5; F_ex.table[3].fd := -1;
F_ex.table[4].lettre := 'f'; F_ex.table[4].poids := 15;
F_ex.table[4].fg := -1; F_ex.table[4].fd := -1;
F_ex.table[5].lettre := 'c'; F_ex.table[5].poids := 12;
F_ex.table[5].fg := -1; F_ex.table[5].fd := -1;
F_ex.table[6].lettre := 'd'; F_ex.table[6].poids := 8;
F_ex.table[6].fg := -1; F_ex.table[6].fd := -1;
```

Fin des indications pour Pascal

Première partie : fonctions de base pour l'algorithme de Huffman

❒ 11 – On considère une forêt contenant *nb_nœuds* nœuds et un entier k vérifiant $1 \leq k \leq nb\_nœuds$; il s'agit d'écrire une fonction déterminant l'indice du nœud dont le poids est le plus petit parmi les k premiers nœuds de la table de la forêt, c'est-à-dire parmi les nœuds qui se trouvent entre les indices 0 et $k - 1$ de cette table. S'il y a plusieurs nœuds de plus petit poids dans l'intervalle indiqué, la fonction renverra le plus petit indice de ceux-ci.

Caml : Écrire en Caml une fonction `indice_du_min` telle que si `F` est une valeur de type `Foret` et `k` une valeur entière strictement positive ne dépassant pas `F.nb_noeuds`, alors `indice_du_min F k` renvoie l'indice recherché.

Pascal : Écrire en Pascal une fonction `indice_du_min` telle que si `F` est une variable de type `Foret` et `k` une variable de type `Integer` strictement positive et ne dépassant pas `F.nb_noeuds`, alors `indice_du_min(F, k)` renvoie l'indice recherché.

❒ 12 – On considère une forêt F constitué d'au moins deux H_arbres. Il s'agit de faire en sorte que, dans la table de F, les deux racines de plus petits poids soient les deux dernières racines dans la partie de la table consacrée aux racines de la forêt. Autrement dit, il s'agit d'examiner les racines de F (c'est-à-dire les nœuds qui se trouvent entre les indices 0 et *nb_arbres* – 1 de la table) pour mettre, par des échanges, les deux racines de plus petits poids aux indices *nb_arbres* – 2 et *nb_arbres* – 1 ; plus précisément, on mettra la racine de plus petit poids à l'indice *nb_arbres* – 1 et la racine « d'avant-dernier » plus petit poids à l'indice *nb_arbres* – 2. Par exemple, après ce traitement, la table décrivant *F_ex* devient :

table :

Indice du nœud	0	1	2	3	4	5	6
lettre	'a'	'b'	'g'	'e'	'f'	'c'	'd'
poids	20	13	10	9	15	12	8
fg	4	–1	–1	5	–1	–1	–1
fd	3	–1	6	–1	–1	–1	–1

S'il y a des égalités entre les poids qui conduisent à plusieurs couples possibles de racines de plus petits poids, on choisira alors les deux racines de plus petits indices. On utilisera la fonction définie dans la question ❒ 11.

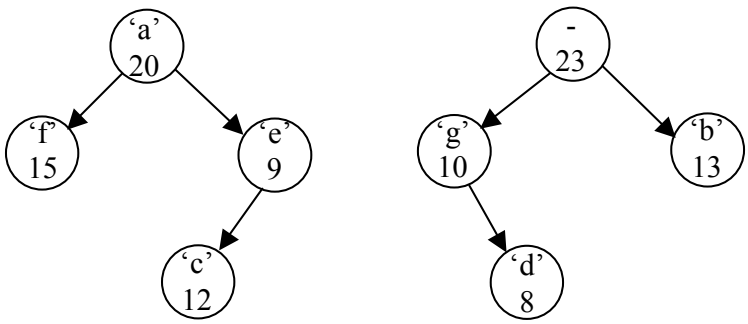
Caml : Écrire en Caml une fonction `deux_plus_petits` telle que si `F` est une valeur de type `Foret` vérifiant `F.nb_arbres` ≥ 2, alors `deux_plus_petits F` transforme le champ `table` de `F` de façon à obtenir l'ordre cherché.

Pascal : Écrire en Pascal une procédure `deux_plus_petits` telle que si `F` est une variable de type `Foret` vérifiant `F.nb_arbres` ≥ 2, alors `deux_plus_petits(F)` transforme le champ `table` de `F` de façon à obtenir l'ordre cherché.

