Semestre 1/Master 1 2012-2013

Projet de MI015 ALGAV(Algorithmique Avancée)

HUFFMAN DINAMIQUE

Cours : Michèle Soria

TD-TME : Antoine Genitrini, Maryse Pelletier, Philippe Trébuchet, Binh-Minh Bui Xuan

Par : SUN Xin 3103249

Département informatique spécialité SAR

Sommaire

1. Introduction
2. Principal
3. Structure de donnée
4. Fonctions utilisées
5. Codes
6. Conclusion
7. Tests

I. Introduction

Huffman statique

2 fois de lecture de fichier :

1er fois => compter la frequence de chaque caractere.

construire l'arbre Huffman.

2em fois => compression

chaque fois prendre les 2 éléments ont le plus petits poids

La caractere a plus de frequence -> code plus court

VS

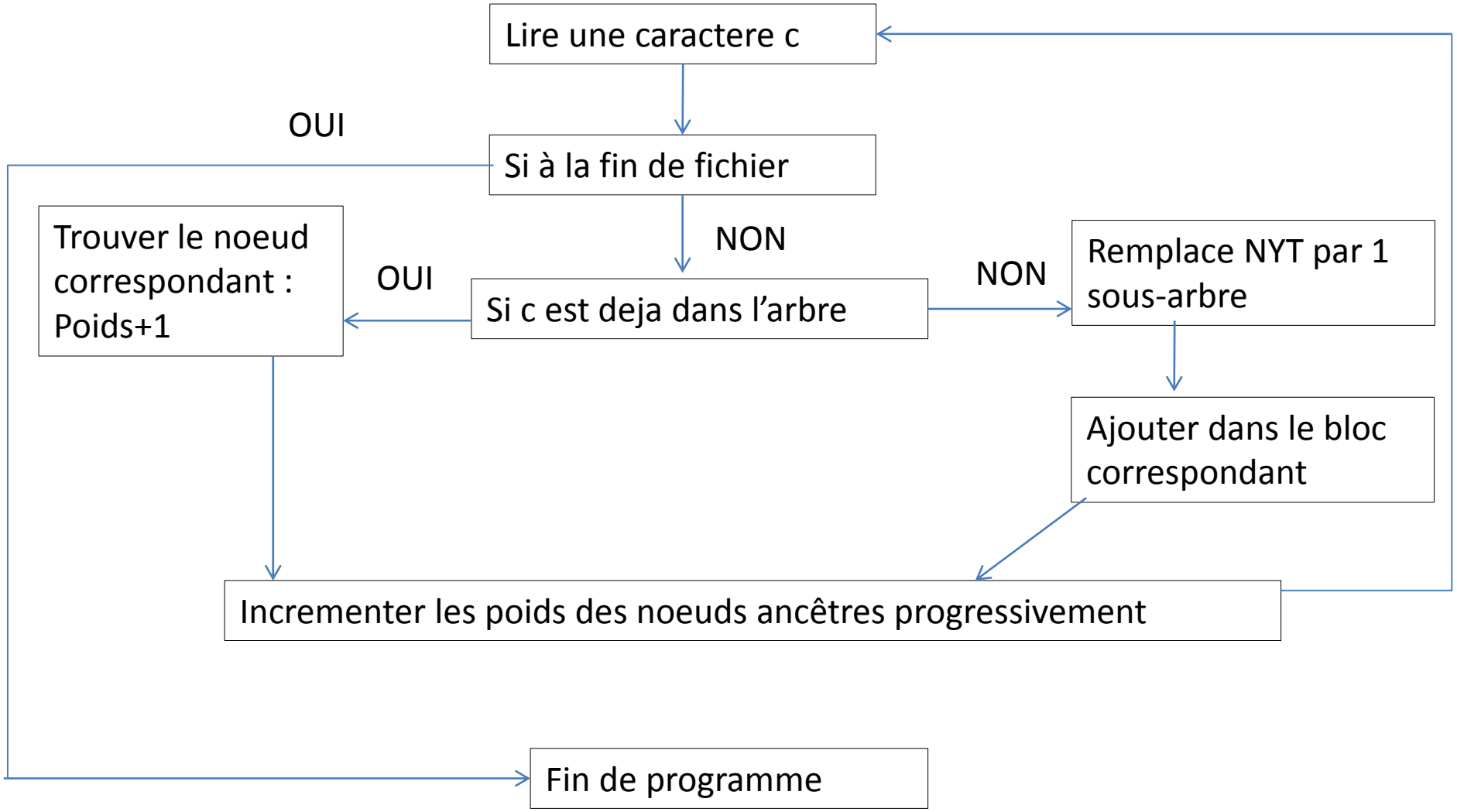
Huffman dynamique

1 seule fois de lecture du fichier

Constuire l'arbre Huffman adaptatif dynamiquement

Utilisé dans plusieurs domaines, ex: image HD

II. Principal

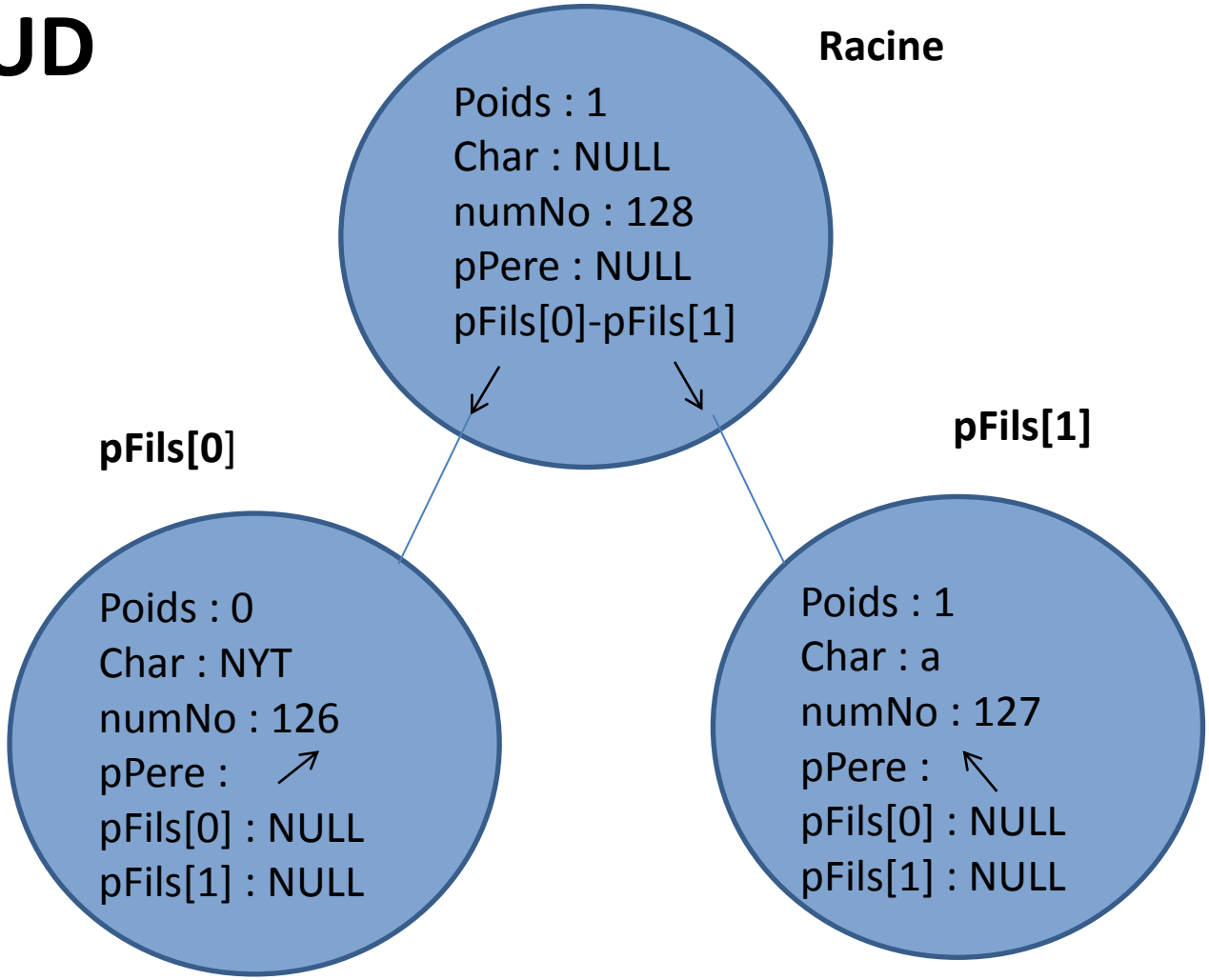


III. Structure de donnée

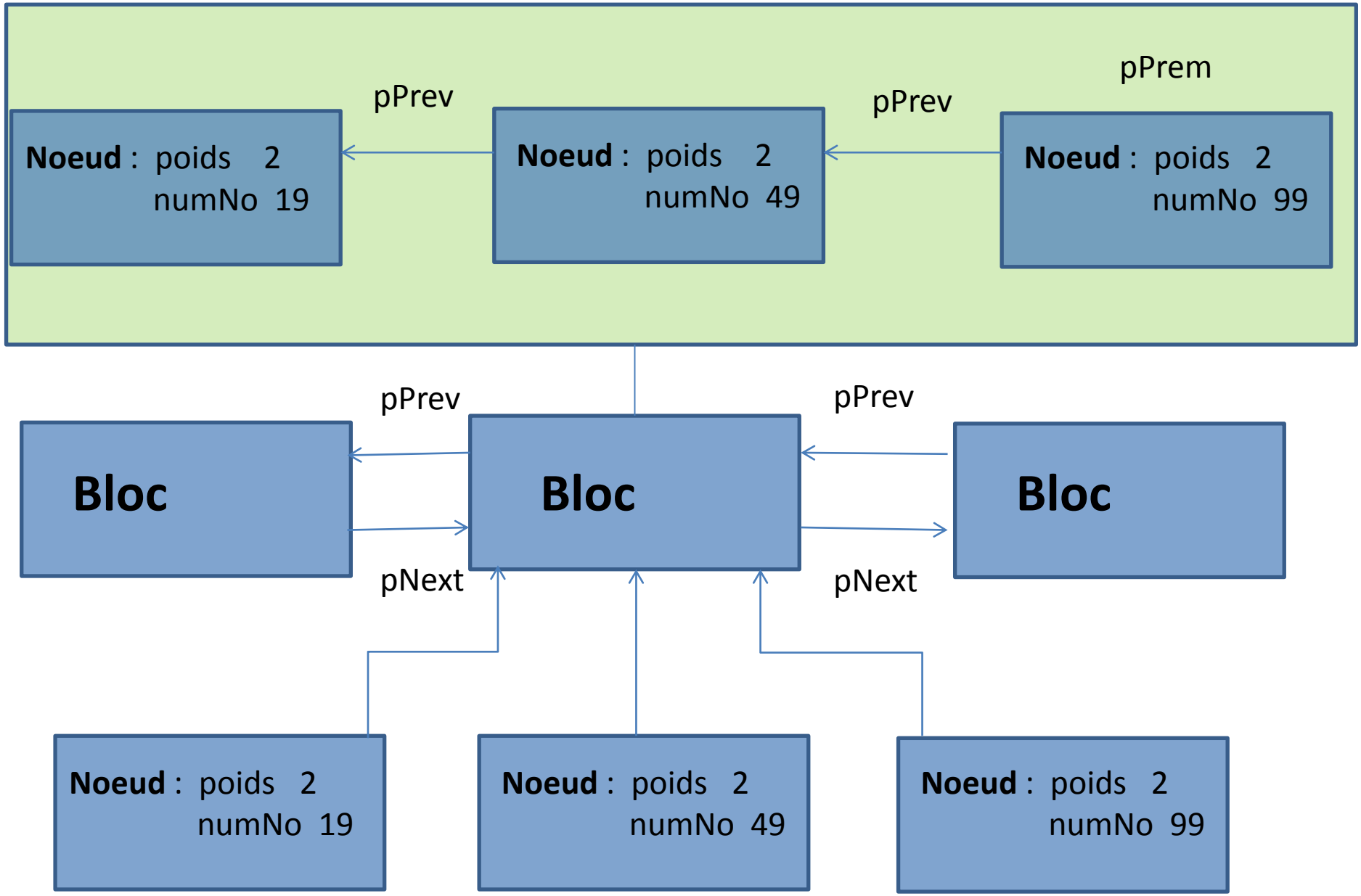
```
typedef struct NOEUD {
            int poids;
             int Char;
             int numNo;
             struct NOEUD *pPere;
             struct NOEUD *pFils[2]; /*      pointeur vers 2 fils noeuds :
                                                    pFils[0] => fils gauche
                                                    pFils[1] => fils droit */
             struct BLOC *pbloc;
             struct NOEUD *pPrev;   }NOEUD,*PNOEUD;


typedef struct BLOC {
            PNOEUD pPrem;   /* le noeud qui a le plus grand poids dans ce bloc */
            struct BLOC *pPrev;
             struct BLOC *pNext;   }BLOC,*PBLOC;
```

