Auteurs: Spontex

Projet Algorithmique Programmation:

Le Codage De Huffman

1.L arbre de Huffman

Les données à encoder sont sous la forme d'une chaîne de caractères. Le principe du codage de Huffman est d' exploiter la fréquence de chacun des caractères, pour les représenter par quelque chose de plus adapté que leur code ASCII, qui lui est fixe. On associe ainsi un code plus court aux caractères les plus fréquemment rencontrés.

Exemple:
si on associe a chaque lettre un code binaire:
(a :0) ; (b :11) ; (c :1001) ; (d : 1000) ; (r :101).
nombre de caractères : 10

le mot: «abracadabradabradabra»
nombre de caractères : 21

devient « 01110101-00101000-01110101-00001110-10100001-11010 »
un paquet de 8 bits fait 1 caractères ainsi
nombre de caractères : 6

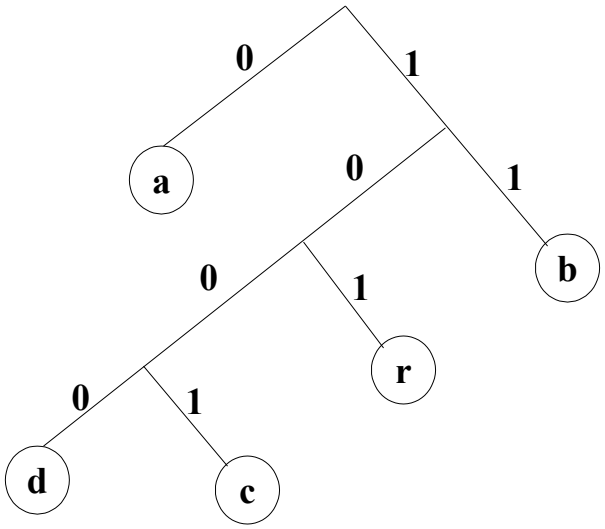
Si on inclus le Codage du fichier: on a ainsi 6+10=16 caractères < 21 .

On a donc ainsi gagner de la place après le codage.

Malheureusement pour le décodage, si le code est mal choisi, exemple (a :1) le code «11» n indique pas s'il faut le transformer en «a a» ou «b» . Il faut donc absolument que notre codage soit «préfixe», c'est à dire qu'il ne faut pas qu'aucun code ne soit préfixe l'un de l'autre.

Un tel codage peut se représenter par un arbre binaire dont les feuilles sont les caractères, et ou les arcs vers les fils sont étiquetés par des 0 en destination d'un fils gauche, et par un 1 en destination d'un fils droit. Le code d' un caractère est alors donné par des étiquettes des arcs de la racine jusqu'à la feuille étiquetée par ce caractères.

Exemple:



Le choix d'un tel arbre nous assure que le code transcrit grâce a lui est bien préfixe.

En effet, le chemin pour aller de la racine jusqu'à une feuille dans un arbre binaire est unique, on ne peut que faire des choix qui sont définitif, soit aller a gauche (rajouter un 0), soit aller a droite (rajouter un 1) , sans possibilité de retour en arrière.

