Structures de données

Christophe Damas
José Vander Meulen

Planning

- S2 – début S3: Structures linéaires, ensembles et dictionnaires
- fin S3: Files de priorité
- S4: Graphes
- S5 – S6 – S7: Projet
- S8 – S9 –S10: Arbres
- S11 – S12: Code de Huffmann

Evaluation

- En juin
  - 25 % projet sur les graphes
  - 75 % examen sur machine
- En septembre
  - 100 % examen sur machine

STRUCTURES LINÉAIRES, ENSEMBLES ET DICTIONNAIRES :
RAPPEL + IMPLEMENTATION JAVA

Attention, cette présentation n'est qu'un bref rappel.
Pour plus d'informations, référez-vous à la javadoc.

Vecteur

```
public class Vector<E> {
    boolean isEmpty();
    int size();
    E elementAt(int index);
    void add(int index, E element);
    void add(E element);
    E remove(int index);
    String toString();
}
```

- Complexité ?

Pile

```
public class Stack<E> extends Vector<E> {
      void push(E item);
      Object pop();
      Object peek();
}
```

File

```
public interface Queue<E>{
    boolean empty();
    void add(E e);
    E poll();
    E peek();
    int size();
    String toString();
}
```

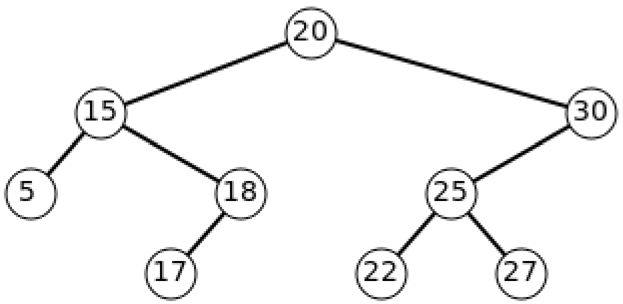
- Les classes implémentant cette interfaces utilisent soit un tableau (ArrayDeque) ou soit le chaînage (LinkedList)

Ensemble

```
public interface Set<E>{
    boolean isEmpty() ;
    int size();
    boolean add(E e) ;
    boolean contains(Object o) ;
    boolean remove(Object o) ;
    String toString() ;
}
```

Ensemble: implémentation

- Ensemble non trié
  - table de hashage: HashSet
    - Operations en O(1)
- Ensemble trié
  - implémente l'interface SortedSet
    - Méthodes supplémentaires first(), last()
  - arbre binaire: TreeSet
    - Operations en O(log(N))



Dictionnaire

```
public interface Map <K,V>{
    boolean isEmpty() ;
    int size();
    boolean put(K key,V value);
    boolean containsKey(K key);
    Object remove(Object key);
    Object get(Object key);
}
```

Dictionnaire: Implémentation

- Dictionnaire non trié: HashMap
  - Table de hashage
    - Opérations: O(1)
- Dictionnaire trié: TreeMap
  - Utilisation arbre binaire
    - Operations: O(log(N))
  - Méthodes supplémentaires pour obtenir les clés les plus basses/hautes

Files de priorité

Christophe Damas
José Vander Meulen

File de priorité

```
public interface PriorityQueue{
    void insert (Comparable v);
    Comparable max();
    Comparable delMax();
    boolean isEmpty();
    int size();
}
```

Implémentation

- Pour implémenter les files de priorité, on pourrait utiliser certaines des structures vues jusqu'à présent.

complexité	Vecteur/Liste non trié	Vecteur/Liste trié
insert		
delMax		

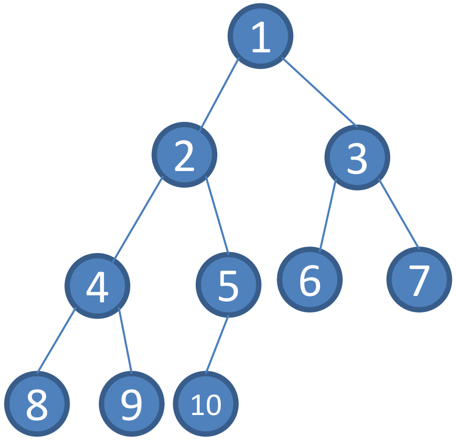
Implémentation: Heap

- Il existe une implémentation où les opérations se font en O(log(n)).
  - Donc, l'insertion et le traitement de n éléments se feront en O(n*log(n)).

complexité	Vecteur/Liste non trié	Vecteur/Liste trié	Tas
insert	O(1)	O(n)	O(log(n))
delMax	O(n)	O(1)	O(log(n))

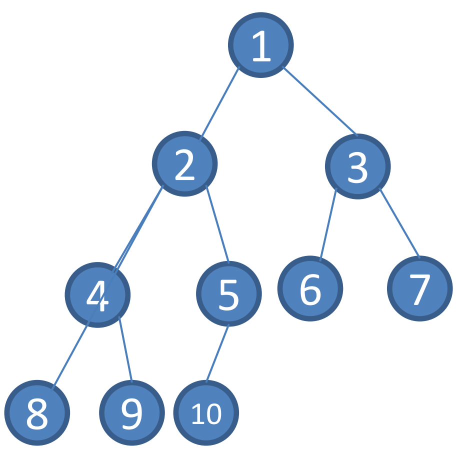
Arbre binaire complet

- Arbre dont toutes les feuilles sont au même niveau ou sur deux niveaux adjacents et si toutes les feuilles situées au niveau le plus bas sont le plus à gauche possible.



Implémentation arbre binaire complet

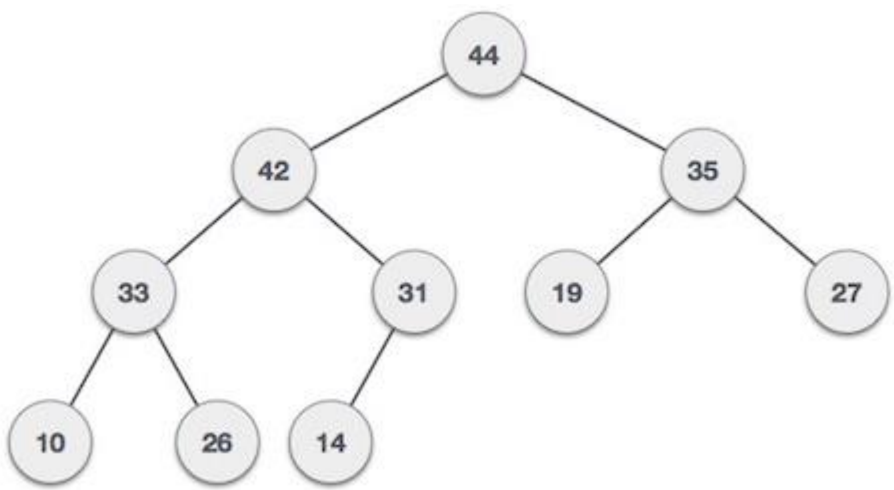
- Il est facile d'implémenter un arbre binaire complet à l'aide d'un tableau.
  - La racine est en position 1
  - Les enfants d'un nœud sont en position 2k et 2k+1



-	1	2	3	4	5	6	7	8	9	10	
0	1	2	3	4	5	6	7	8	9	10	

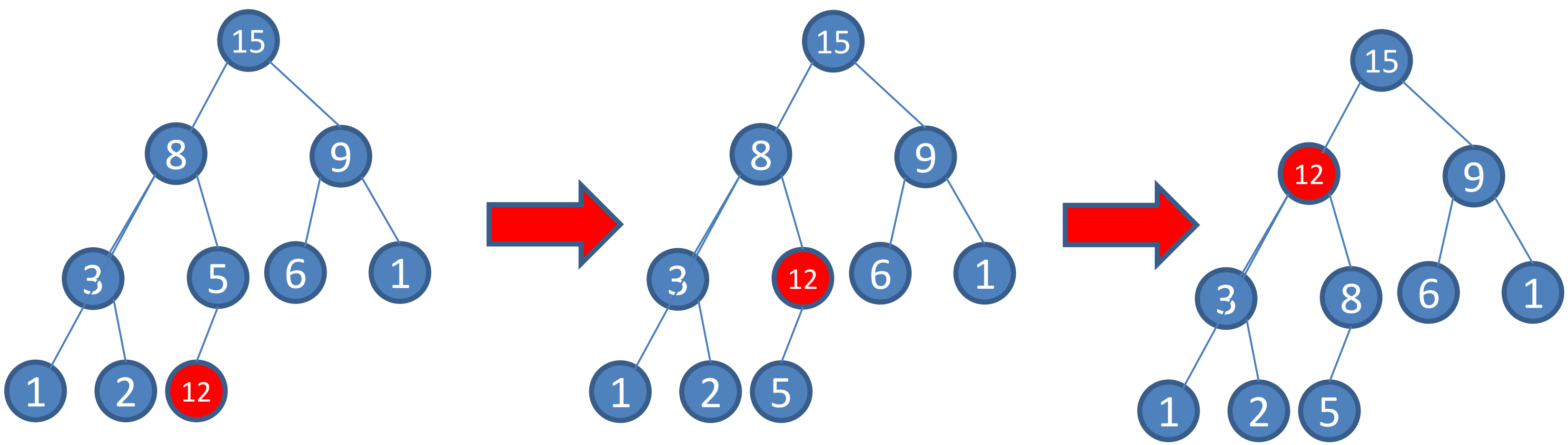
Tas

- Un tas est un arbre binaire complet pour lequel la valeur de chaque noeud est supérieure ou égale à celle de ses fils