NOEUD


Racine
Poids : 1
Char : NULL
numNo : 128
pPere : NULL
pFils[0]-pFils[1]
pFils[0]
pFils[1]
Poids : 0
Char : NYT
numNo : 126
pPere :
pFils[0] : NULL
pFils[1] : NULL
Poids : 1
Char : a
numNo : 127
pPere :
pFils[0] : NULL
pFils[1] : NULL

BLOC


pPrev
pPrev
pPrem
Noeud : poids 2
numNo 19
Noeud : poids 2
numNo 49
Noeud : poids 2
numNo 99
pPrev
pPrev
Bloc
Bloc
Bloc
pNext
pNext
Noeud : poids 2
numNo 19
Noeud : poids 2
numNo 49
Noeud : poids 2
numNo 99

IV. Fonctions utilisées

main()

int afficher(PNOEUD pNoeud);

int compression();

void renouvellementBloc(PNOEUD pNoeud);

void incrementer(PNOEUD pNoeud);

void renouvellementNoeud(char c);

int compression()

On utilise la boucle ’while’, chaque fois lire 128 caractères,jusqu’a la fin de fichier entrée. Puis on fait le traitement pour chaque caractère lue avec appel la fonction renouvellementNoeud()

void renouvellementNoeud(char c)

Applée par la fonction compression(). Faire le traitement de noeud.

Pour chaque caractère lue, il y a 2 cas :

a. caractère nouvelle =>

1. remplace NYT par sous-arbre
2. appelle la fonction renouvellementBloc() avec le noeud(père de nouveau NYT) comme paramètre pour ajouter les nouveaux noeuds dans le bloc correspondant
3. appelle la fonction incrementer() pour modifier les poids des noeuds ancêtres progressivement => à partir du noeud du grand père du nouvelle NYT

b. caractère déjà existé sans l’arbre =>

étape 3 de cas a) => à partir de le noeud existé contenant cette caractère

void renouvellementBloc(PNOEUD pNoeud)

Applée par la fonction renouvellementNoeud().
Ajouter les nouveaux noeuds dans le bloc correspondant.
La paramètre entrée : pNoeud => le noeud qu'on doit le mettre dans le bloc

cas 1 : on trouve le bloc (poids de bloc = poids de pNoeud)
mettre ce noeud dans la bonne position du bloc (numNo)

cas 2 : on ne trouve pas le bon bloc
=> soit n'exit pas, soitbloc vide(cas particulier)

cas 2.1 : il n'existe aucun bloc => init bloc

cas 2.2 : créer un nouveau bloc qui stocke tous les noeud
dont poids= poids de pNoeud

void incrementer(PNOEUD pNoeud)

Applée par la fonction renouvellementBloc (). La paramètre pNoeud est le noeud doit etre incrementer son poids.