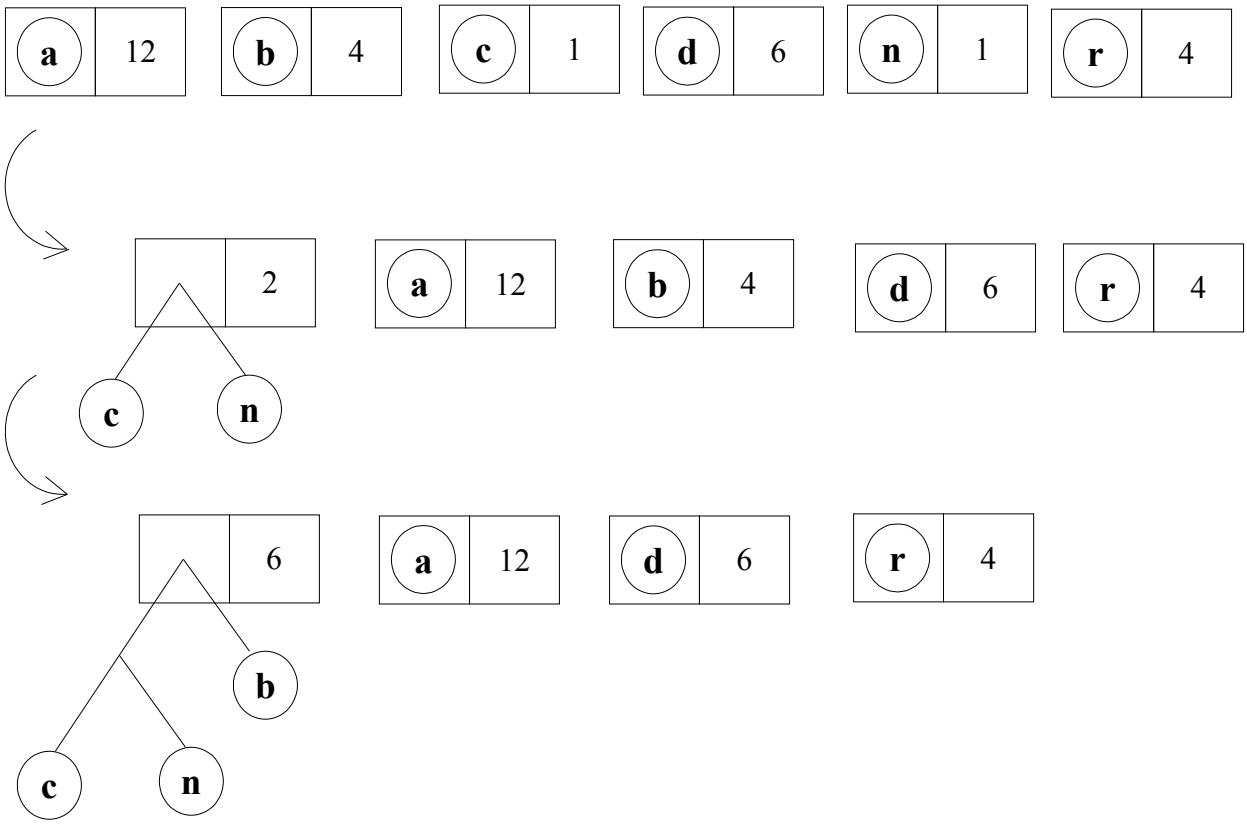
Comme dit précédemment l arbre de Huffman dépend seulement des fréquence des caractères. Il faut donc les calculer, voir les triés.

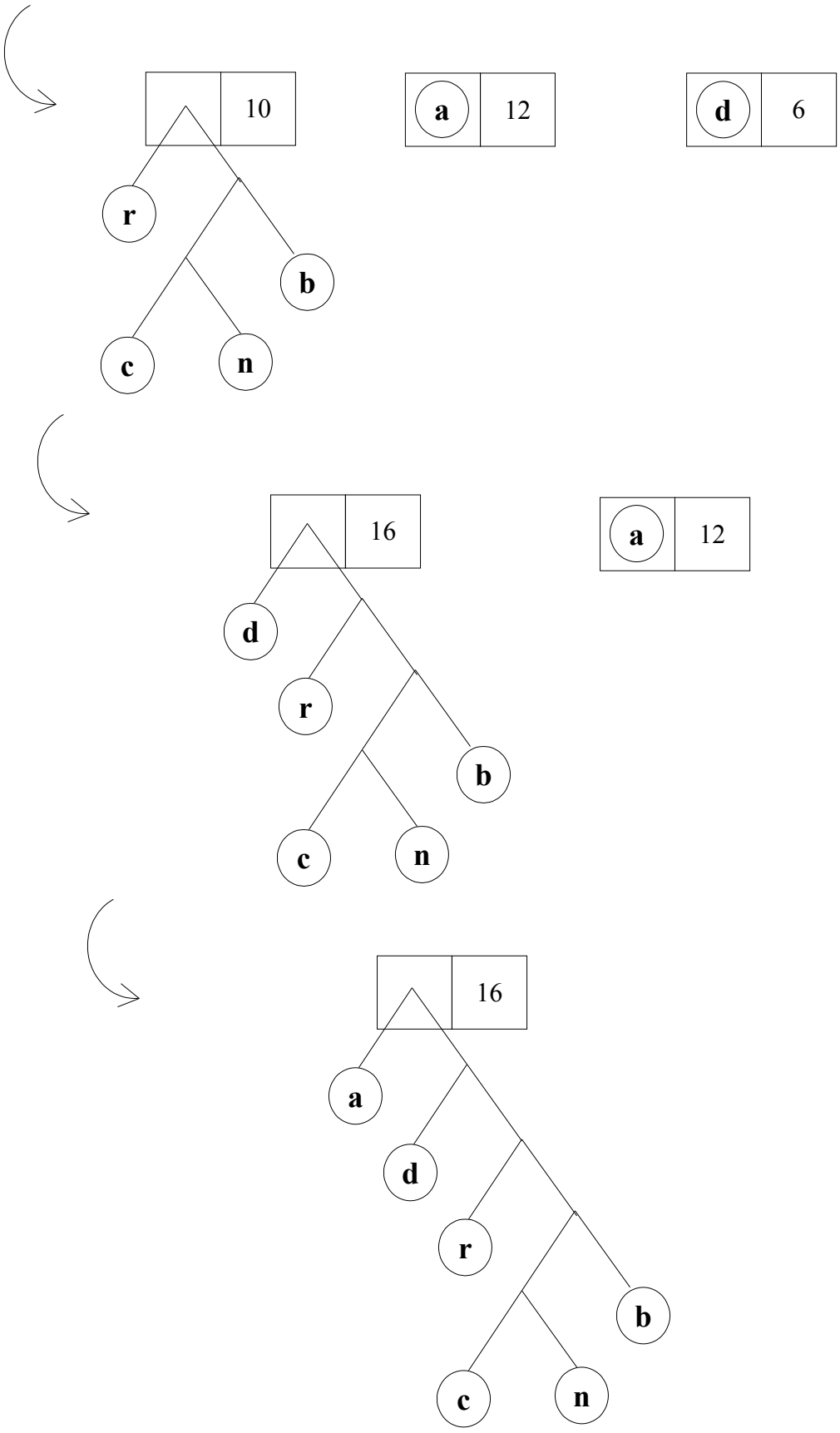
L algorithme de Huffman est le suivant: on considère un ensemble d'arbres auxquels on associe un entier. Au départ cet ensemble est constitué de feuilles. Chacune correspond aux caractères présents dans le texte, et sont associés à leur fréquence. Tant qu'il y a au moins deux éléments dans cet ensemble, on extrait les deux qui ont les plus petites fréquences, on construit alors un nouvel arbre dont les fils sont les éléments extraits, on étiquette cet arbre par la somme des étiquettes de ses fils, et on le place dans notre ensemble, et ainsi de suite.

