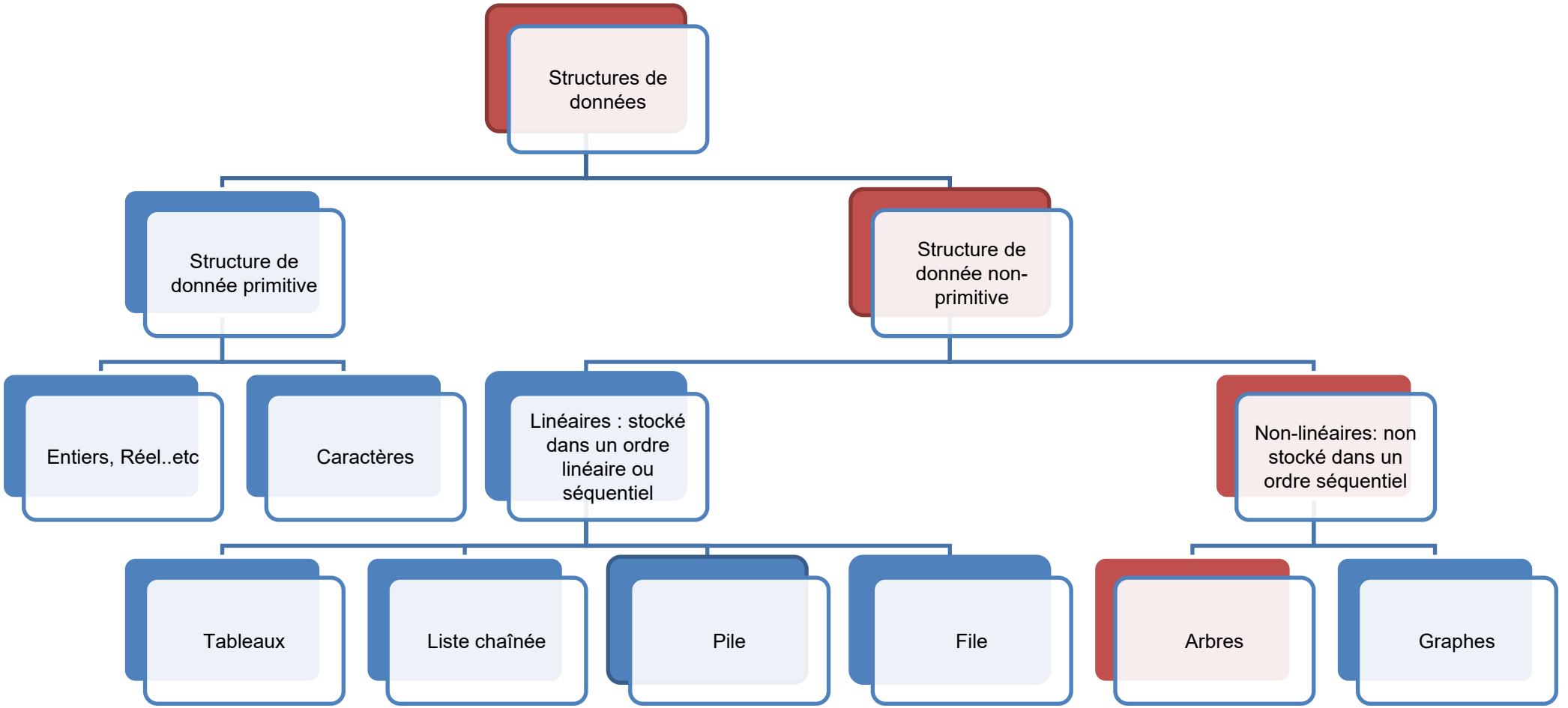
Course descriptions : semester 2, Data Science and Big Data..

Programmation & Algorithmique II

CM14: Les arbres (3)

CLASSIFICATION DES STRUCTURES DE DONNÉES

- **Structures de données primitives et non primitives**



PLAN

1. Introduction
2. Définition
3. Types des arbres
    1. Arbres généraux
    2. Forêts
    3. Arbres binaires
    4. Créer un arbre binaire à partir d'un arbre général
    5. Parcours d'un arbre binaire
    6. Opérations sur les arbres binaires
    7. Codage (arbre) Huffman

RAPPEL : REPRÉSENTATION CHAÎNÉE DES ARBRES BINAIRES

Dans la représentation chainée d'un arbre binaire, chaque nœud aura trois parties : l'élément de données, un pointeur vers le nœud gauche et un pointeur vers le nœud droit.

```
typedef int TElement;
typedef struct node * tree;
struct node
{
        TElement value;
        tree left;
          tree right;
};
```

Chaque arbre binaire a un pointeur ROOT, qui pointe vers le nœud racine (élément le plus haut) de l'arbre.

OPERATIONS SUR ARBRE BINAIRE

- **Créer arbre**
  - Cette fonction crée un arbre binaire (nœud)

```
tree Create(TElement val,tree ls, tree rs)
{
        tree res;
        res = malloc(sizeof(*res));
        if( res == NULL )
        {
                fprintf(stderr,"Impossible d'allouer le noeud");
                return NULL;
        }
        res->value = val;
        res->left = ls;
        res->right = rs;
        return res;
}
```

OPERATIONS SUR ARBRE BINAIRE

- **Arbre (ou sous-arbre) vide,**
  - Cette fonction définie si l'arbre T est vide ou pas

```
bool IsEmpty( tree T) //estvide
{
        return T == NULL;
}
```

- **Accès au fils gauche**
  - Cette fonction retourne le fils gauche de T

```
tree Left( tree T)
{
        if ( IsEmpty(T) )
                return NULL;
        else
                return T->left;
}
```

- **Accès au fils droit**
  - Cette fonction retourne le fils droit de T

```
tree Right( tree T)
{
        if ( IsEmpty(T) )
                return NULL;
        else
                return T->right;
}
```

- **Vérifier si feuille**
  - Cette fonction détermine si T est une feuille

```
bool IsLeaf(tree T)
{
        if (IsEmpty(T))
                return false;
        else if (IsEmpty(Left(T)) && IsEmpty(Right(T)))
                return true;
        else
                return false;
}
```

- **Vérifier si nœud interne**
  - Cette fonction détermine si T est un nœud interne (un nœud interne est un nœud qui n'est pas une feuille)

```
bool IsInternalNode(tree T)
{
        if IsEmpty(T)
                return false;
        else
                return ! IsLeaf(T) ;
}
```

- **Hauteur de l'arbre**
  - Cette fonction retourne la hauteur de T (profondeur maximum de T)

```
unsigned Height (tree T)
{
        if ( IsEmpty(T) )
                return 0;
        else
                return 1 + max( Height(Left(T) ) , Height(Right(T) ) );
}
```