1. On vérifier si le numéro de ce noeud est le plus grand dans son bloc
   1.1. si oui => échange avec le noeud a plus grand de numéro dans son bloc (pNoeud->pBloc->pPrem

2. Puis poids+1

3. Ensuite faire la même chose pour son père.

V. Codes

compression.c

Compilation : gcc –o compression compression.c

Executer : ./compression <nom_de _fichier_entree>

VI. Conclusion

Propriete de l'arbre Huffman adaptatif (sibling property) :

1. Le noeud a plus de poids => son numero de noeud est plus grand
2. Le numero de noeud de pere est toujours plus grand de ses fils

Caractere :

Chaque fois avant le lecture d'une nouvelle caractere, l'arbre est bon etat(complet).

Chaque nouvelle caractere pout provoquer la modification de l'arbre.

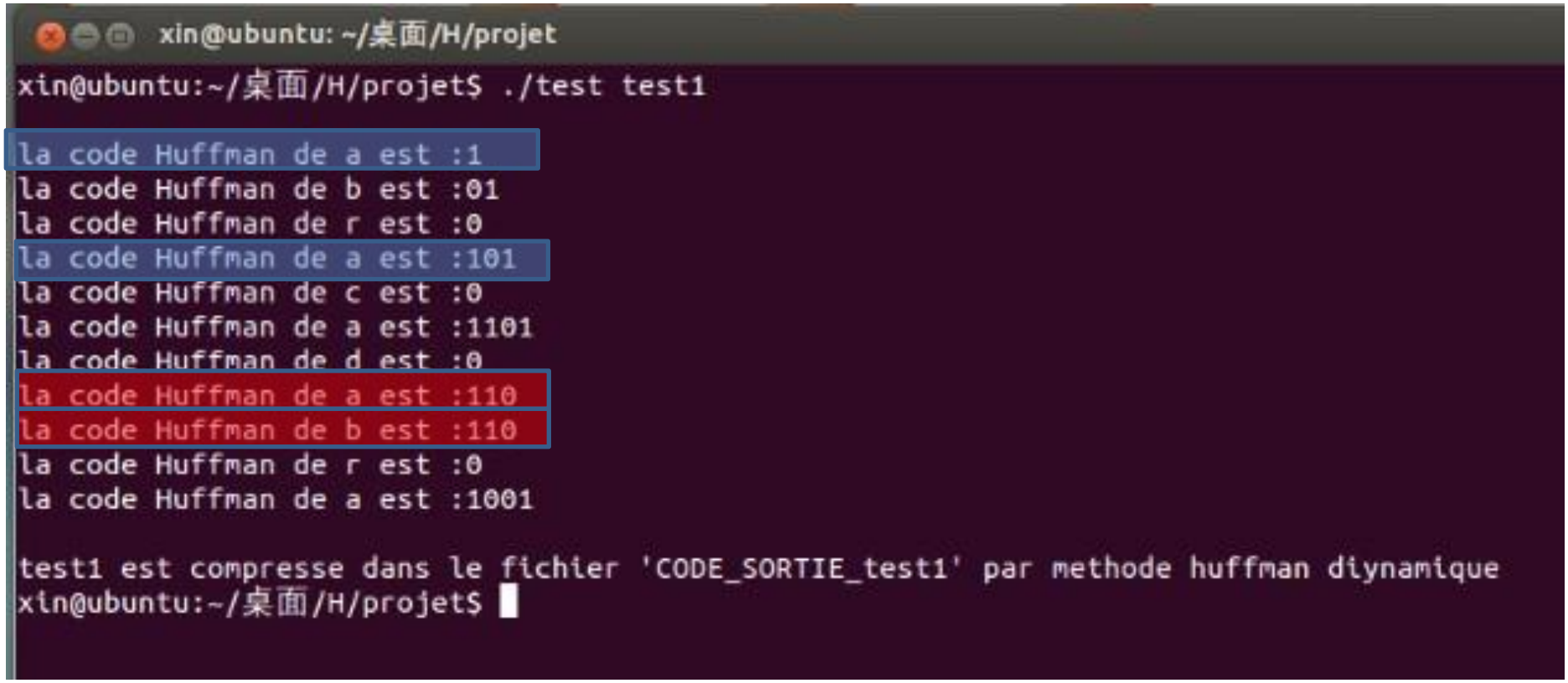
Pour la même caractère, c'est possible de sortir avec les codes différentes.