Exemple:

le mot: «abracadabradabradabradadadan»

caractères	fréquences
a	12
b	4
c	1
d	6
n	1
r	4





On souhaite de plus que cet arbre soit unique, or il reste des *ambiguïtés*.
On les enlèves:

- le fils gauche est celui d'étiquette la plus faible.
- En cas d'égalité, on se réfère au plus petit caractère (pour l'ordre lexicographique) présent dans les feuilles du sous arbre.

2. Algorithme d'Huffman avec le tri par fusion

Plusieurs structures de données peuvent être utilisées pour le calcul du codage.
Le Vector et le trie par fusion pour le classer du plus petit au plus grand a été notre première idée.
En effet, nous cherchons à retirer les 2 éléments avec le plus petit poids.

a) L'Algorithme

```
Huffman(liste l)
        TRI_FUSION(l)
        tant que taille[l]>1 faire
                fus ← FUSION(RETIRER_PREMIER(l),RETIRER_PREMIER(l))
                INSERER(fus)  //insérer l'élément a la bonne place dans la liste
        fin tant que
        retourner RETIRER_PREMIER(l)
```

b) Etude de la Complexité

Selon le cour, le tri par fusion a une complexité $O(n \log n)$
et la boucle tant que a une complexité O(n^2)
car insérer au pire des cas parcours toute la liste a chaque itération
(soit : n-2 +n-3+....+1= (n-1)(n-1)/2
donc la complexité au pire des cas de cet algorithme est

O(n^2)

c) Pourquoi utilisé le tas

La structure tas permet de gagner beaucoup en complexité car elle permet de classer beaucoup plus rapidement en effet il n'est pas utile de classer toute la liste au début car seul le terme le plus petit nous intéresse .
La structure tas permet de retrouver facilement et le plus rapidement ce plus petit terme.
Nous verrons dans la question 5 que la complexité de l'algorithme d'Huffman avec les tas est plus petite qu'avec un tri par fusion.

3. Algorithme d'Huffman dans notre projet

a) L'Algorithme

Détail de l'algorithme utilisé dans le projet

Fonction pour calculer l'arbre de Huffman

```
Huffman( liste tas)
        tant que taille[tas]>1 faire
                fusion ← fusion(extraire_min(tas),extraire_min(tas));
                insérer_tas(fusion);
        fin tant que
        retourner extraire_min(tas);
```

Fonction pour extraire le minimum du tas c'est à dire le haut du tas

```
extraire_min(liste tas)
        si taille[tas] < 1
                alors erreur «débordement négatif»
        max ← tas (1)
        tas(1)← tas(taille[tas])
        taille[tas] ← taille[tas]-1
        entasser(tas,1)
        retourner max;
```

Fonction entasser utilisée dans la fonction insérer_tas

```
entasser(liste tas,nombre i)
l ← gauche(i)
r ← droit(i)
si tas ≤ taille[tas] et tas[l]>tas[i]
        alors max ← l
        sinon max ← i
si r ≤ taille[tas] et tas[r]>tas[max]
        alors max ← r
si max ≠ i
        alors échanger tas[i] ↔ tas[max]
                entasser(tas,max)
```