OPERATIONS SUR ARBRE BINAIRE

- **Le nombre de nœuds**
  - Cette fonction retourne le nombre de nœud de T

```
unsigned NbNode(tree T)
{
        if( IsEmpty(T) )
                return 0;
        else
                return 1 + NbNode(Left(T)) + NbNode(Right(T));
}
```

- **Le nombre de feuilles**
  - Cette fonction détermine le nombre de feuilles de T

```
unsigned NbLeaves( tree T)
{
        if( IsEmpty(T) )
                return 0;
        else if ( IsLeave(T) )
                return 1;
        else
                return NbLeaves(Left(T)) + NbLeaves(Right(T));
}
```

- **Le nombre de nœuds internes**
  - Cette fonction détermine le nombre de nœuds internes de T

```
unsigned NbInternalNode(tree T)
{
        if (IsEmpty(T))
                return 0;
        else if(IsLeave(T))
                return 0;
        else
                return 1 + NbInternalNode(Left(T)) +
                                NbInternalNode(Right(T));
}
```

OPERATIONS SUR ARBRE BINAIRE

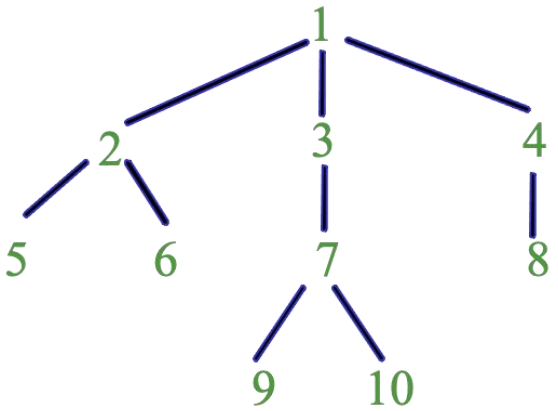
- **Parcours DFS (Depth First Search)**
- **préfixe (type 1), Infixe (type 2), postfixe (type 3)**
  - Cette fonction permet de parcourir T en DFS

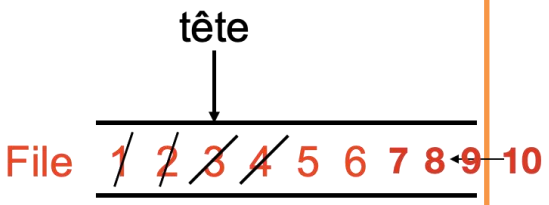
```
void DFS(tree T, char Type)
{
        if( ! IsEmpty(T) )
        {
                if( Type ==1 ) { /* traiter racine */ //printf }
                DFS(Left(T), Type);
                if (Type == 2) { /* traiter racine */ //printf }
                DFS(Right(T), Type);
                if( Type == 3){ /* traiter racine */ //printf }
        }
}
```

OPERATIONS SUR ARBRE BINAIRE

- **Parcours en largeur BFS (Breath First Search)**
  - Cette fonction effectue un parcours en largeur (BFS) de T

```
void BFS(tree T)
{       tree Temp;
        File F;
        if( ! IsEmpty(T) )
        {       Enfiler(&F,T);
                while( ! EstVide(F) )
                {       Defiler(&F, &Temp);
                        /* Traiter la racine */ //printf
                        if( ! IsEmpty(Left(Temp)) )
                                Enfiler(&F, Left(Temp));
                        if( ! IsEmpty(Right(Temp)) )
                                Enfiler(&F, Right(Temp));
}       }       }
```



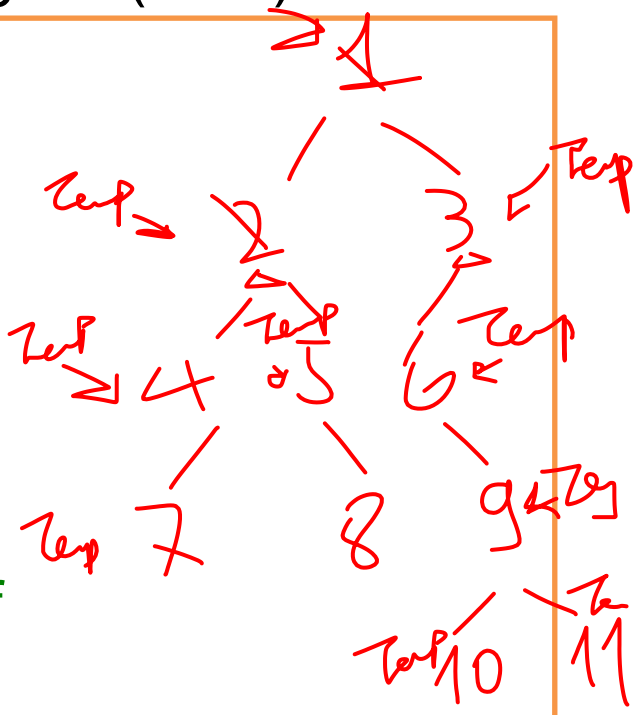


Liste *L* = (1, 2, .

OPERATIONS SUR ARBRE BINAIRE

- **Parcours en largeur BFS (Breath First Search)**
  - Cette fonction effectue un parcours en largeur (BFS) de T

```
void BFS(tree T)
{       tree Temp;
        File F;
        if( ! IsEmpty(T) )
        {       Enfiler(&F,T);
                while( ! EstVide(F) )
                {       Defiler(&F, &Temp);
                        /* Traiter la racine */ //printf
                        if( ! IsEmpty(Left(Temp)) )
                                Enfiler(&F, Left(Temp));
                        if( ! IsEmpty(Right(Temp)) )
                                Enfiler(&F, Right(Temp));
}       }       }
```