ex : caractere 'a' de test1 (en bleu)

Pour 2 différentes caractères quelconque, c'est possible de sortir avec le même code

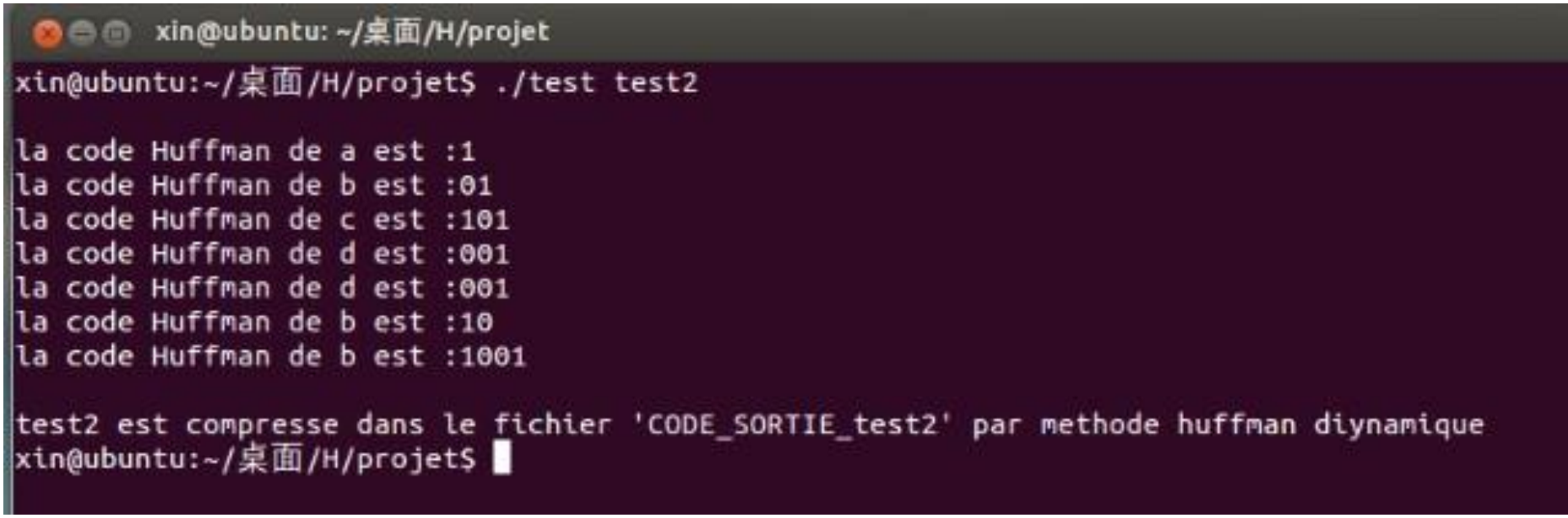
ex : caractere 'a' et 'b' de test1 (en rouge)