Fonction qui insère un élément a la fin du tas et restasse le tas

```
insérer_tas(élément el)
        taille[tas] ← taille[tas]+1
        i ← taille[tas]
        tant que i>1 et tas[père(i)]<el
                faire   tas[i] ← tas[père(i)]
                        i ← père(i)
        tas[i] ← el
```

Retourne le Père de l'élément

```
père(nombre i)
        retourner i/2
```

Retourne le fils gauche de l'élément

```
gauche(nombre i)
        retourner 2i
```

Retourne le fils droit de l'élément

```
droit(nombre i)
        retourner 2i+1
```

b) Etude de la Complexité

D'après le cour, la fonction entasser a une complexité O(lg n)
La fonction extraire_min a une complexité O(lg n) car cette fonction n'effectue qu'une quantité de travail constant en plus du temps en O(lg n) nécessaire pour entasser.
La fonction insérer_tas a une complexité O(lg n) pour un tas à n éléments, puisque le chemin suivi depuis la nouvelle feuille jusqu'à la racine a la longueur O(lg n).
la fonction tant que de Huffman fait au plus n boucle donc la complexité total de cet algorithme est

O(n lg n)

c) La Terminaison

A chaque boucle tant que on retire 2 éléments et on en remet un donc le nombre d'éléments dans que tas diminue a chaque boucle de 1, donc quand il ne reste plus qu'un élément dans le tas cet élément et l'arbre final d'Huffman. A ce moment la boucle tant que s'arrête et la fonction retourne le dernier élément.

4.Codage

```
Algo Table_Codage{
 Tableau de Chaine de Caractères Table [0..255];
 Arbre Huffman
 si  Huffman est une feuille
       Table[Huffman.valeur]<-"0"
        retourne la Table
 fin si
 Creer_Table_Codage("",Table,Huffman);
 retourne  Table;
```

```
Algo Creer_Table_Codage(Chaine de caractere Codage, Tableau de Chaine de Caractères Table [0..255], Arbre Huffma,)
      si Huffman est une feuille
       Table[Huffman.valeur]<-Codage de valeur
      sinon Creer_Table_Codage (Codage+"0",Table,Huffman.gauche )
            Creer_Table_Codage(Codage+"0",Table,Huffman.droit )
     fin si
```

complexité du parcours de l arbre en préfixe: **O(n)**

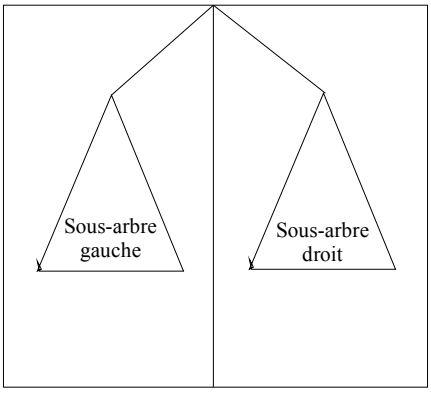
5.Représentation graphique de l'arbre

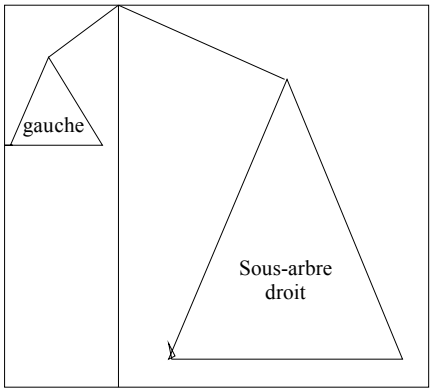
Complexité: O(n) avec n le nombre de noeuds et feuilles de l arbre.

```
public void Affiche (int x1,int x2,int y,Graphics g){
      int tmp;
            if (this.Estfeuille()){
                                    g.setColor(Color.white);
                                    g.fillOval((2*x1+x2)/2-5,y-5,20,20);
                                    g.setColor(Color.black);
                                    g.drawOval((2*x1+x2)/2-5,y-5,20,20);
                                    g.drawString(""+this.valeur,((2*x1+x2)/2)+2,y+7);
                                   //arret de la recursivité et affichage de la feuille

            }
            else {              tmp=(9*x2/10)*(fGauche.hauteur+1)/
(fGauche.hauteur+this.fDroit.hauteur+2);
                                g.drawLine((2*x1+x2)/2,y,(2*x1+tmp)/2,y+75);
                                  this.fGauche.Affiche(x1,tmp,y+75,g);       //affiche la partie gauche
                                g.drawLine((2*x1+x2)/2,y,(2*x1+x2+tmp)/2,y+75);
                                this.fDroit.Affiche(x1+tmp,x2-tmp,y+75,g);      //affiche la partie droite
                                }}
```

Pour l'affichage de l arbre on utilise un algorithme récursif, a chaque passage dans la boucle si le sous arbre en cours n est pas une feuille on découpe la fenêtre en deux partie, ainsi aucun recouvrement n'est possible. On attribue au sous-arbre qui contient le moins d'éléments une plus petite partie proportionnelle au nombre d'éléments de l'arbre.