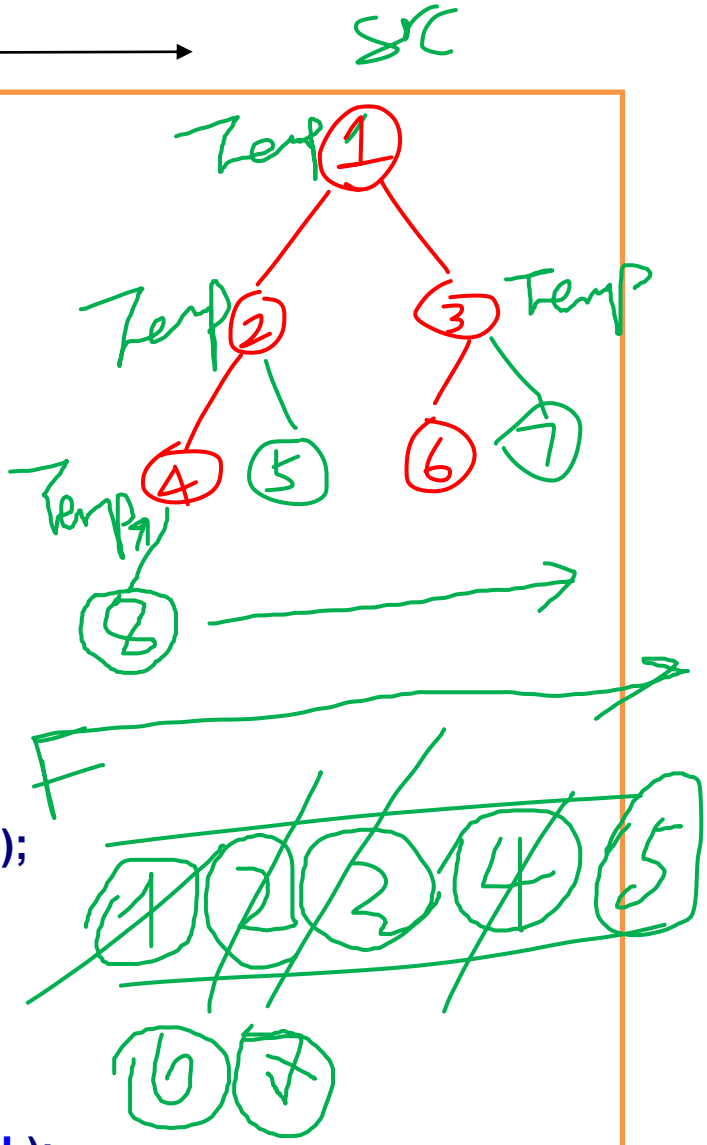


- **Ajout d'élément**
  - Cette fonction permet d'ajouter un élément dans l'arbre binaire selon les règles suivantes (insertion en margeur-BFS):
    - Les éléments sont ajoutés de gauche à droite dans un même niveau
    - Un élément ne peut être rajouté dans un niveau sauf si le niveau précédant est rempli

Complet

OPERATIONS SUR ARBRE BINAIRE

```
void AddElt(tree src, TElement elt) {
        tree Temp;
        File F;
        if ( src == NULL ){
                src = Create(elt,NULL,NULL);
        }
        else { //utiliser le parcours BFS avec file
                Enfiler(&F,src);
                while( ! EstVide(F) )
                {
                        Defiler(&F, &Temp);
                        if( ! IsEmpty(Left(Temp)) )
                                Enfiler(&F,Left(Temp));
                        else {
                          Temp->left = Create(elt,NULL,NULL);
                          break;
                        }
                        if( ! IsEmpty(Right(Temp)) )
                                Enfiler(&F,Right(Temp));
                        else {
                          Temp->right = Create(elt,NULL,NULL);
                          break;
                        }
}       }       }
```



OPERATIONS SUR ARBRE BINAIRE

- **Recherche dans un arbre**
  - Cette fonction détermine si ***elt*** existe dans T

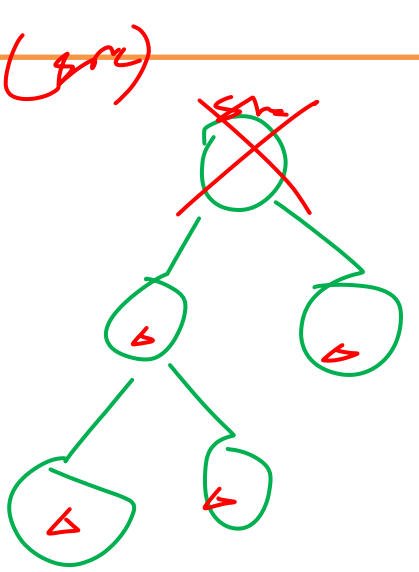
```
bool Exist(tree src , TElement elt)
{
        if ( IsEmpty(src) )
                return false;
        else if ( src->value == elt )
                return true;
        else
                return Exist(Left(src), elt) || Exist(Right(src), elt);
}
```

- **Suppression d'un arbre**
  - Cette fonction supprime src et libère l'espace mémoire occupé

```
void Erase(tree * src) {
        tree ls = Left(*src);
        tree rs = Right(*src);

        if( ! IsEmpty(*src) ) {
                Erase( &ls );
                Erase( &rs );

                free( *src );
                *src = NULL;
        }
}
```

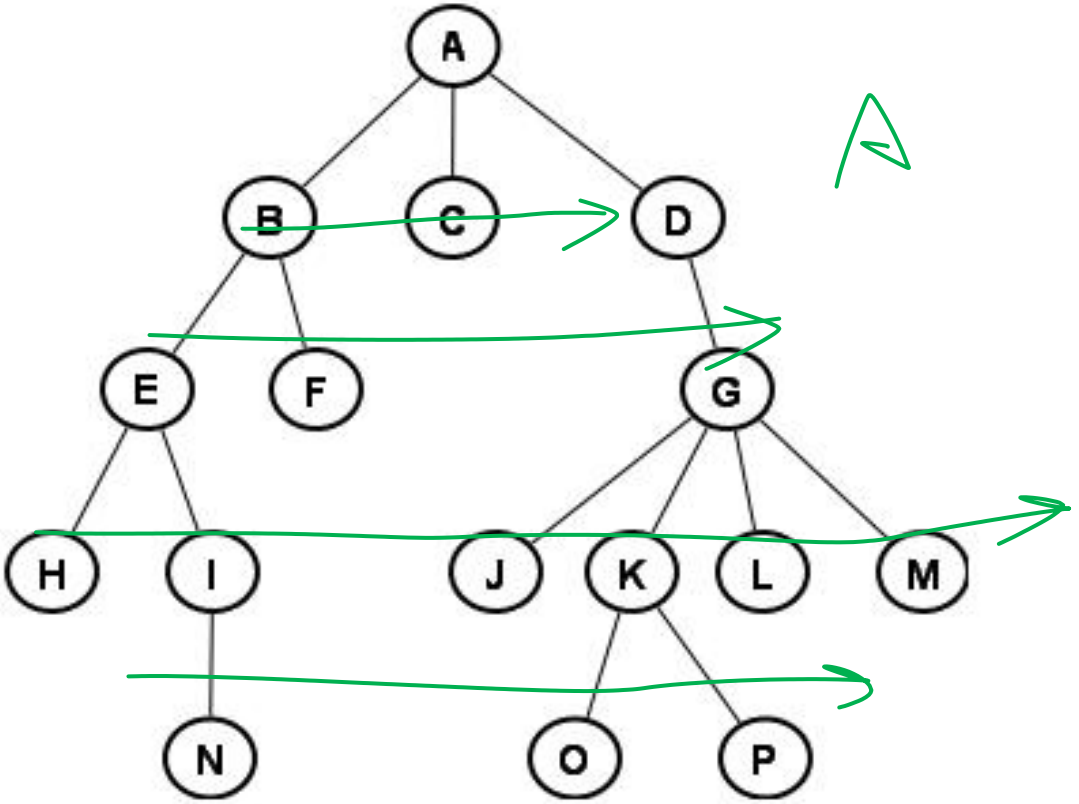


QUIZ

Quel est le résultat du parcours BFS de cet arbre :