❒ 13 – On considère une forêt F possédant au moins deux H_arbres. On définit une transformation de F nommée **assemblage** de F. Soient r_1 et r_2 deux racines de plus petits poids ; on suppose que le poids de r_1 est inférieur ou égal à celui de r_2. L'assemblage de F consiste à ajouter à F un nœud dont :

- le poids est la somme des poids des nœuds r_1 et r_2 ;
- le fils gauche est r_1 ;
- le fils droit est r_2.

Le nœud ajouté est donc la racine d'un H_arbre de la forêt obtenue par l'assemblage et le nombre total de H_arbres de la forêt a diminué de 1.
La lettre contenue par ce nouveau nœud n'a pas d'importance et n'est pas spécifiée. On ne cherche pas à ce que, après assemblage, les racines des H_arbres respectent l'ordre de la question ❒ 12.
Si on applique cette transformation à la forêt *F_ex*, celle-ci devient la forêt ci-dessous, dans laquelle on n'a pas précisé de valeur pour la lettre du nœud ajouté :



Pour cette forêt, *nb_arbres* = 2, *nb_nœuds* = 8 et la table peut être :

table :

Indice du nœud	0	1	2	3	4	5	6	7
lettre	'a'	-	'g'	'e'	'f'	'c'	'd'	'b'
poids	20	23	10	9	15	12	8	13
fg	4	2	–1	5	–1	–1	–1	–1
fd	3	7	6	–1	–1	–1	–1	–1

Caml : Écrire en Caml une fonction `assemblage` telle que, si `F` est une valeur de type `Foret` vérifiant `F.nb_arbres` ≥ 2, alors `assemblage F` transforme `F` selon les indications fournies ci-dessus. On utilisera la fonction `deux_plus_petits` définie dans la question précédente.

Pascal : Écrire en Pascal une procédure `assemblage` telle que, si `F` est une variable de type `Foret` vérifiant `F.nb_arbres` ≥ 2, alors `assemblage(F)` transforme `F` selon les indications fournies ci-dessus. On utilisera la fonction `deux_plus_petits` définie dans la question précédente.

Deuxième partie : propriétés pour l'algorithme de Huffman

Remarque : dans les illustrations de cette partie, lorsqu'un champ n'est pas précisé dans un nœud, cela signifie que sa valeur n'intervient pas.

La **hauteur** d'un nœud d'un arbre est définie de la façon suivante :

- la hauteur de la racine vaut 0 ;
- la hauteur d'un nœud autre que la racine vaut un de plus que la hauteur de son père.

La hauteur d'un nœud n est notée $h(n)$.

On dit dans un arbre qu'un nœud n est de **hauteur maximum** s'il n'existe pas de nœud de hauteur strictement plus grande que $h(n)$ dans cet arbre ; un nœud de hauteur maximum est nécessairement une feuille.

Deux feuilles d'un arbre sont dites **sœurs** si elles ont le même nœud pour père.

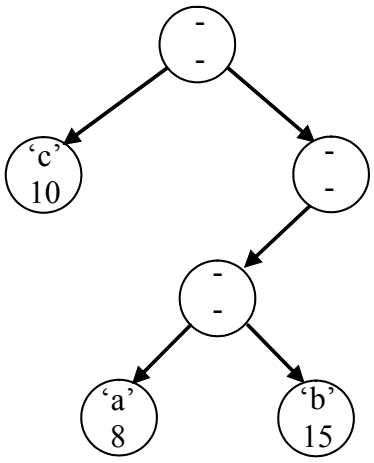
Étant donné un H_arbre A, on appelle **évaluation** de A la quantité :

$$e(A) = \sum_{f \in \{\text{feuilles de } A\}} h(f).poids(f) .$$

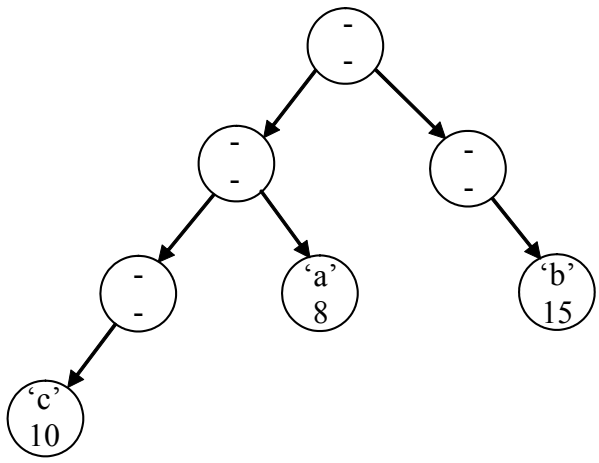
Pour le H_arbre *A_ex* ci-dessous, on a : $e(A\_ex) = 1 \times 10 + 3 \times 8 + 3 \times 15 = 79$.