6.Etude des Taux de compression

Nous avons vu qu'il fallait également joindre la table de codage au texte compressé. intéressons nous a sa taille.

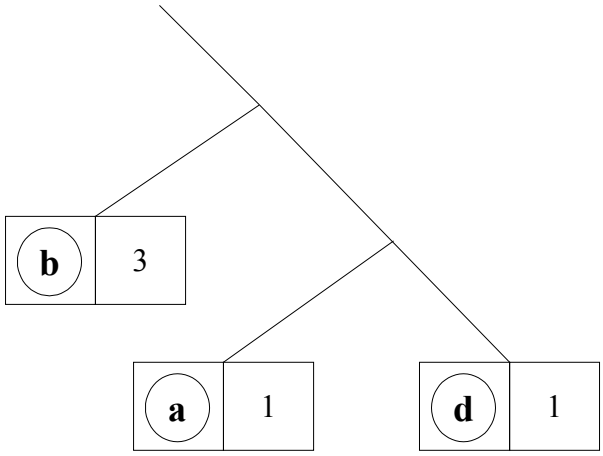
Pour un texte écrit **sans ponctuations**, et **en majuscule :**

-l'arbre le plus petit possible est l arbre qui ne contient qu'une feuille, soit un seul caractères.
Taux de compression: Très Elevé
Valeur: 90% de taux de compression

-l'arbre le plus grand possible est l arbre qui contient tout les caractères possible, soit dans le cas qui nous intéresse 26 caractères.
Taux de compression: Aléatoire
Valeur: de 0 à 90% de taux de compression

-**l'arbre le plus profond** est l'arbre le plus grand possible est tel que la fréquence d'une lettre soit supérieure a la fréquence des deux lettres de poids inférieure précédente.
Taux de compression: Elevé
Valeur: 75% de taux de compression

exemple:



n.b: pour les taux de compression on ne compte pas l'implémentation de l arbre.

-**l'arbre le plus complet**, c'est à dire que toute les feuilles sont a la même profondeur, est l'arbre qui ne contient des lettres que de même fréquences.
Taux de compression: Bon et Constant
Valeur: environ 80% de taux de compression

7.Codage de l arbre

L'arbre construit, il faut essayer de trouver la meilleure façon de l'intégrer au fichier pour qu'il prenne le moins de place possible.

1. Méthode naïve

Comme dans l exemple donné au début

Exemple:
si on associe a chaque lettre un code binaire:
(a :0) ; (b :11) ; (c :1001) ; (d : 1000) ; (r : 101)

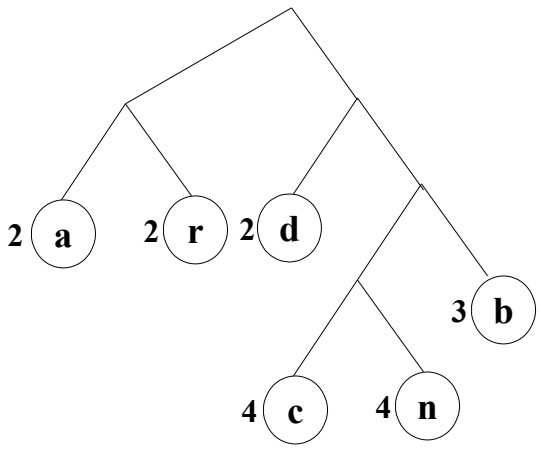
On associe a chaque caractère son codage.
Ou on peut encore associé a chaque caractère sa fréquence.

Malheureusement, si on associe a chaque caractère sa fréquence donc un int (codé sur 4 octet), on se retrouve avec un arbre codé qui est 5 fois plus grand que tout les caractères à codé réunis. De même si on associe le codage de chaque lettre, on est limité par la taille maximale du plus grand code, et a partir d un code de taille supérieure à 8 on peut perdre la même place voir plus que pour l association des fréquence. Ce codage n est pas constant.