A- ABEHINFCDGJKOPLM

B- ABCDEFGHIJKLMNOP



5 minute

浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

CERGY PARIS UNIVERSITÉ

CODAGE (ARBRE) HUFFMAN (1)

Le codage Huffman est un **algorithme d'encodage** développé par David A. Huffman qui est largement utilisé comme une technique de **compression de données** sans perte. L'algorithme de codage Huffman utilise une table de code de longueur variable pour coder un caractère source où la table de code de longueur variable est dérivée sur la base de la probabilité estimée d'occurrence du caractère source.

L'idée clé derrière l'algorithme Huffman est qu'il code **les caractères les plus courants en utilisant des chaînes de bits plus courtes** que celles utilisées pour les caractères sources moins communs.

CODAGE HUFFMAN EXAMPLE :

Comment codé ***abracadabra*** avec le moins de bits possibles ?

Codage à longueur fixe

- 00000110100001000001100000 01101000
  avec :

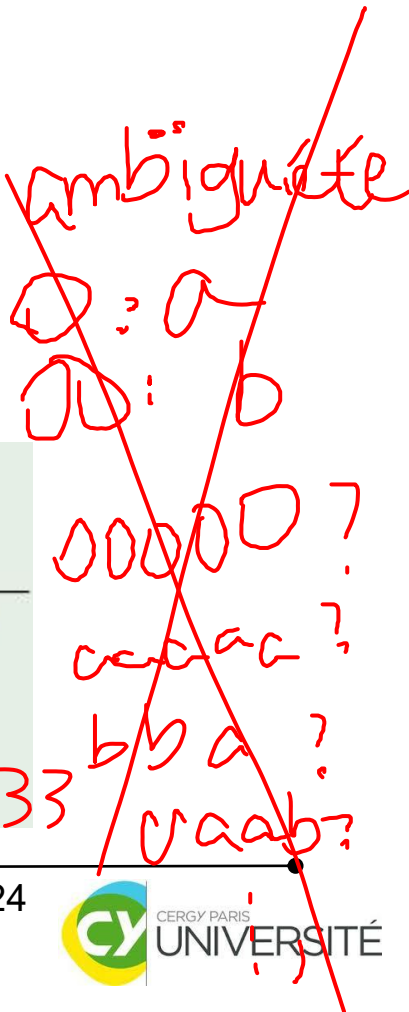
a	b	c	d	e	r
000	001	010	011	100	101

Codage à longueur variable

- avec :

a	b	c	d	e	r
0	101	100	111	1101	1100

  - abracadabra
  - 01011100010011101011000



CODAGE (ARBRE) HUFFMAN (2)

Remarques

- Les caractères n'ont pas tous la même fréquence,
  - Supposons les fréquences suivantes:

a	b	c	d	e	r
45	13	12	16	9	5

- les caractères fréquents occupent beaucoup de place...
- Idée : codage de longueur variable : peu de bits pour les fréquents, plus pour ceux qui le sont moins.

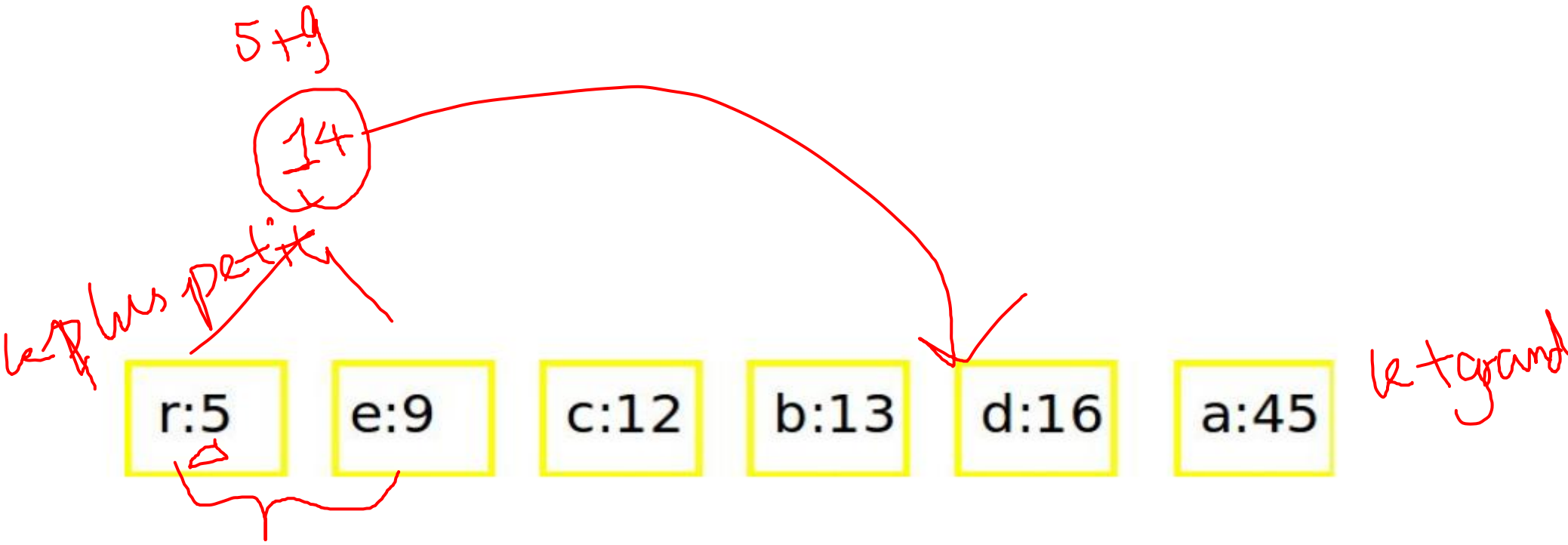
CODAGE (ARBRE) HUFFMAN (2)

L'algorithme fonctionne en créant un arbre binaire de nœuds qui sont stockés dans un tableau:

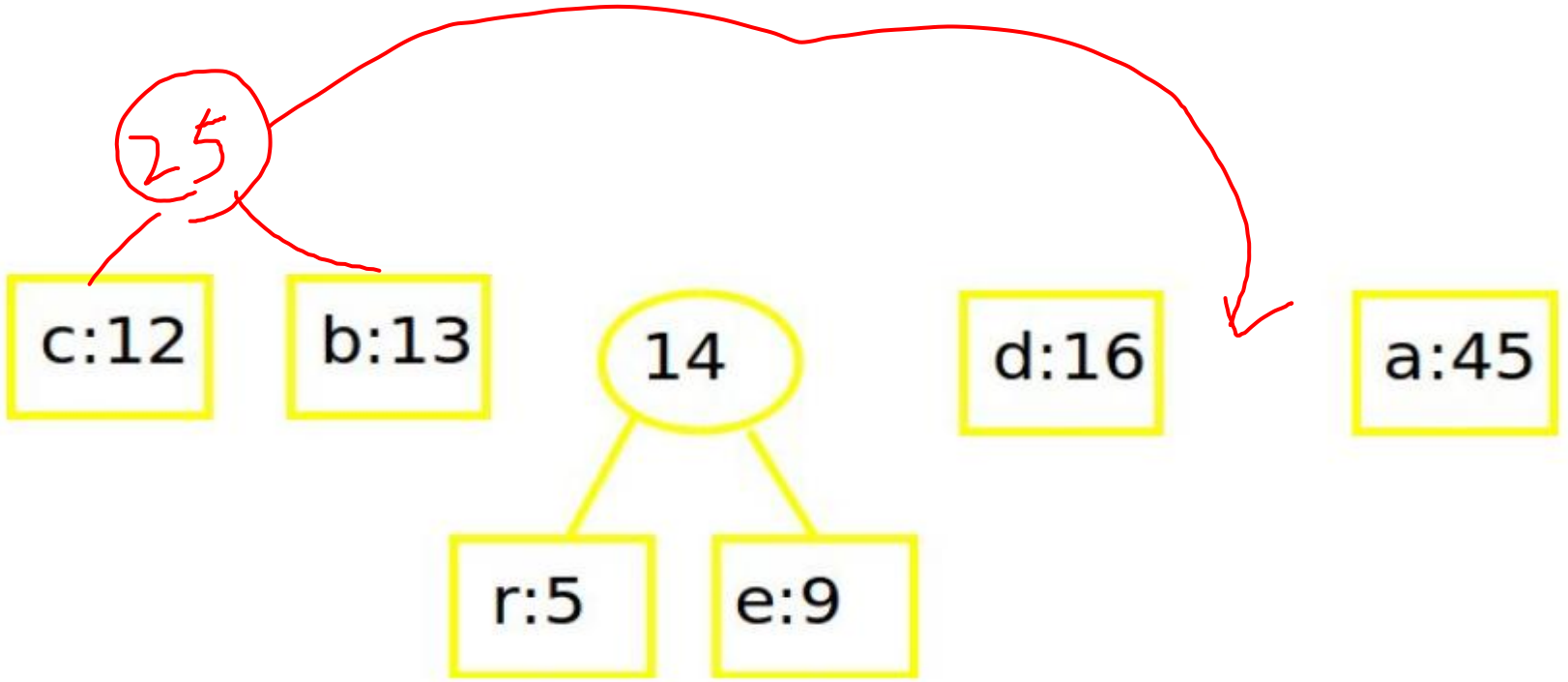
- Un nœud peut être soit un nœud feuille ou un nœud interne.
- Initialement, tous les nœuds de l'arbre sont au niveau des feuilles et stockent le caractère source et sa fréquence d'occurrence (également connu sous le nom de poids).
- Alors que le nœud interne est utilisé pour stocker le poids et contient des liens vers ses nœuds fils, le nœud externe (feuille) contient le caractère.
- Traditionnellement, un « 0 » représente la sélection le fils gauche et un « 1 » représente la sélection du fils droit.
- Un arbre complet qui a *n* nœuds feuille aura *n - 1* nœuds internes,

.