On rappelle que les poids des nœuds sont des entiers strictement positifs.

On dit que deux H_arbres ont **mêmes feuilles** s'ils ont le même nombre de feuilles et que les contenus des feuilles de l'un sont les mêmes que les contenus des feuilles de l'autre. On dira par exemple que les deux H_arbres *A_ex* et *A'_ex* ci-dessous ont mêmes feuilles.



Le H_arbre *A_ex*



Le H_arbre *A'_ex*

On dit qu'un H_arbre A est **optimal** s'il n'existe pas d'autre H_arbre ayant les mêmes feuilles et ayant une évaluation strictement inférieure à celle de A.

❒ 14 – Montrer que, si un H_arbre A est optimal, alors tout nœud interne de A admet deux fils. Après avoir montré ce résultat, transformer le H_arbre *A_ex* en un H_arbre *B_ex* de mêmes feuilles que *A_ex* et d'évaluation inférieure en s'appuyant sur l'argument de la preuve ; on indiquera pour cet exemple le gain d'évaluation obtenu.

❒ 15 – Soient f_1 et f_2 deux feuilles d'un H_arbre A optimal vérifiant la relation $h(f_1) > h(f_2)$; montrer qu'alors on a $poids(f_1) \leq poids(f_2)$. Après avoir montré ce résultat, transformer le H_arbre *B_ex* obtenu à la question ❒ 14 en un H_arbre *C_ex* de mêmes feuilles et d'évaluation inférieure à celle de *B_ex* en s'appuyant sur l'argument de la preuve ; on indiquera le gain d'évaluation obtenu.

❒ 16 – On considère un H_arbre A et deux feuilles f_1 et f_2 de A de plus petits poids. Montrer qu'il existe un H_arbre optimal de mêmes feuilles que A dans lequel f_1 et f_2 sont de hauteur maximum.

❒ 17 – On considère un H_arbre A et deux feuilles f_1 et f_2 de A de plus petits poids. Montrer qu'il existe un H_arbre optimal de mêmes feuilles que A dans lequel f_1 et f_2 sont sœurs.

On considère un H_arbre A et deux feuilles f_1 et f_2 qui sont sœurs dans A ; on note p_1 et p_2 les poids respectifs de ces feuilles. On note n le père (commun) de f_1 et f_2 et on considère la transformation suivante :

- *poids*(n) devient égal à $p_1 + p_2$;
- on supprime f_1 et f_2 ; n devient donc une feuille.

Le champ *lettre*(n) n'a pas d'importance et n'est donc pas spécifié.
On note B le H_arbre ainsi obtenu.
On dit qu'on a obtenu B **en simplifiant A par coupe de f_1 et f_2**.

❒ 18 – Établir une relation entre $e(A)$, $e(B)$, p_1 et p_2.

❒ 19 – On considère maintenant un H_arbre A et deux feuilles f_1 et f_2 qui sont sœurs dans A et de plus petits poids. On simplifie A par coupe de f_1 et f_2 et on obtient ainsi un H_arbre B. Montrer que A est optimal si et seulement si B est optimal.

Troisième partie : l'algorithme de Huffman

On appelle **chaîne binaire** une suite composée de 0 et de 1.

On considère dans toute cette partie un texte T écrit avec un alphabet Σ possédant au moins deux caractères. On souhaite coder ce texte avec une chaîne binaire. Pour cela, on décide d'associer une chaîne binaire à chaque caractère de l'alphabet Σ ; la chaîne associée à un caractère est appelée **mot de code** ; l'ensemble des mots de code est le **codage de l'alphabet**. On peut alors coder le texte caractère par caractère en mettant bout à bout successivement les mots de code de tous les caractères de T ; on obtient alors le texte codé qui sera noté $C(T)$.

L'objectif est de choisir les mots de code pour que le décodage du texte soit possible et que la chaîne binaire $C(T)$ soit la plus courte possible.

❒ 20 – On considère l'alphabet Σ_ex = {'a', 'b', 'c', 'd', 'e', 'f'}.

Un codage de Σ_ex est indiqué dans le tableau ci-contre.

On suppose que le texte codé $C(T)$ est 1000011011 ; déterminer le texte T.

caractère	code
'a'	00
'b'	101
'c'	11
'd'	0101
'e'	011
'f'	100

On dit qu'un mot de code u est **préfixe** d'un autre mot de code v si la chaîne v commence par u ; par exemple, u = 01 est préfixe de v = 0110.

On dit que le codage d'un alphabet est **préfixe** si aucun mot de code n'est préfixe d'un autre mot de code.