2. Méthode de la profondeur

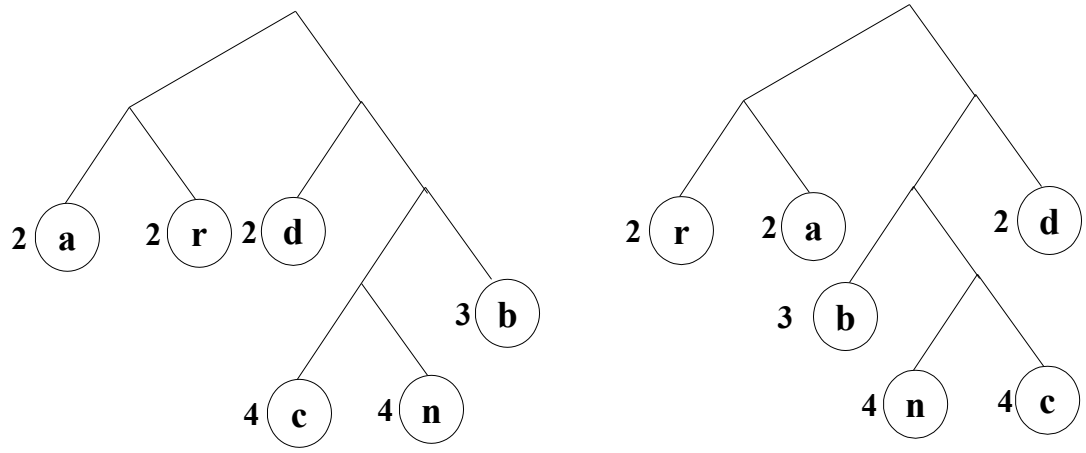
Une bonne méthode est de par exemple associé aux feuilles la taille de leur profondeur. Pour des caractères contenu dans le code ASCII standard, la profondeur ne peut excéder 255, ce qui correspond a la taille d un octet.

Exemple:



Codage de l'arbre: **a 2 r 2 d 2 b 3 c 4 n 4**

Récupération de l'arbre possibles:

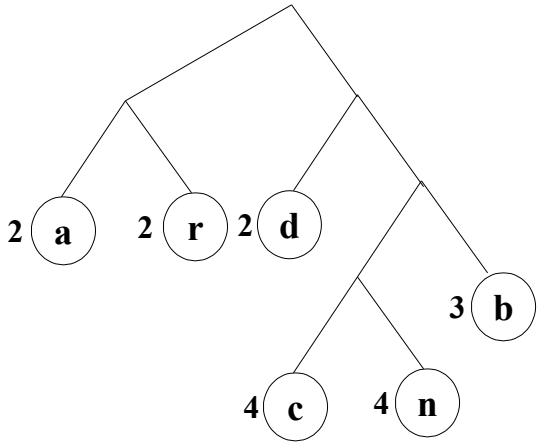


On peut donc retrouvé le même arbre, a la rotation miroir près. La taille du codage des lettres par contre elle ne bouge pas. Les codages obtenues par deux arbres qui se distinguent seulement par la résolution des ambiguïtés sur le choix des minimaux est donc en terme de taux de compression totalement équivalent.

Pour lever l ambiguïté, on peut utiliser un ordre préfixé pour parcourir l'arbre, on récupérera l'arbre dans cette ordre, aucune ambiguïté ne pourras être ainsi soulevée.

Sur notre exemple le codage de l'arbre devient
Codage de l'arbre: **a 2 r 2 d 2 c 4 n 4 b 3**

ce codage ne peut donner ensuite que l'arbre suivant:

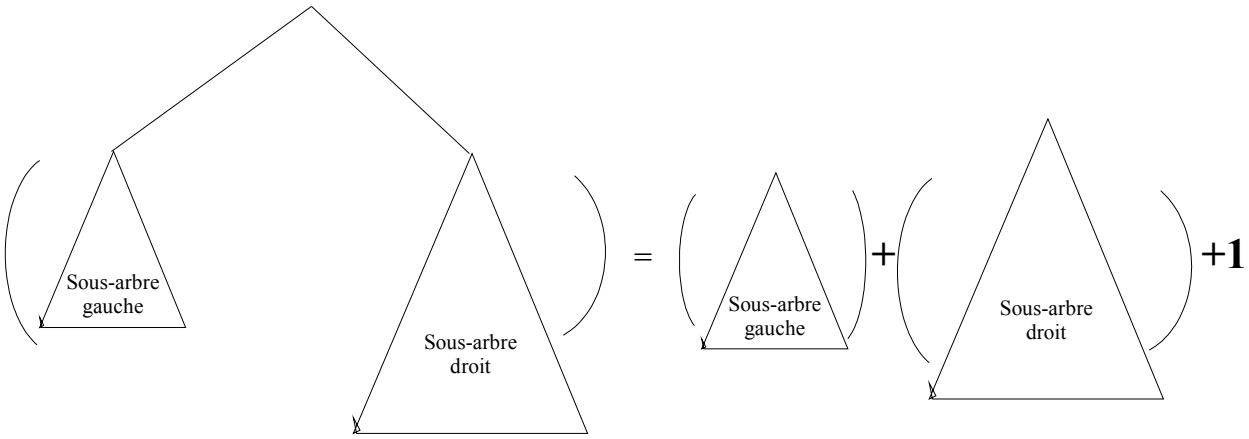


On a donc ainsi pu gagner de la place lors du codage de l arbre, on se retrouve avec un arbre codé qui est seulement 2 fois plus grand que tout les caractères à codé réunis.