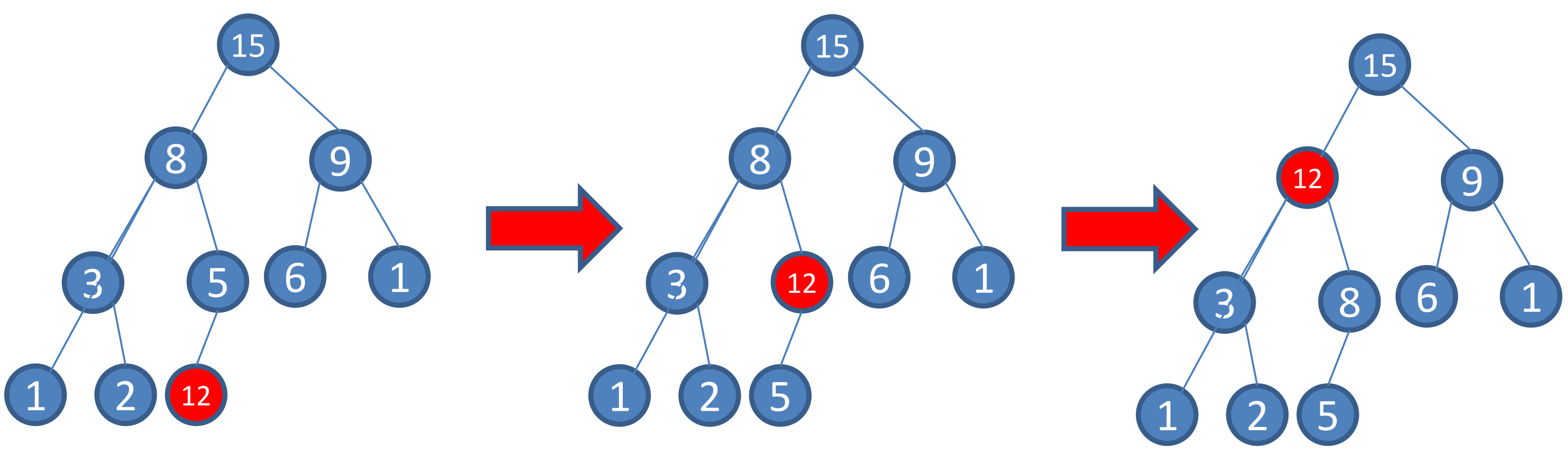
Algorithme sur les tas: swim

- Si la propriété du tas est violé car un noeud est plus grand que son parent, on peut régler le problème en échangeant ce noeud avec son parent



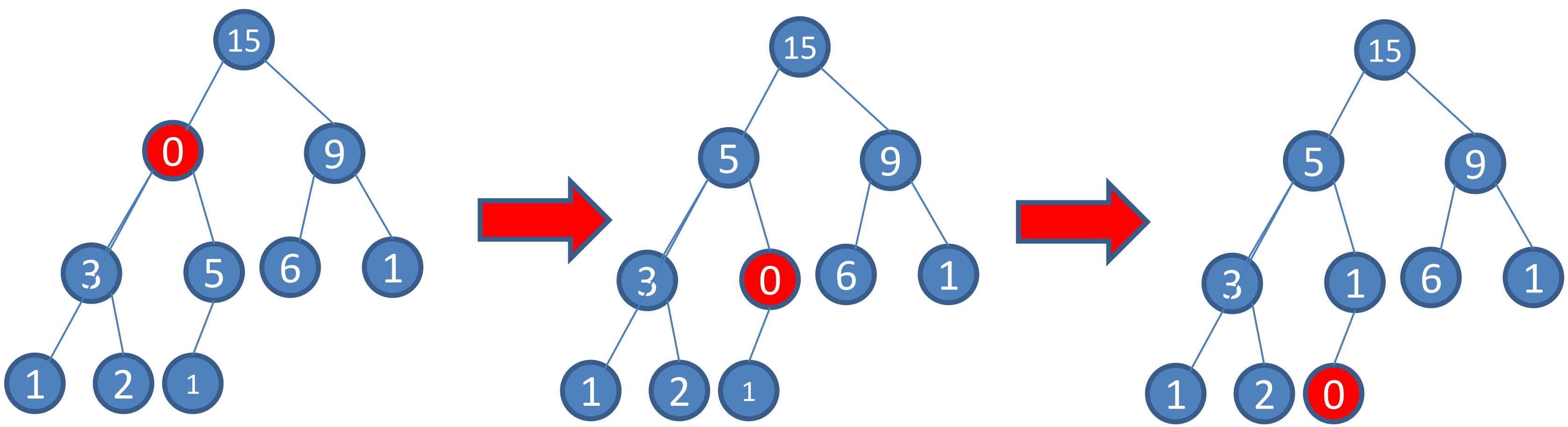
swim: pseudo-code

```
private void swim(int k){
      while (k>1 && less(k/2,k)){
            exchange(k/2,k);
            k=k/2;
      }
}
```



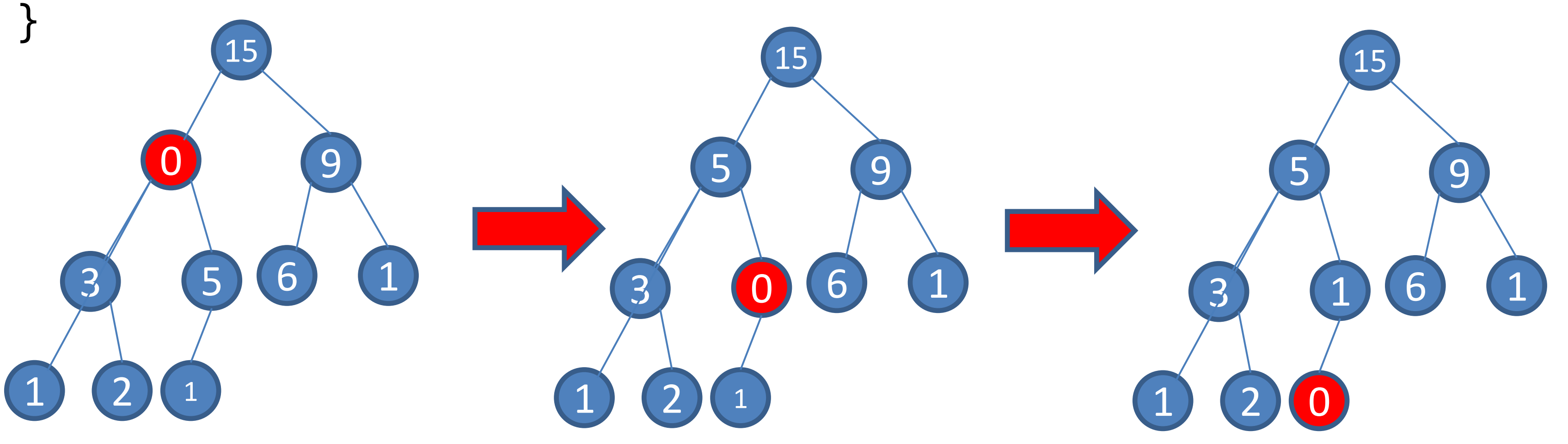
Algorithme sur les tas: sink

- Si la propriété du tas est violé car un noeud est plus petit que ses enfants, on peut régler le problème en échangeant ce noeud avec le plus grand de ses enfants



sink: pseudo-code

```
private void sink(int k){
       while (2*k<=N){
              int j=2*k;
              if(j<N && less(j,j+1)) j++:
              if (!less(k,j)) break;
              exchange(k,j);
              k=j;
       }
}
```