VII. Certains Tests



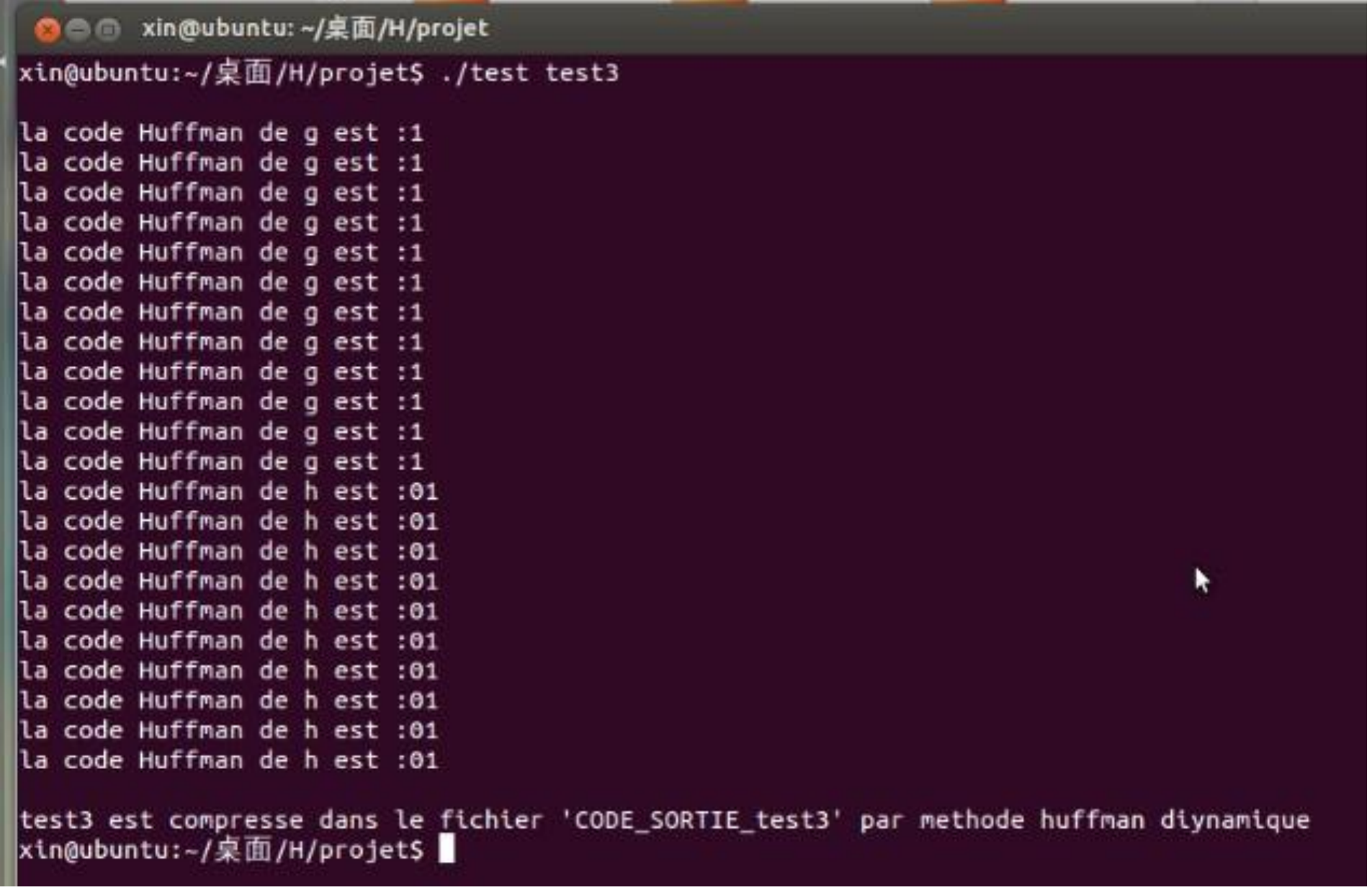
Contenu de fichier entrée “test1” : abracadabra

Contenu de fichier sortie “CODE_SORTIE_test1” : 10101010110101101100 1001



Contenu de fichier entrée “test2” : abcddbb

Contenu de fichier sortie “CODE_SORTIE_test2” : 10110100100110100 1



Contenu de fichier entrée “test3” : gggggggggggghhhhhhhhhh

Contenu de fichier sortie “CODE_SORTIE_test3”:11111111111010101010101010101

Merci !