La méthode Lukasiewicz

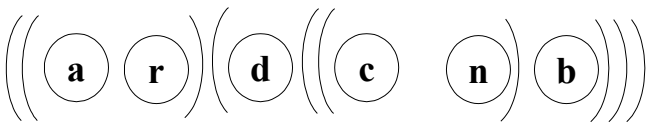
```
public void WriteArbre(Arbre Huffman){ // fonction qui écrit le codage de l arbre
String Codage=Huffman.Codage();        //on stocke le codage de l arbre
for(int i=0;i<Codage.length();i++){    //on écrit le codage de l arbre
            if(Codage.charAt(i)=='0'){this.WriteBit(0);this.WriteByte(Codage.charA(++i));}
 //apres un 0, on ecrit l octet de la feuille correspondante
            else {this.WriteBit(1);}
}
this.WriteBit(1);   //bit indiquant la fin de l'arbre
}
```

Cependant même si la méthode de la profondeur reste une bonne méthode, la méthode de Lukasiewicz est encore meilleur. C'est une méthode récursives, qui procède comme suit, lorsque l on rencontre un noeud on rajoute **1** à la liste a droite, et on traite les sous-arbres, lorsque l on rencontre une feuille on note son étiquette précédé d'un **0.**



Codage de l'arbre: 0a0r10d0c0n10b11

ce codage ne peut donner ensuite que l'arbre suivant:



L' intérêt de ce codage, c est qu'il est codé en binaire les 0 représentent des feuilles et les 1 des noeuds. Il ne prend donc qu'un bit lors du codage d'un arbre pour une feuille ou un noeud donné. La taille du codage devient donc environ n+(2*n/8) avec n représentant le nombre de caractères 2*n représente le nombre de **0** pour les feuilles + le nombre de **1** pour les noeuds.