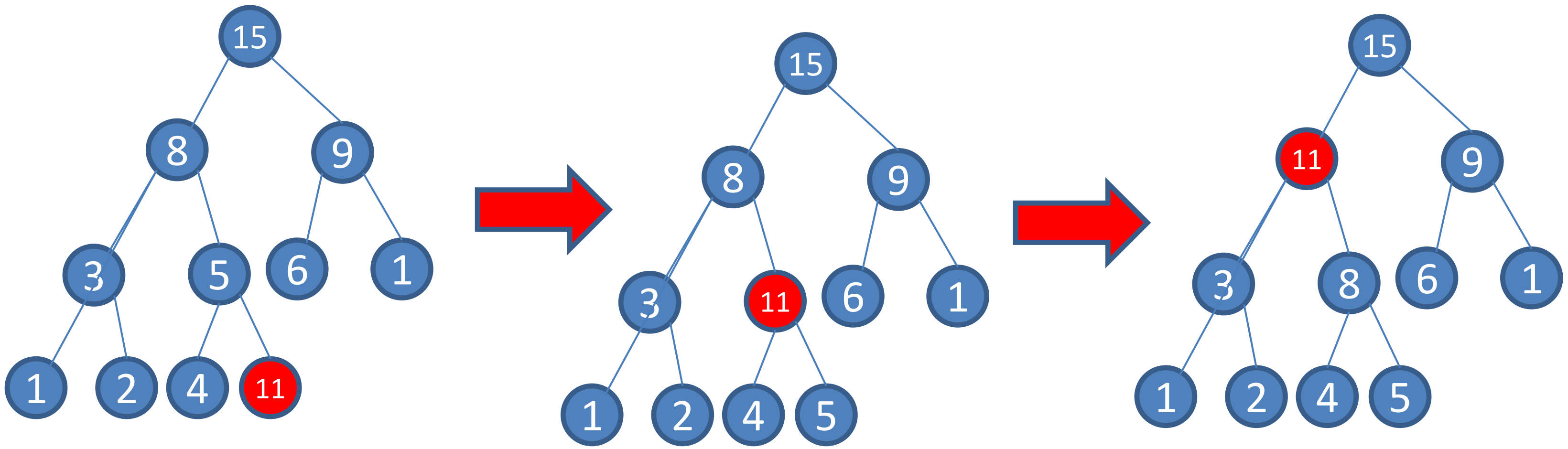


Retour aux méthodes

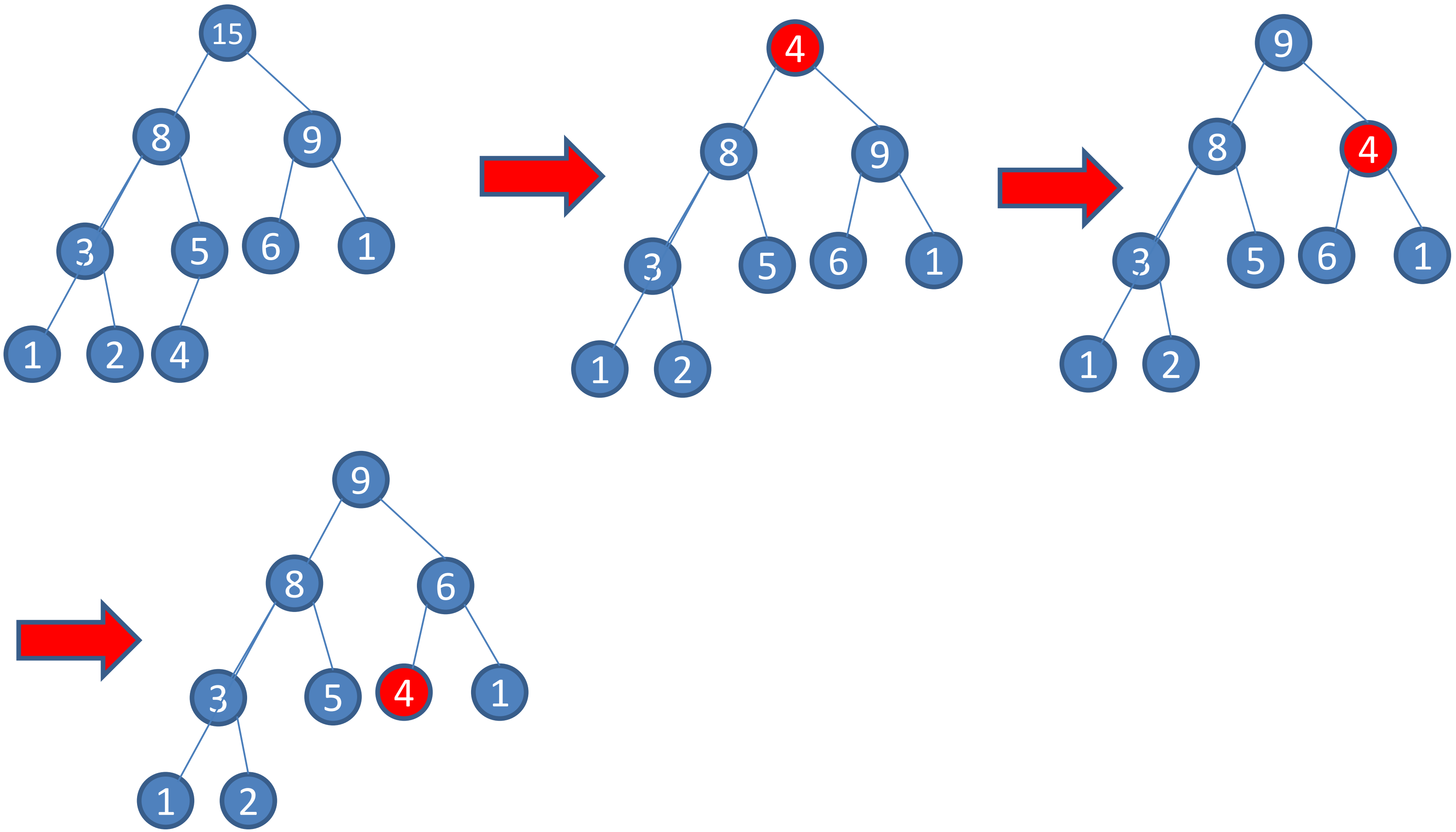
```
public interface PriorityQueue{
      void insert (Comparable v);
      Comparable max();
      Comparable delMax();
      boolean isEmpty();
      int size();
}
```

- insert: on ajoute la valeur à la fin du tableau, on incrémente la taille du tas et on exécute swim sur ce noeud pour rétablir la propriété du tas
- delMax: On enlève l'élément maximum (la racine) et le remplace par le dernier élément du tas sur lequel on exécute sink pour rétablir la propriété du tas

Insertion de la valeur 11



Suppression du maximum



Complexité

- Les 2 opérations en O(log(n)) car l’arbre binaire est complet

Heap Sort

- Si on insère n éléments dans une file de priorité, puis qu'on supprime le maximum, la dernière case du tableau devient libre. On peut donc y stocker le maximum. En continuant de la sorte, le tableau sera trié.
- Le gros avantage de Heapsort est d'être en O(n*log(n)) même dans le pire des cas (contrairement à Quick Sort qui est $O(n^2)$).
- Heapsort utilise une technique légèrement modifiée pour construire le tas initial contenant tous les éléments du tableau. Elle utilise uniquement le sink().

En java

- class PriorityQueue
- Voir Javadoc

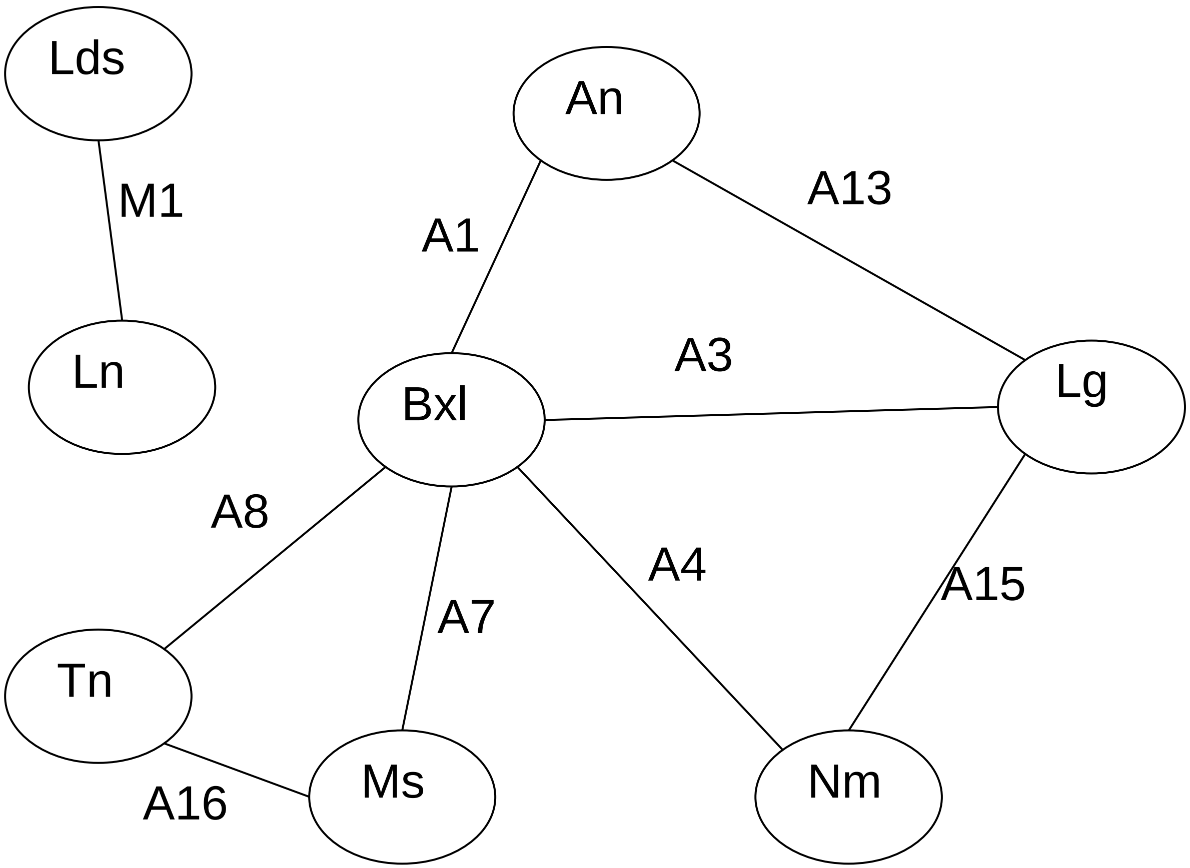
Exercice sur papier

- Insérez les éléments suivants dans un tas dans cet ordre: 57 85 44 21 23 52 17 7 95
- Donnez le tableau représentant ce tas après ces 9 insertions
- Enlevez deux fois le maximum et représenter le tableau après cette opération