Question de sujet :

1. l'algorithme n'échange jamais un noeud avec un de ses ancêtres

Parce que on échange un noeud avec le noeud a plus grand de numéro de noeud dans le même bloc. Mais le poids de père est toujours supérieur a ses fils, c'est a dire ils ne sont pas dans le même bloc. Doncn'échange jamais un noeud avec un de ses ancêtres.

2. Quel est (en fonction de la taille de l'alphabet des symboles) le nombre maximal d'échanges réalises lors d'une modification de l'arbre ?

Pour un arbre Huffman dont hauteur H. La nouvelle caractère entrée est c. Si c est déjà dans l'arbre, et le noeud contenant c est au font de l'arbre. On suppose si chaque noeud de ses ancêtres a besoin d'échanger lors incrémenter son poids.

le nombre maximal d'échanges est alors H+1.

3. Pour T = abracadabra, avec le code initial sur 5 bits :

a = 00000; b = 00001; c = 00010; d = 00011; : : : ; r = 10001; : : :

le texte compresse est 00000 000001 0010001 0 10000010 0 110000011 0 110 110 0 (les espaces servent a separer les dierentes transmissions).

Expliciter les dierentes etapes de la production du texte compresse, en construisant au fur et a mesure l'arbre de Huffman.

T = abracadabra,

a = 00000; b = 00001; c = 00010; d = 00011; r = 10001;

1. #

entrée : a

sortie : 00000

2. (51,1)

(49,0,#) (50,1,a)

entrée : b

sortie : (code de #)+code de b

=> 000001

3. (51,2)

(49,1) (50,1,a)

(47,0,#) (48,1,b)

entrée : r

sortie : 0010001

4. (51,3)

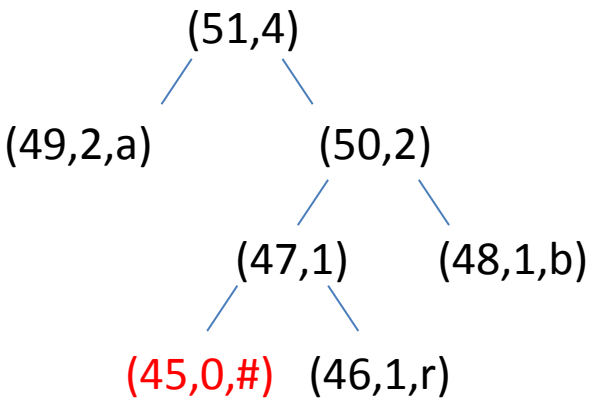
(49,1,a) (50,2)

(47,1) (48,1,b)

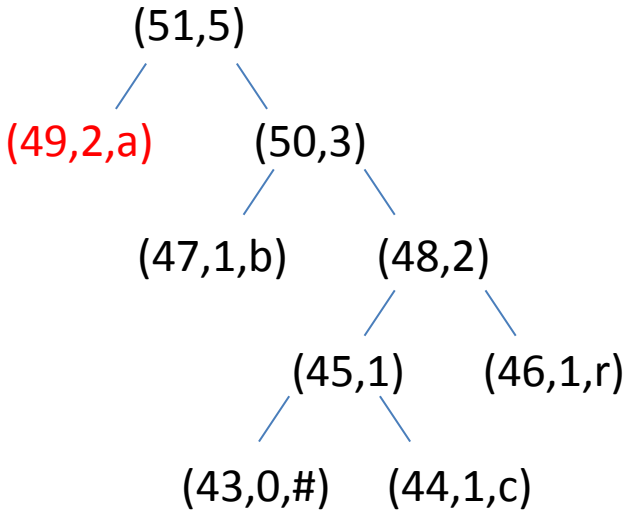
(45,0,#) (46,1,r)

entrée : a

sortie : 0

5. 

entrée : c sortie : 10000010

6. 

entrée : a sortie : 0

7. entrée : d sortie : 110000011

....