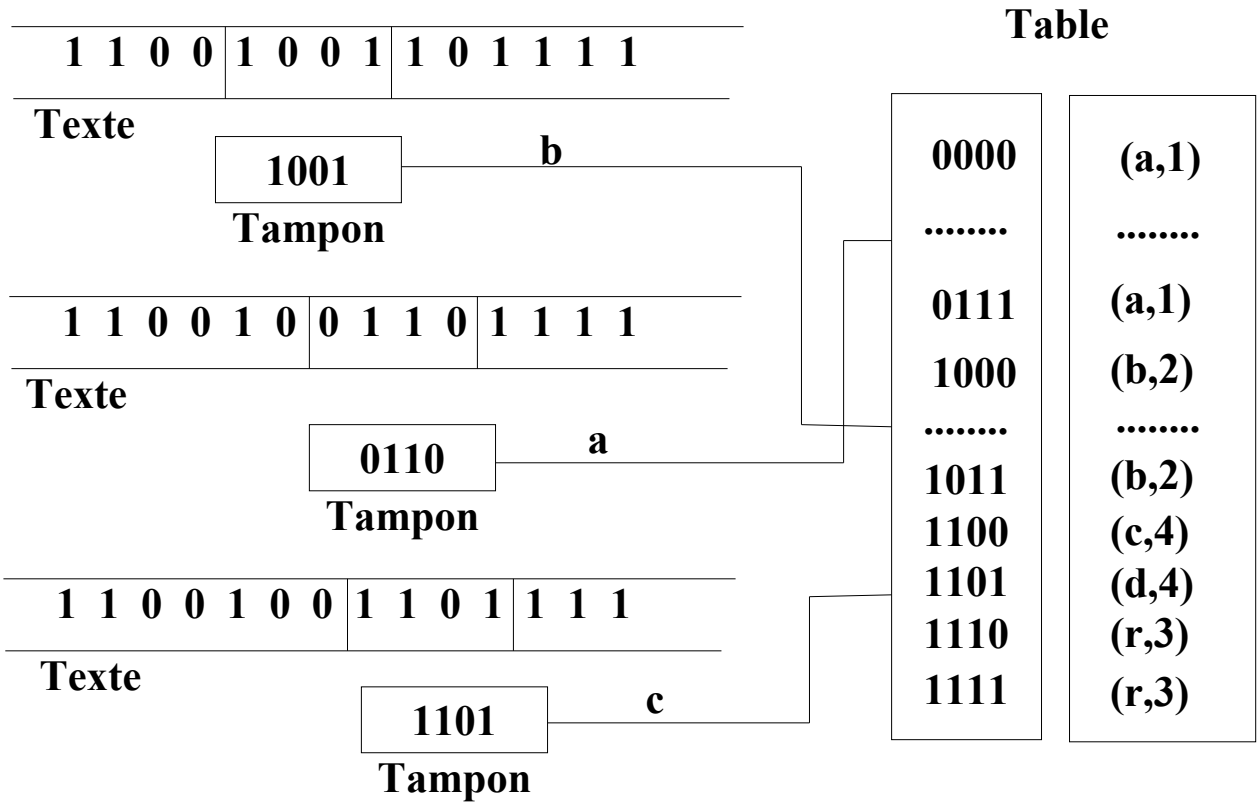
8.Le Décodage

La méthode simple pour le décodage. Après la reconstruction de l'arbre, on parcours l'arbre en allant a gauche si on rencontre un **0 ,** et a droite si on rencontre un **1**, jusqu à rencontrer une feuille, dont on relève l'étiquette. Puis on reprend le parcours de l arbre par la racine, jusqu'à la fin du fichier.

Il existe cependant une autre méthode, on utilise pour cela un table de décodage. On complète le codage de chaque lettre par des zéro pour qu'ils aient tous la même longueur L. On utilise ce mot complété comme entrée dans la table de décodage. Lorsque l'on rentre un mot, on y indique donc sa valeur et la taille réel de son codage. Puis on dispose d'un tampon de longueur L. Son contenu détermine l'entrée de la table, On décode le symbole, et on lit les lettres jusqu'à remplir le tampon.

```
private void Tamponne(){
this.Lecteur=Reader.ReadBit();
this.Tampon+=Fonction.BitToString(Lecteur);}
/**
* Lit le fichier jusqu a la fin et decode les elements au fur a mesure
* @param taille_tableau logarithme de la taille du tableau de decodage
* @throws ErrorFichierCodeIncorrect renvoie l exception si le fichier donné n est pas un fichier codé
*/
private void LectureDuFichier(int taille_tableau)throws ErrorFichierCodeIncorrect{
while(Lecteur!=-1){ Tamponne();
    if(Tampon.length()==taille_tableau){this.WriterDecode(Fonction.StringToInt(Tampon));}
    if(Reader.Available()==0){ //quand on arrive en fin de fichier
        while(Lecteur!=-1){  //on rempli le tampon jusqu au bout
        Tamponne();
        }
        if(FichierInValide()){throw new ErrorFichierCodeIncorrect();} //on peut tester ainsi si le fichier deonné est
valide
        if(Tampon.length()!=0){Tampon=Tampon.substring(0,Tampon.length()-this.Bit_ajouter-1);} //et on enleve
les bit ajouter en fin de fichier au tampon}}}
/**
* Traite le tampon final
* @param taille_tableau logarithme de la taille du tableau de decodage
*/
private void TraitementFinFichier(int taille_tableau){
while(Tampon.length()!=0){
if(Tampon.length()>=taille_tableau){this.WriterDecode(Fonction.StringToInt(Tampon.substring
(0,taille_tableau)));}
else{this.WriterDecode(Fonction.StringToInt(Tampon)<<(taille_tableau-Tampon.length()));}}}
```

Complexité: O(n) avec n le nombre de caractères du fichierAvantage de la table sur l'arbre: on réalise un taux de décodage constant.



<u>Exemple</u> abracadabra

Table des Matières

1.Arbre de Huffman
2.Algorithme d'Huffman avec le tri par fusion
3.Algorithme d'Huffman dans notre projet
4.Codage
5.Représentation graphique de l'arbre
6.Etude des Taux de compression
7.Codage de l arbre
1. Méthode naïve
2.
8.Le Décodage

Programmation:

Classes principales:

1. Class Affichabre

2. Class Arbre

3. Class Arbre_poids

4. Class BitEcriture

5. Class BitLecture

6. Class Codage

7. Class Decodage

8. Class Ecriture

9. Class ErrorFichierCodeIncorrect

10. Class ErrorFichierVide

11. Class Fonction

12. Class Huffman

14. Class Lecture

15. Class TailleCodageSuperieure

16. Class Tas

Ligne de commande pour lancer le projet:
>java **Huffman** [FichierSource][FichierCodageDestination][FichierDecodageDestination][options]
-a: affichage d'informations relative au codage et au décodage
-f: forcer le codage
-c: utilisation du projet en fonction codage uniquement
-d: utilisation du projet en fonction décodage uniquement
-g: utilisation de l interface graphique seul -f et -a sont compatible avec