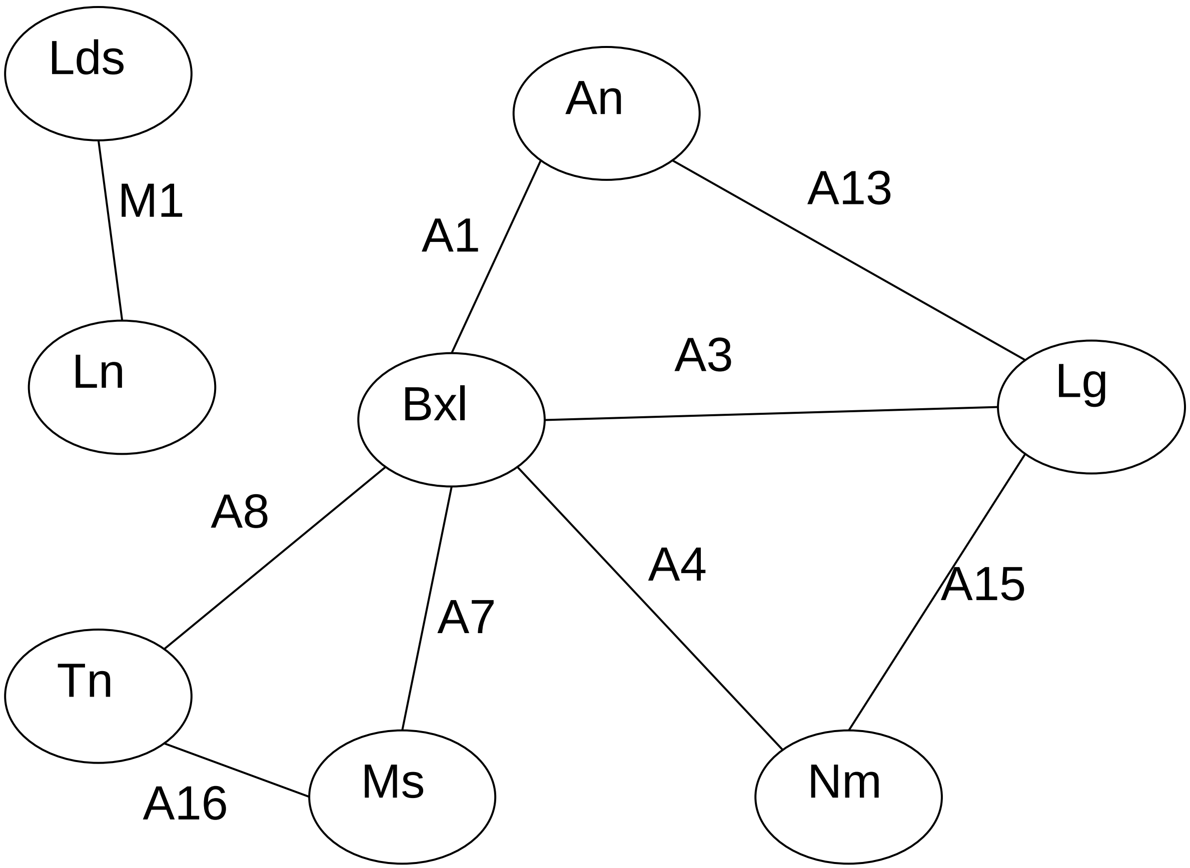
Les Graphes

(slides basés sur ceux de A. Dupont et M. Marchand)