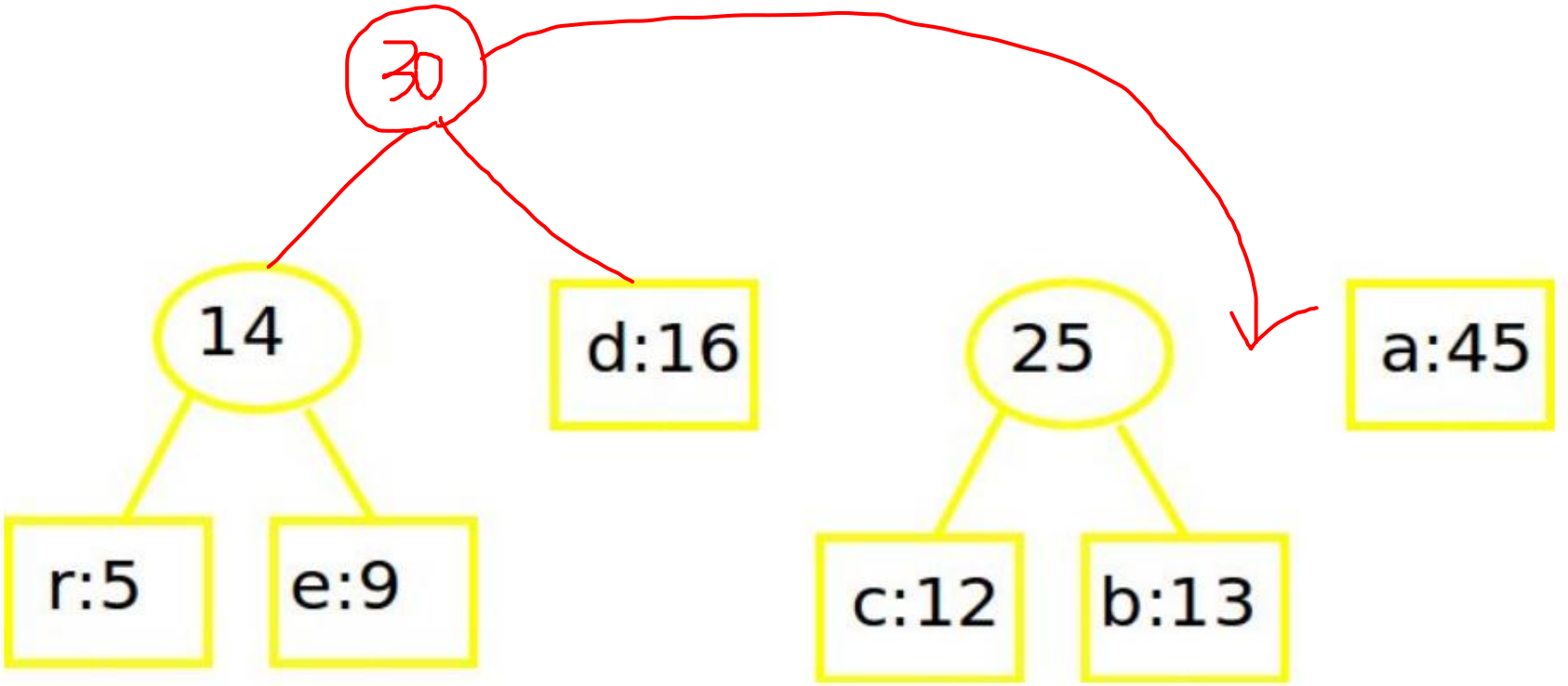
CODAGE (ARBRE) HUFFMAN



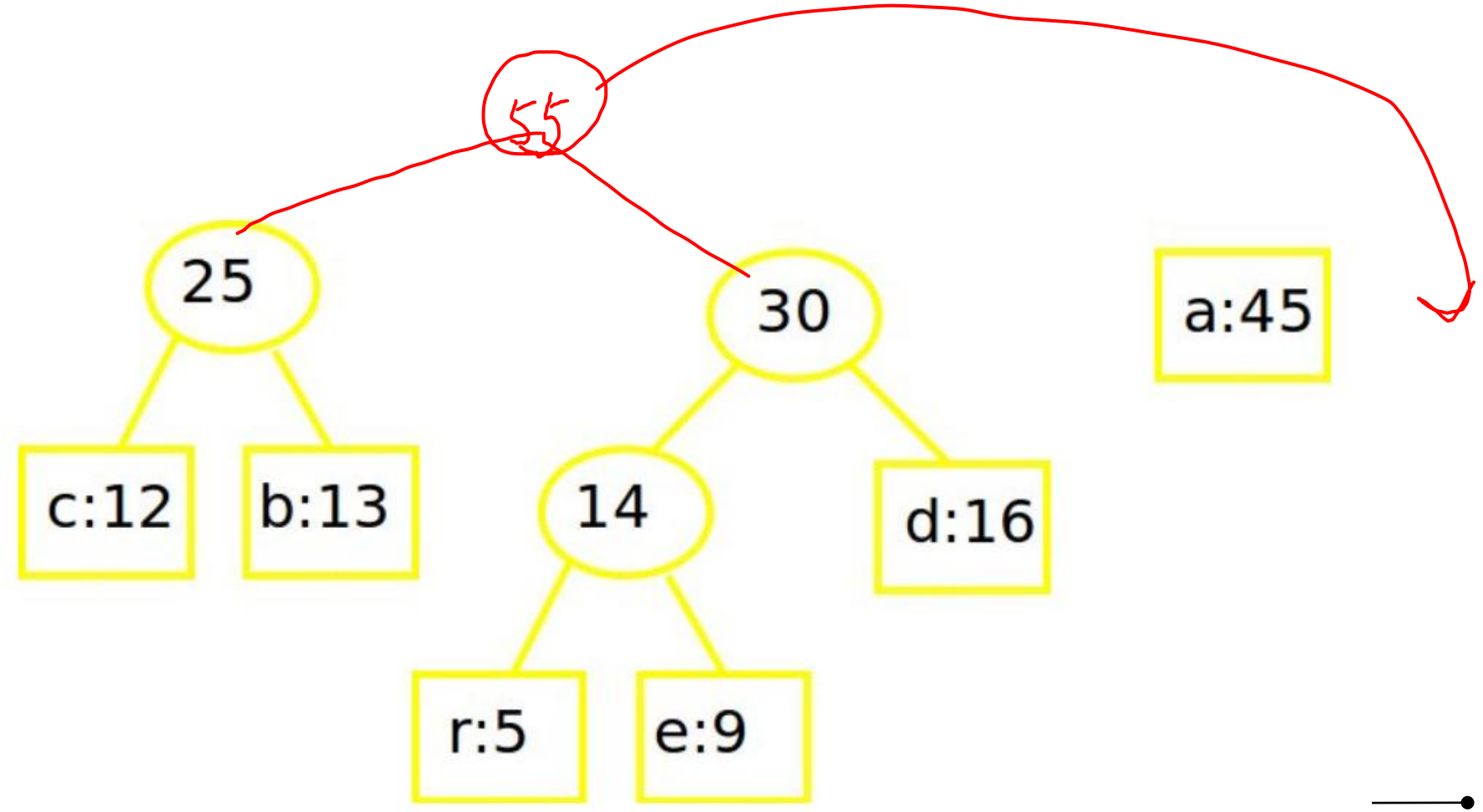
CODAGE (ARBRE) HUFFMAN



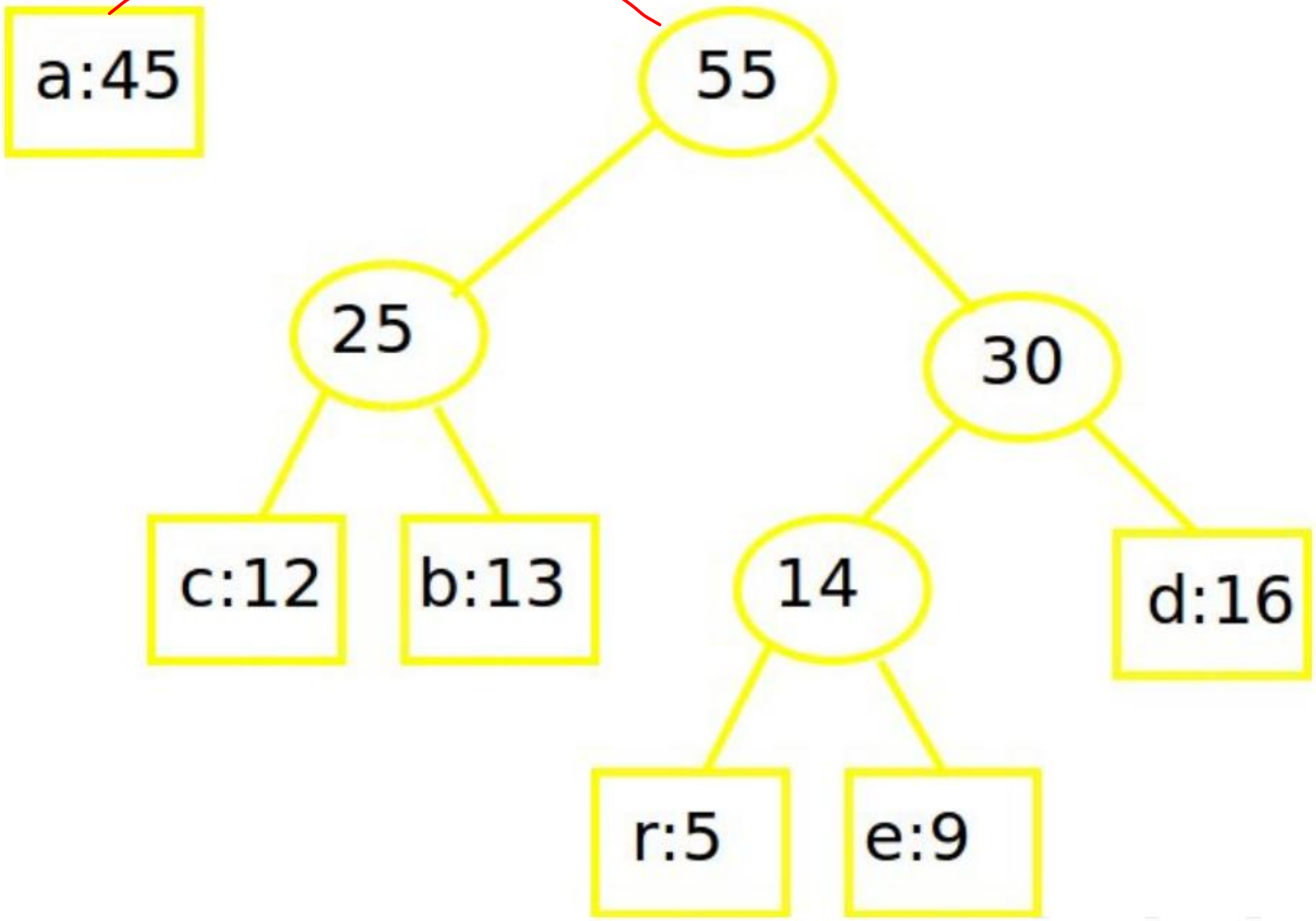
CODAGE (ARBRE) HUFFMAN