❒ 21 – Indiquer si le codage de l'alphabet Σ_ex donné en exemple dans la question ❒ 20 est ou non préfixe. Dire pourquoi il est utile que le codage de l'alphabet soit préfixe.

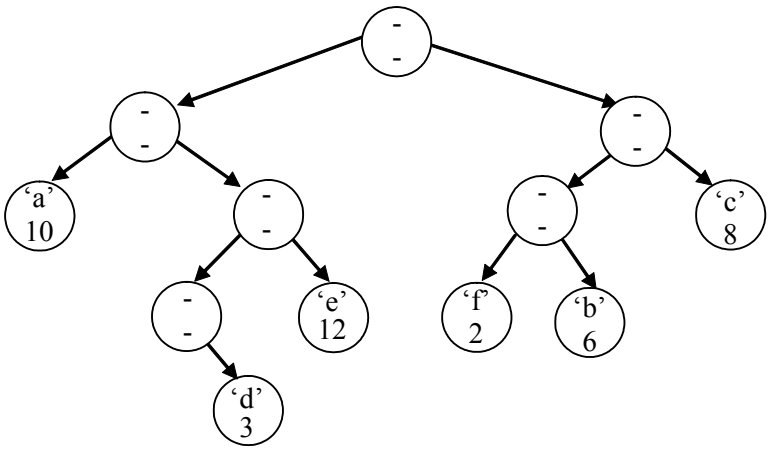
On considérera par la suite un texte T_ex écrit avec l'alphabet Σ_ex ; les nombres d'occurrences des caractères de Σ_ex dans T_ex sont donnés par le tableau ci-contre.

caractère	Nombre d'occurrences
'a'	10
'b'	6
'c'	8
'd'	3
'e'	12
'f'	2

On représente un codage préfixe de l'alphabet Σ par un H_arbre dont les feuilles contiennent les caractères de l'alphabet comme champ *lettre* et le nombre d'occurrences de ce caractère dans le texte T comme champ *poids*. Les champs *lettre* et *poids* des nœuds internes n'ont pas d'importance. Ce H_arbre est construit de sorte que le mot de code d'un caractère c corresponde au chemin de la racine du H_arbre à la feuille contenant c. Plus précisément, pour retrouver un caractère de Σ connaissant son mot de code, on part de la racine du H_arbre ; on

lit le mot de code et, au fur et à mesure, on descend dans le H_arbre ; lorsqu'on rencontre un 0, on descend à gauche, lorsqu'on rencontre un 1, on descend à droite.

Le H_arbre associé au codage de *Σ_ex* indiqué dans la question ❒ 20 est représenté ci-contre. Par exemple, pour aller de la racine au nœud contenant la lettre 'e' de mot de code 011, on descend une fois à gauche puis deux fois à droite.



❒ 22 – Indiquer une relation entre l'évaluation du H_arbre représentant un codage préfixe de l'alphabet et la longueur du texte codé $C(T)$ si T est codé en utilisant ce codage préfixe.

On appelle **codage optimal** de Σ pour T un codage préfixe de Σ minimisant la longueur de $C(T)$. Pour chercher un codage optimal, on cherche un H_arbre d'évaluation minimum et dont les feuilles correspondent aux caractères de l'alphabet Σ munis de leurs nombres d'occurrences. Pour cela, on utilise l'**algorithme de Huffman**. Cet algorithme est initialisé avec une forêt F de H_arbres tous réduits à un nœud et contenant chacun, comme *lettre*, un caractère de l'alphabet Σ et, comme *poids*, le nombre d'occurrences dans T de ce caractère. Tous les caractères de Σ figurent une fois et une seule parmi ces nœuds. Le nombre de nœuds de cette forêt initiale F est donc égal au nombre de H_arbres et encore égal au nombre de caractères dans Σ. On applique ensuite à F la boucle suivante :

tant que F possède au moins deux H_arbres, faire assemblage(F)

ce qui termine l'algorithme de Huffman.

❒ 23 – Prouver que, lorsque l'algorithme décrit ci-dessus est terminé, l'unique H_arbre de la forêt correspond à un codage optimal de Σ pour T.

❒ 24 – Les nombres d'occurrences des caractères de l'alphabet *Σ_ex* dans le texte *T_ex* étant ceux donnés entre les questions ❒ 21 et ❒ 22, déterminer, en utilisant l'algorithme de Huffman, un codage optimal de *Σ_ex* pour *T_ex*. On dessinera le H_arbre obtenu par l'algorithme et on exploitera ce H_arbre pour donner explicitement les mots de code des caractères de *Σ_ex*.

FIN DU PROBLÈME D'ALGORITHMIQUE ET PROGRAMMATION