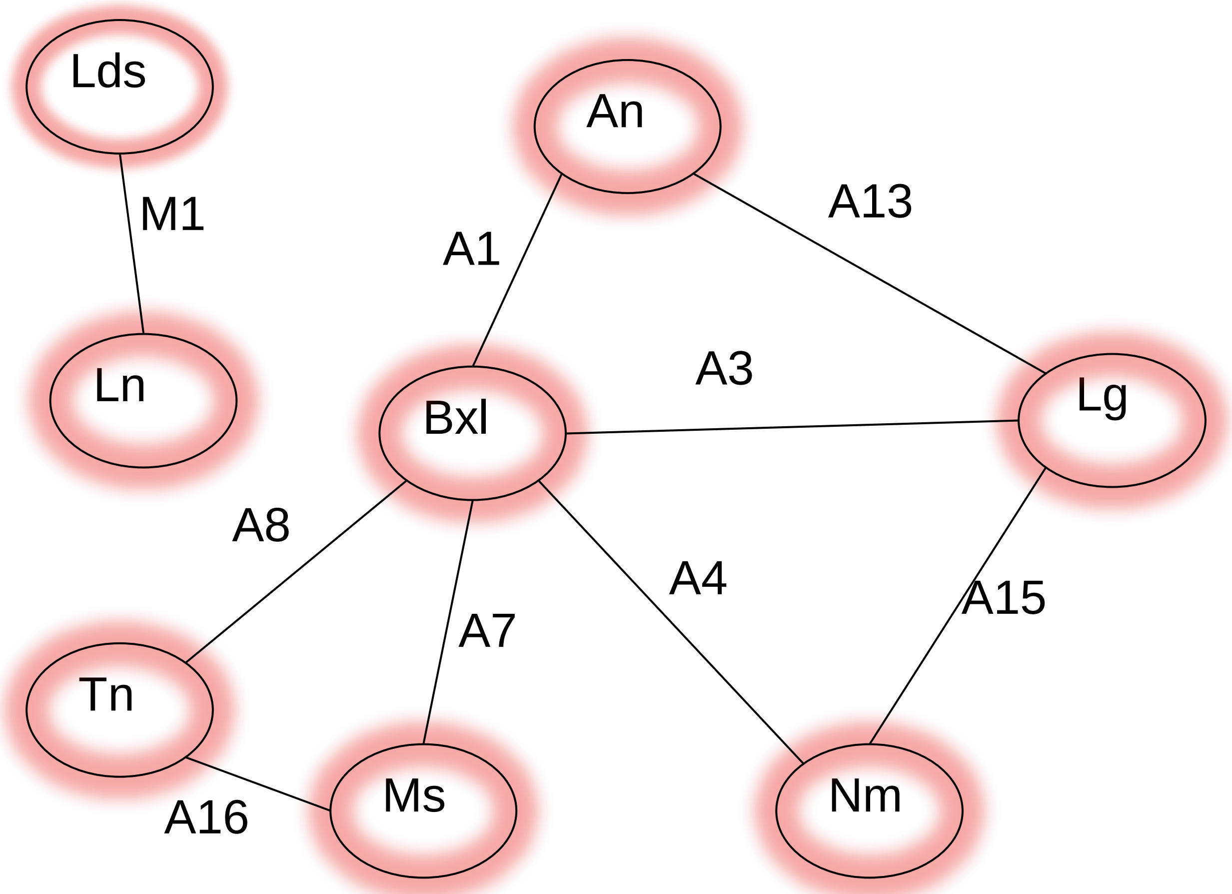
Exemple 1 : graphe non dirigé



Les sommets

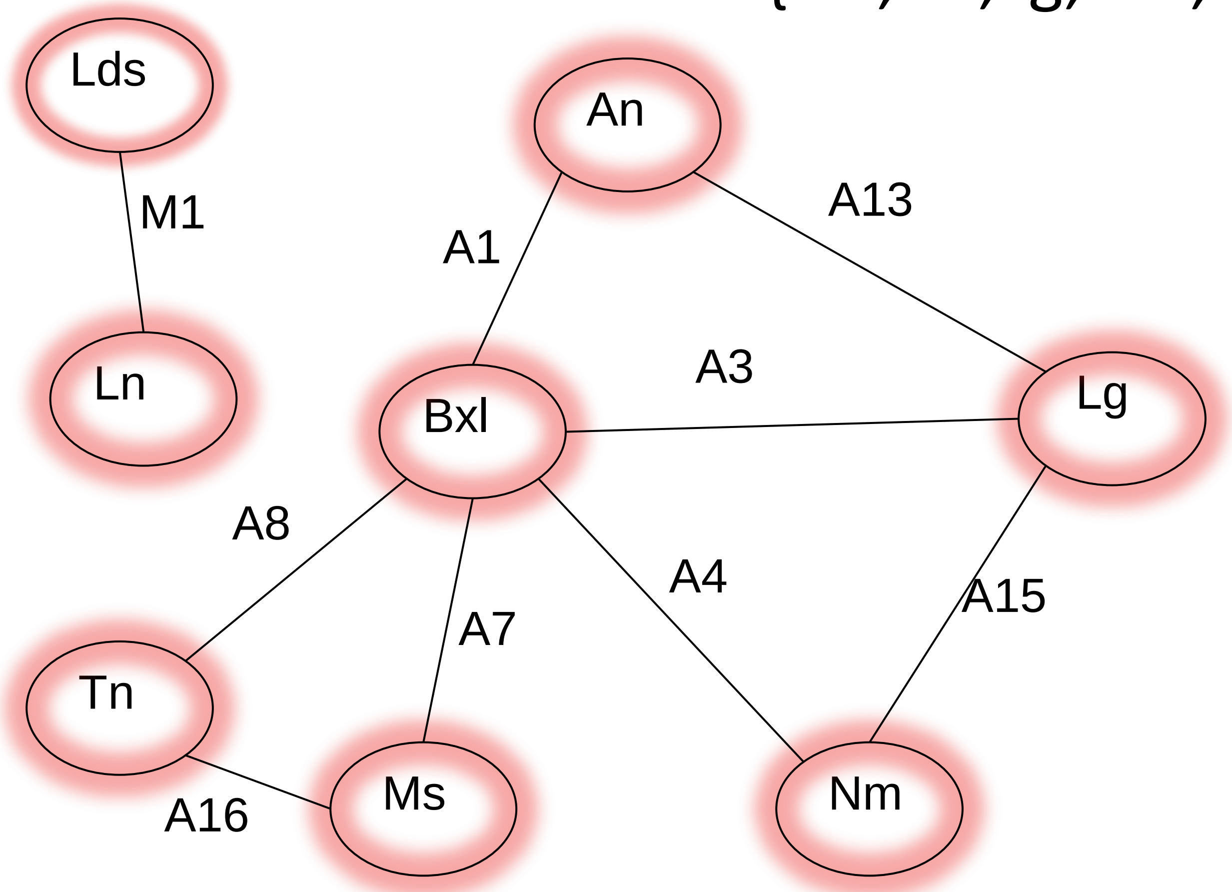


Les sommets

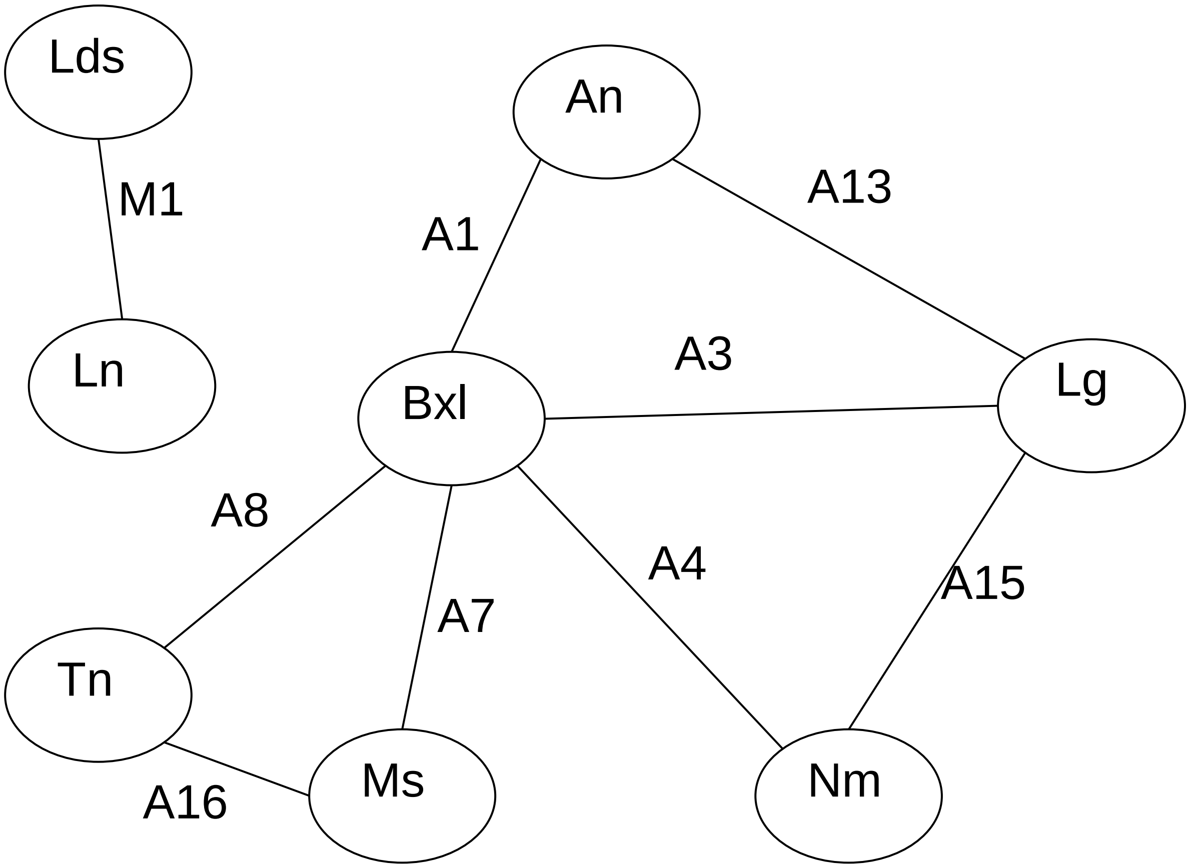


Les sommets

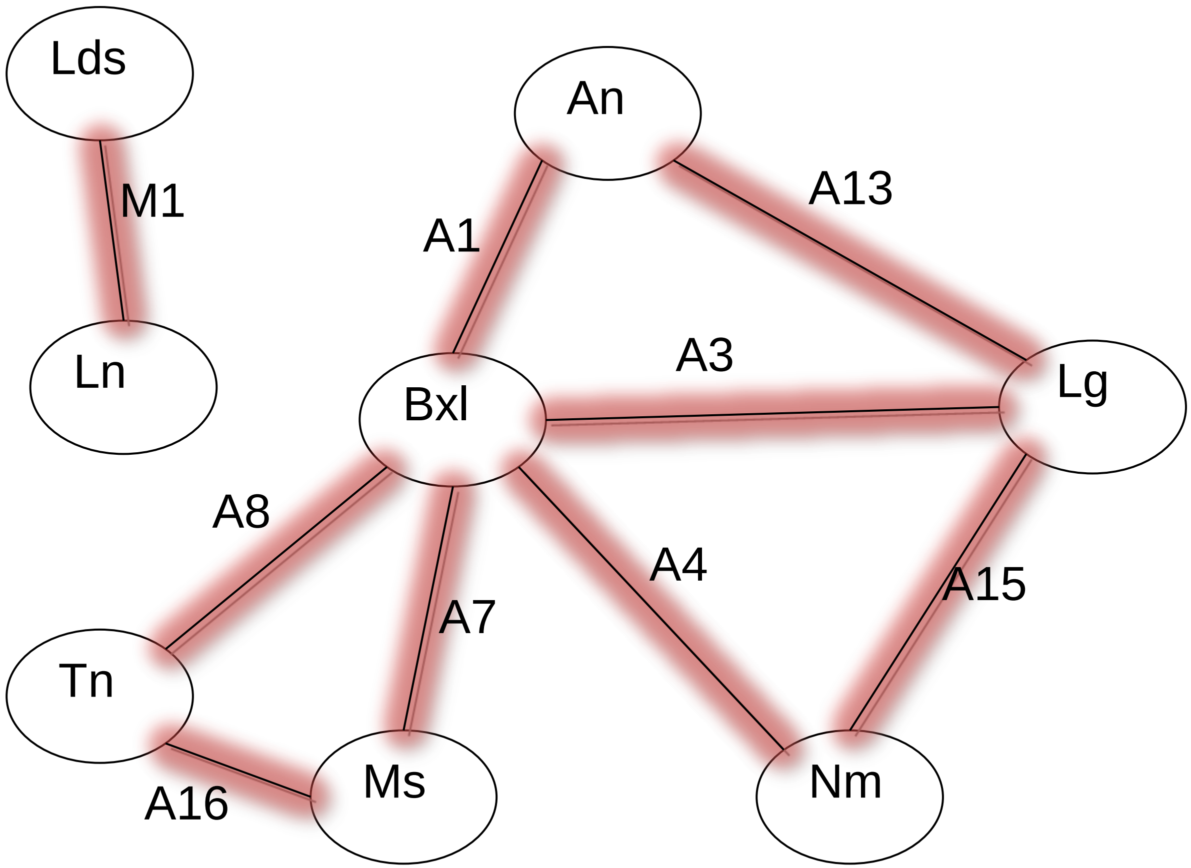