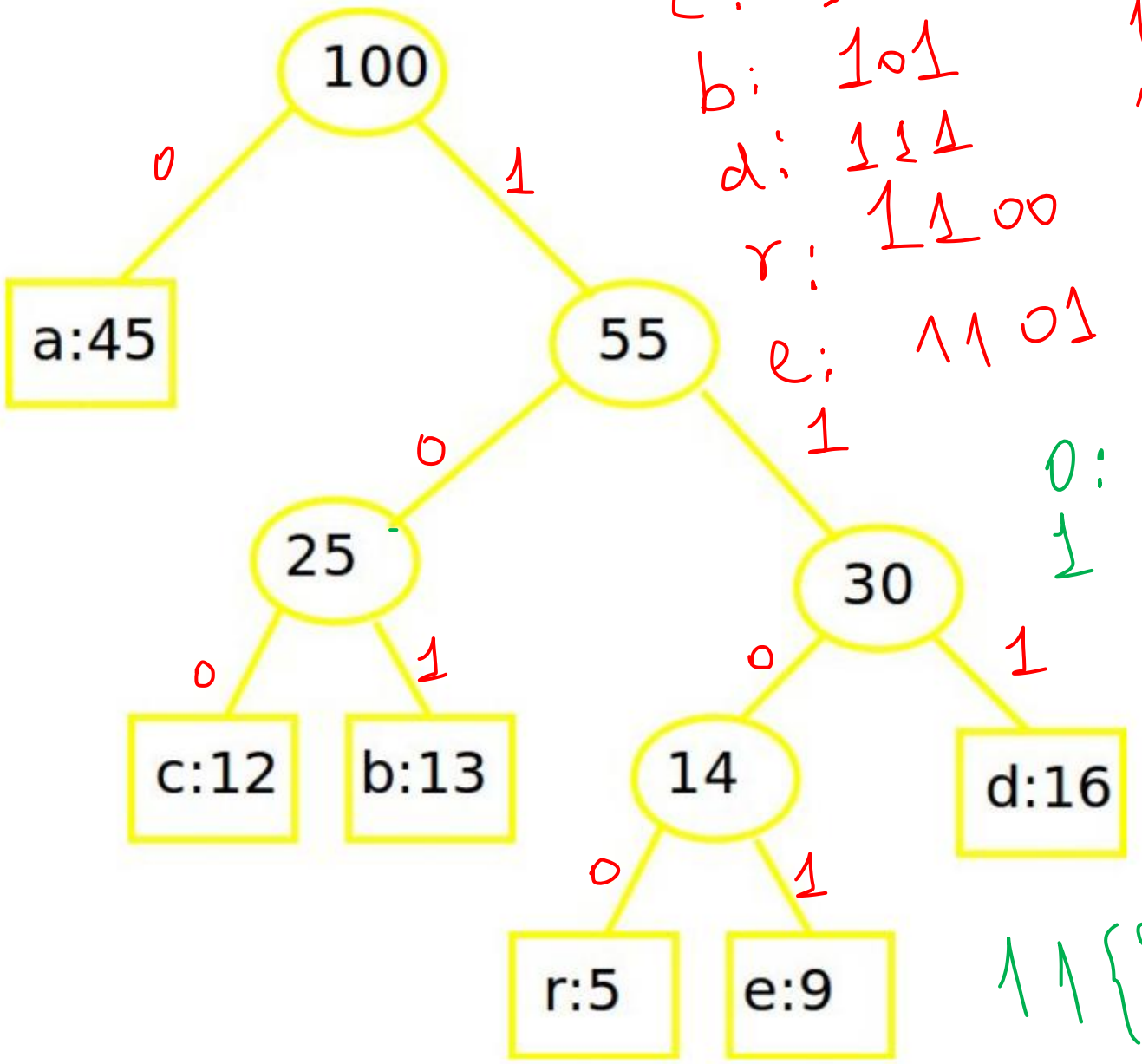
CODAGE (ARBRE) HUFFMAN



CODAGE (ARBRE) HUFFMAN



CODAGE (ARBRE) HUFFMAN



a: 0

c: 100

b: 101

d: 111

r: 1100

e: 1101

45 (1)

12 (3)

13 (3)

16 (3)

5 (4)

9 (4)