S ={Bxl,An,Lg,Nm,Ms,Tn,Ln,Lds}



Les arcs

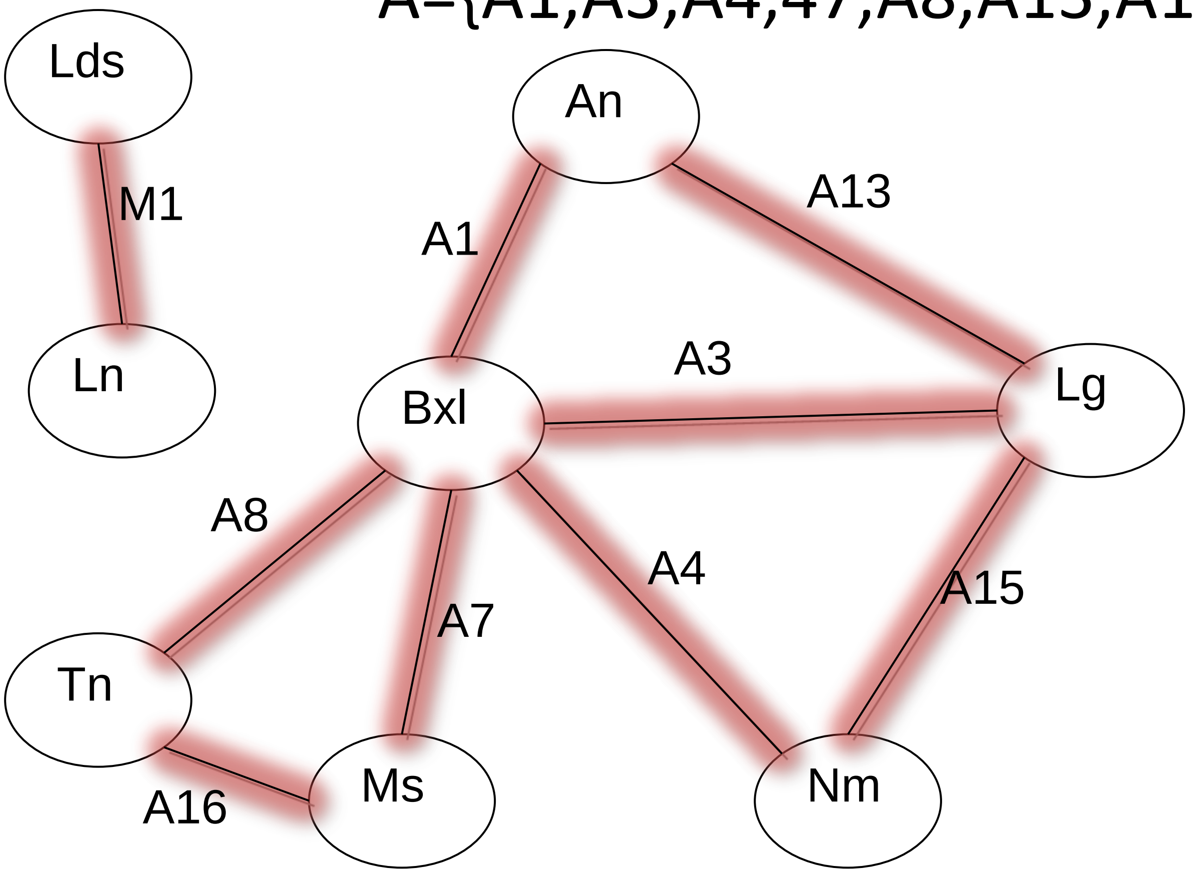


Les arcs

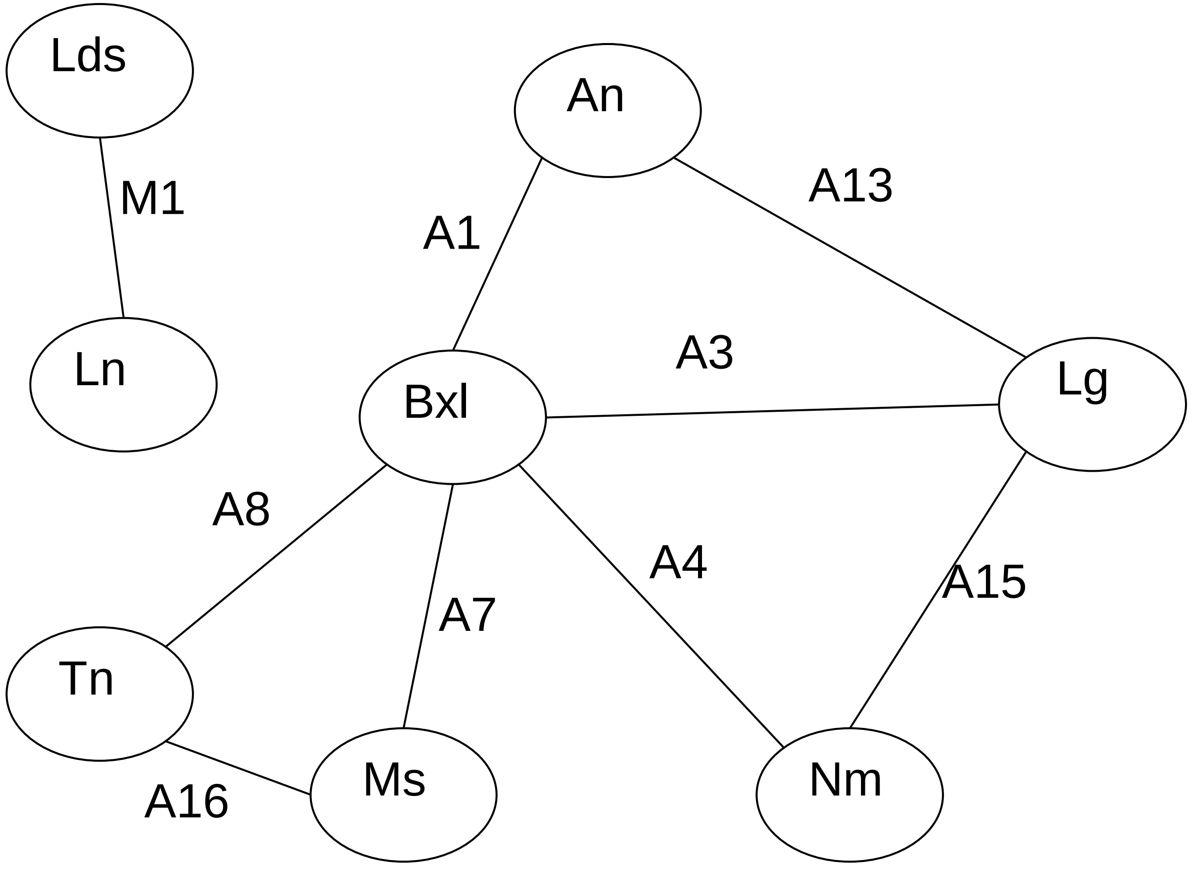


Les arcs

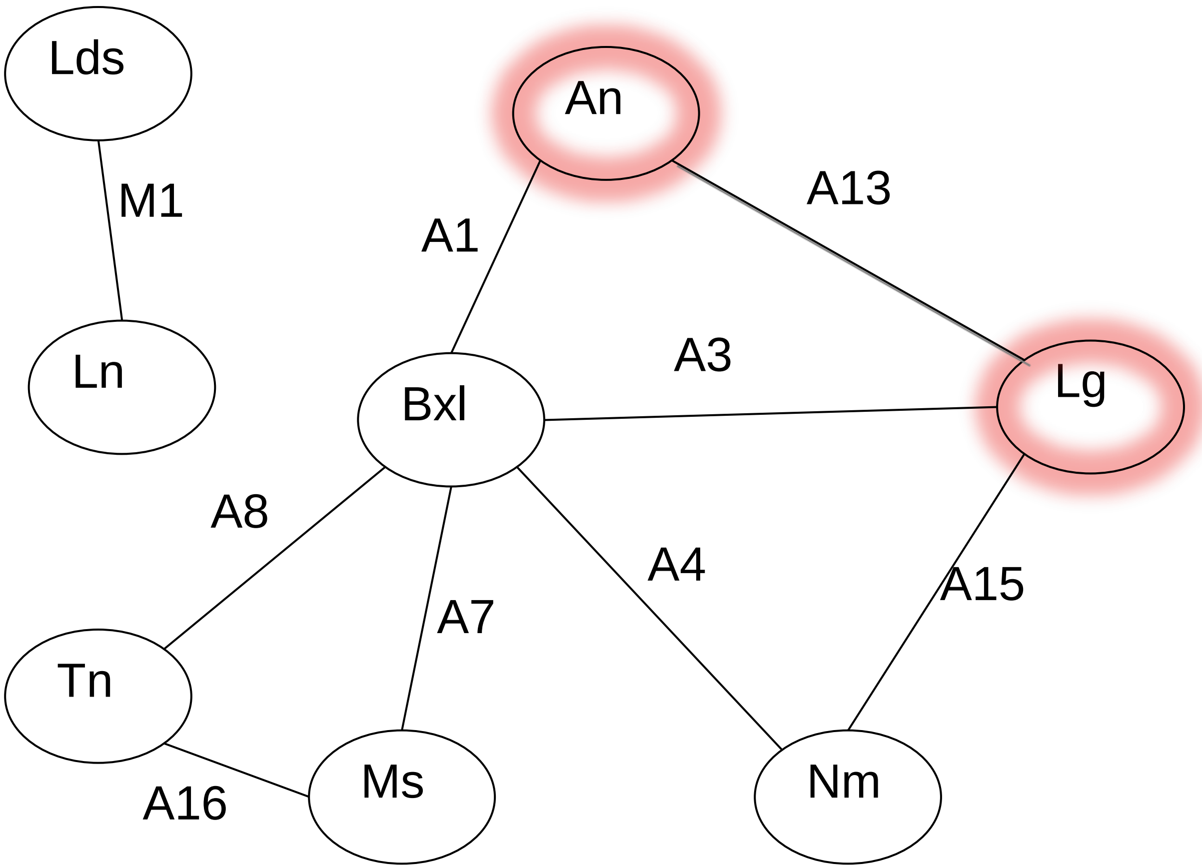
A={A1,A3,A4,47,A8,A13,A15,A16,M1}



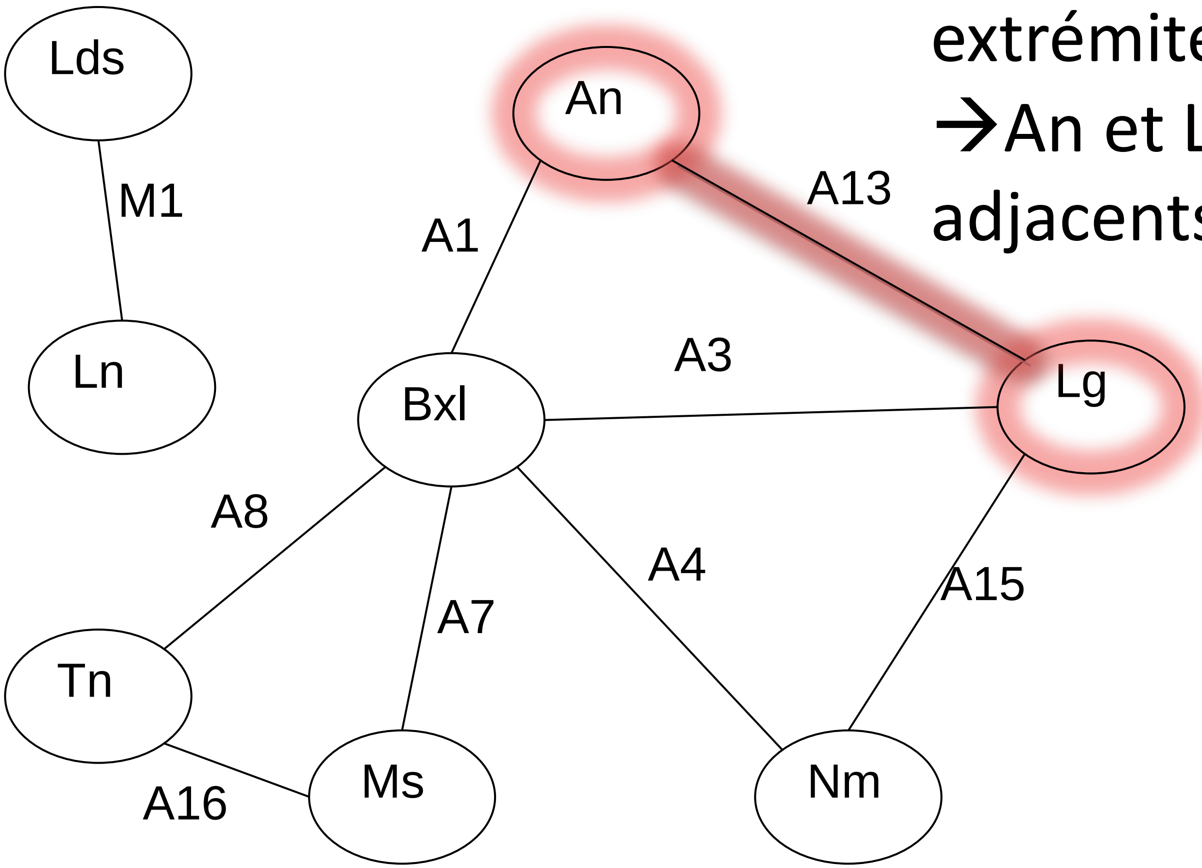
Sommets adjacents



Sommets adjacents

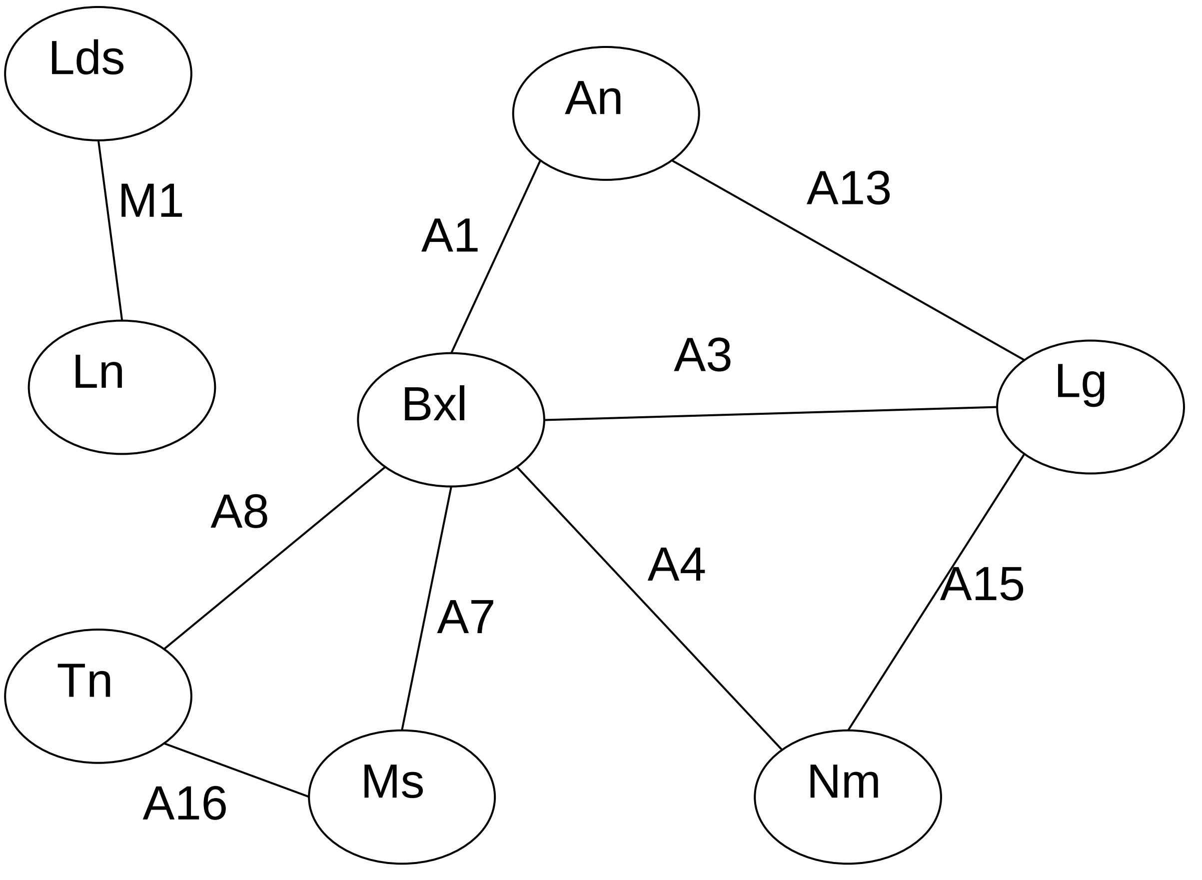


Sommets adjacents

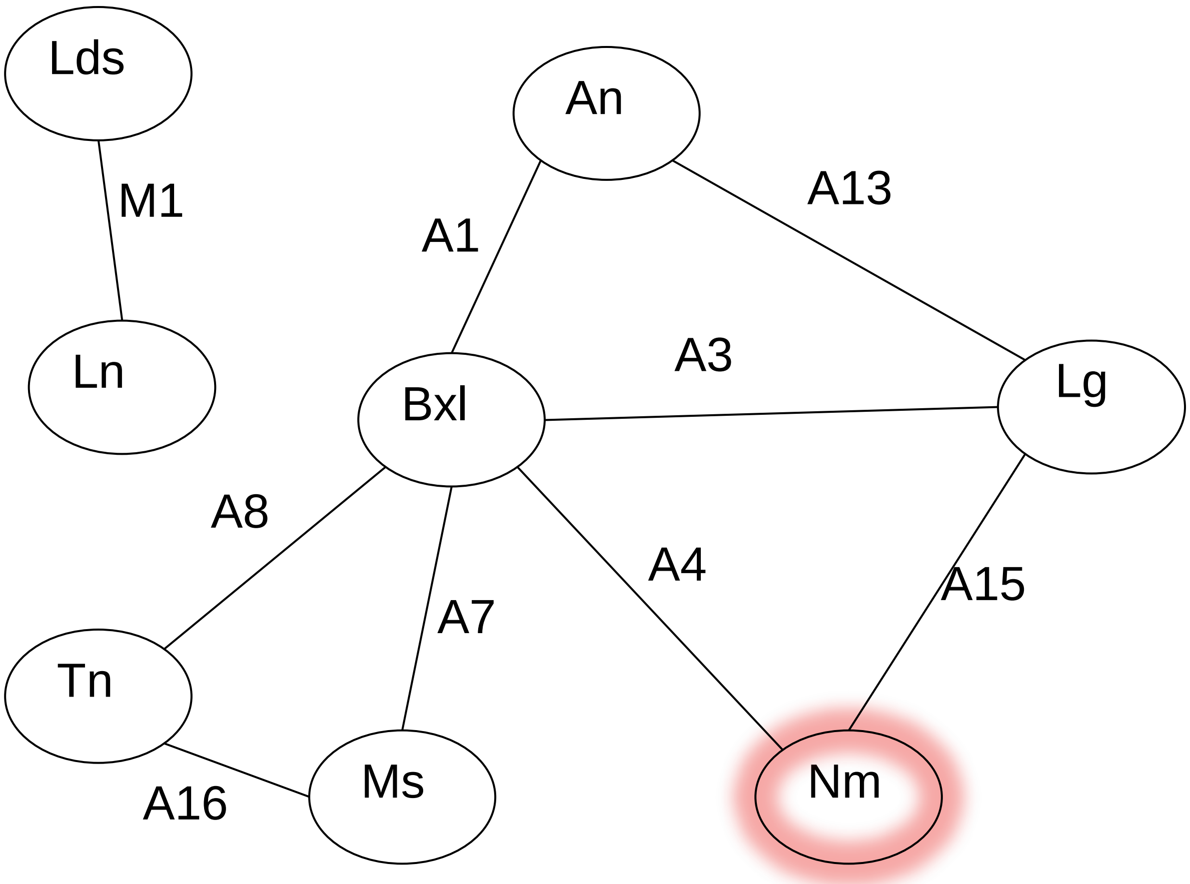


An et Lg sont les extrémités de l'arc A13