0 a

1 0 1 0

b a

0: a

1 { b c d e r

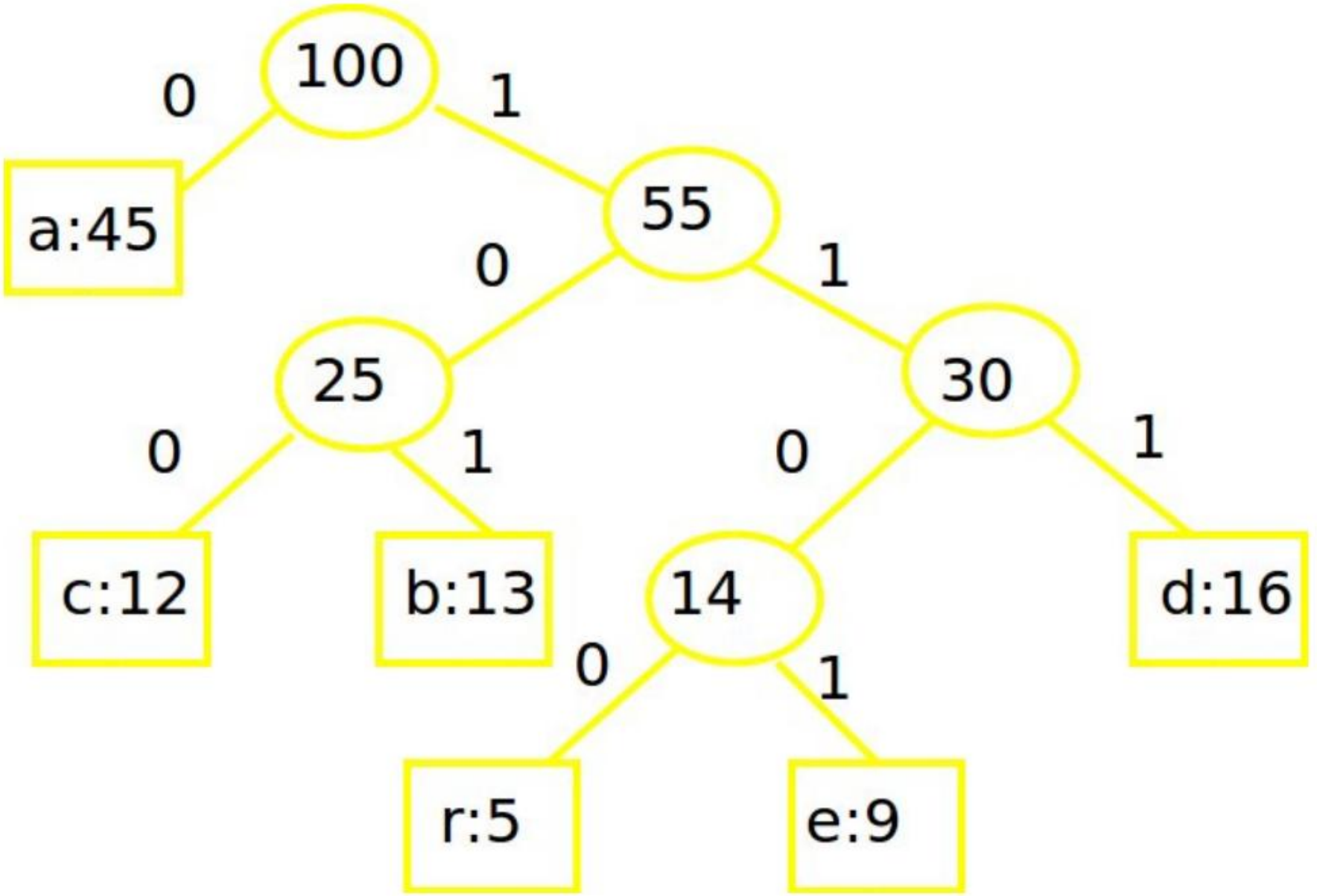
10 { b

11 { d e

110 { r e

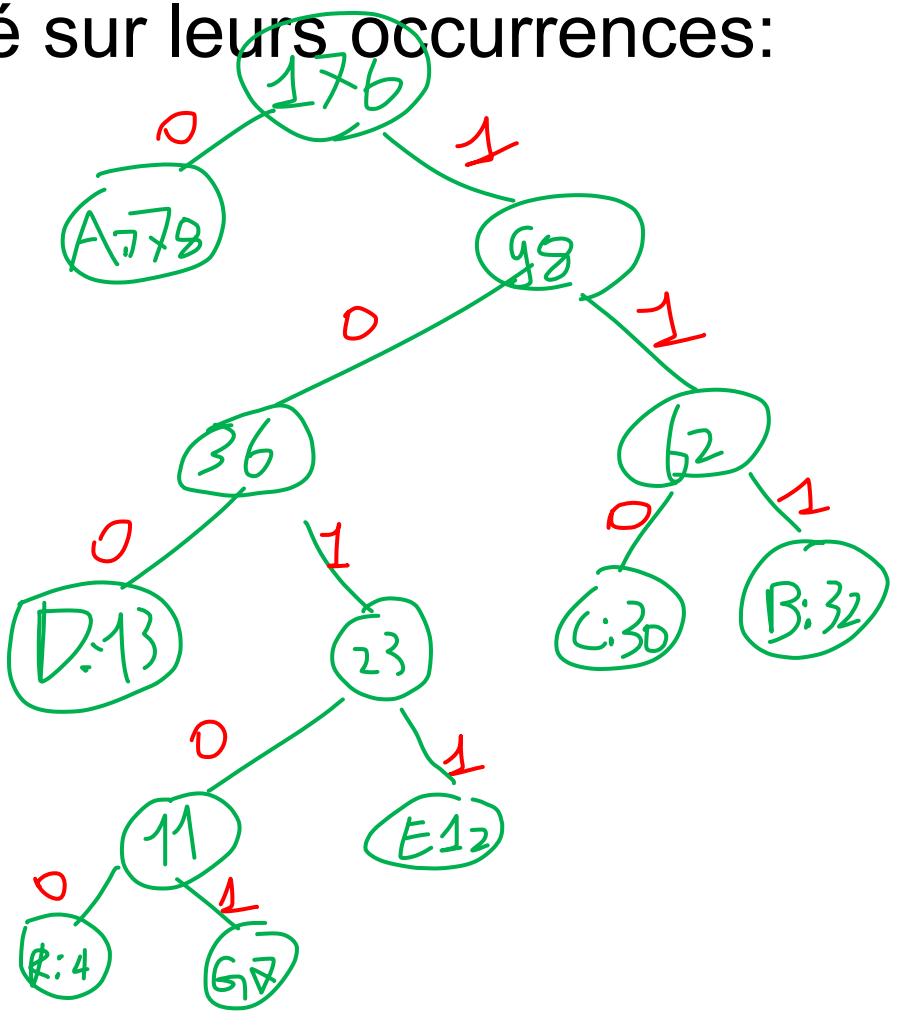
111 d

CODAGE (ARBRE) HUFFMAN



Exercice

- Encodez les caractères suivants par l'encodage Huffman basé sur leurs occurrences:
  - A: 78 0
  - B: 32 111
  - C: 30 110
  - D: 13 100
  - E: 12 1011
  - G: 7 10101
  - R: 4 10100



78×1+
32×3+
30×3+
13×3+
12×4+
7×5+
4×5
=406

Fix = 3×(78+32+---4)=528