→An et Lg sont adjacents

Arcs incidents

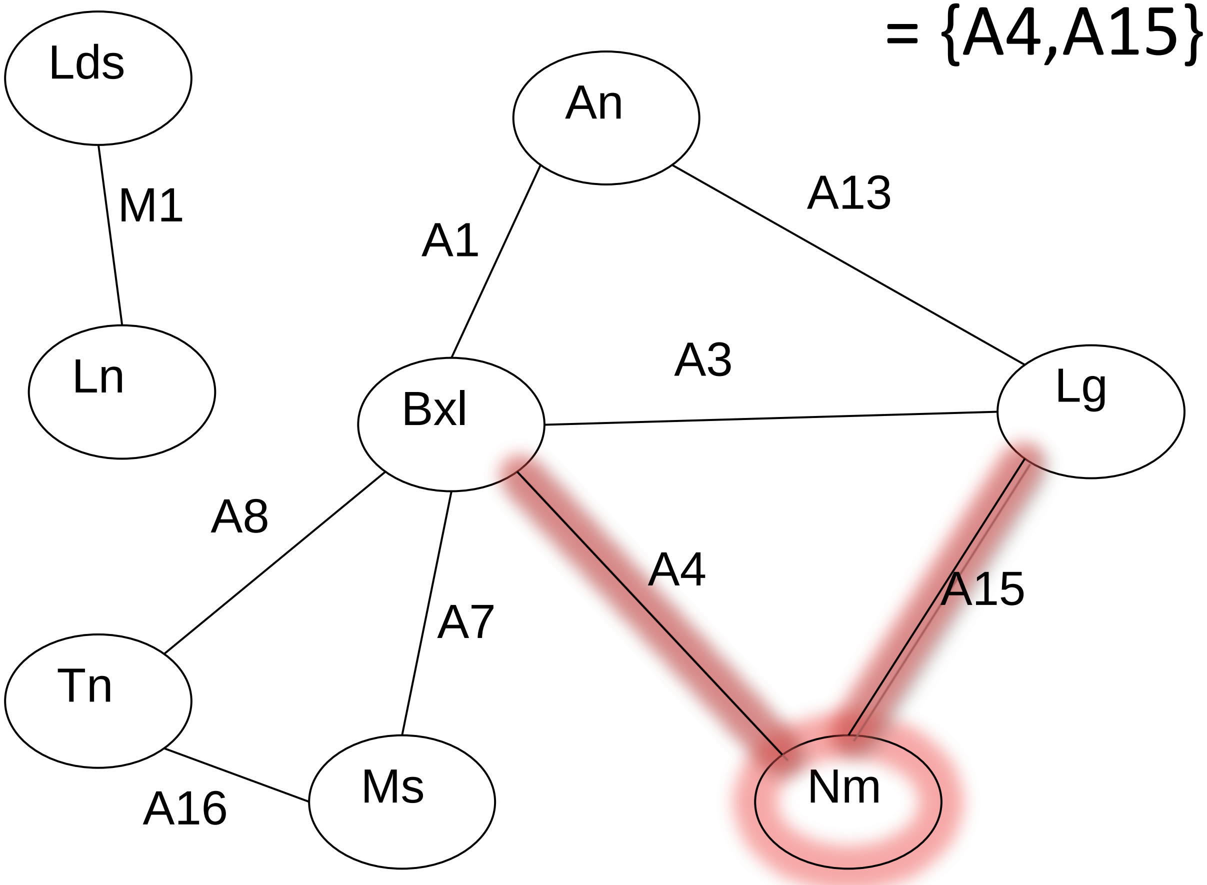


Arcs incidents

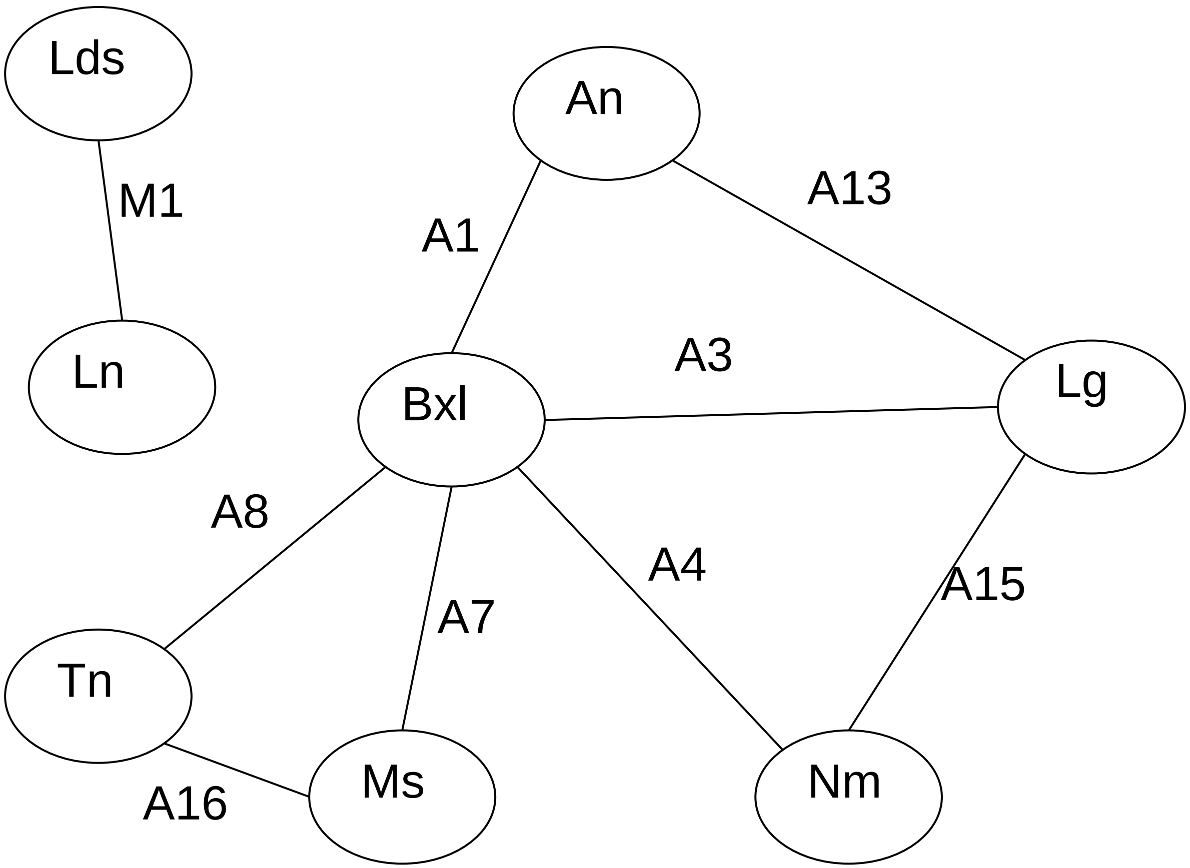


Arcs incidents

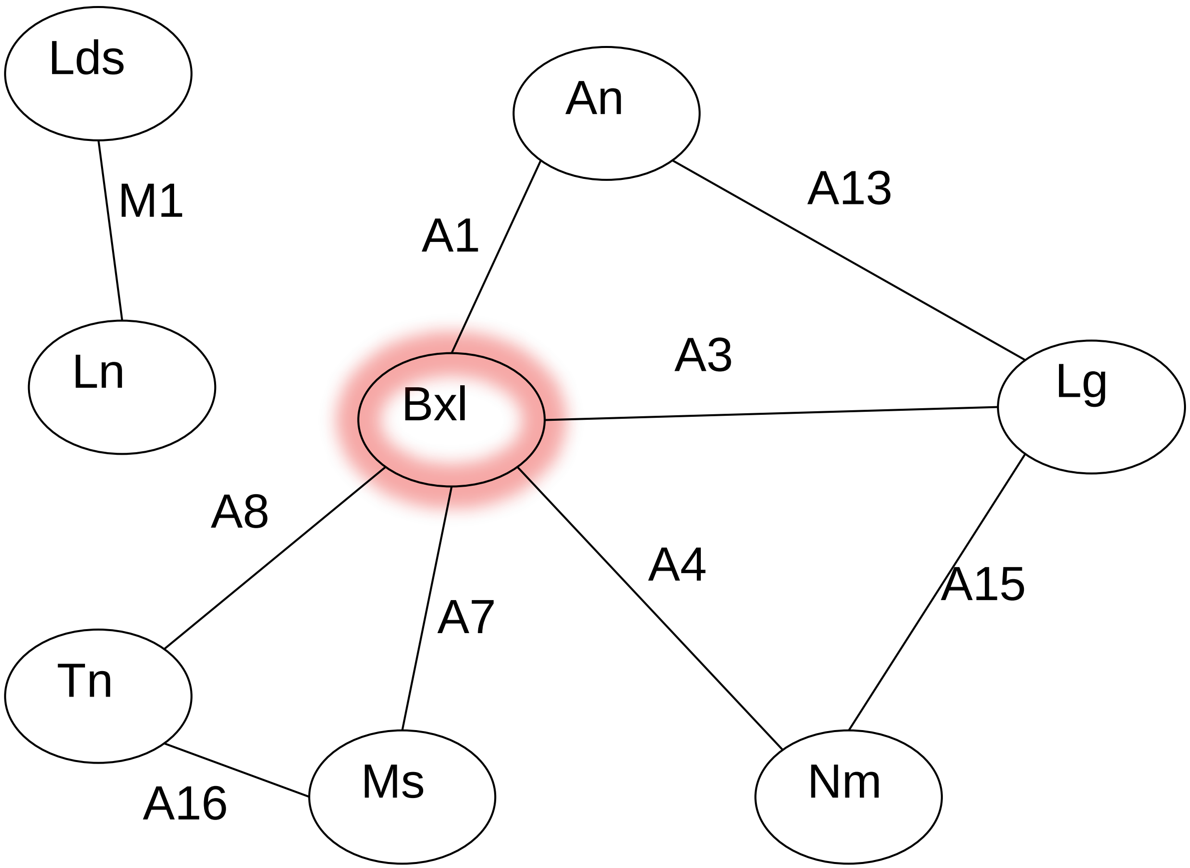
Arcs incidents de Nm
= {A4,A15}



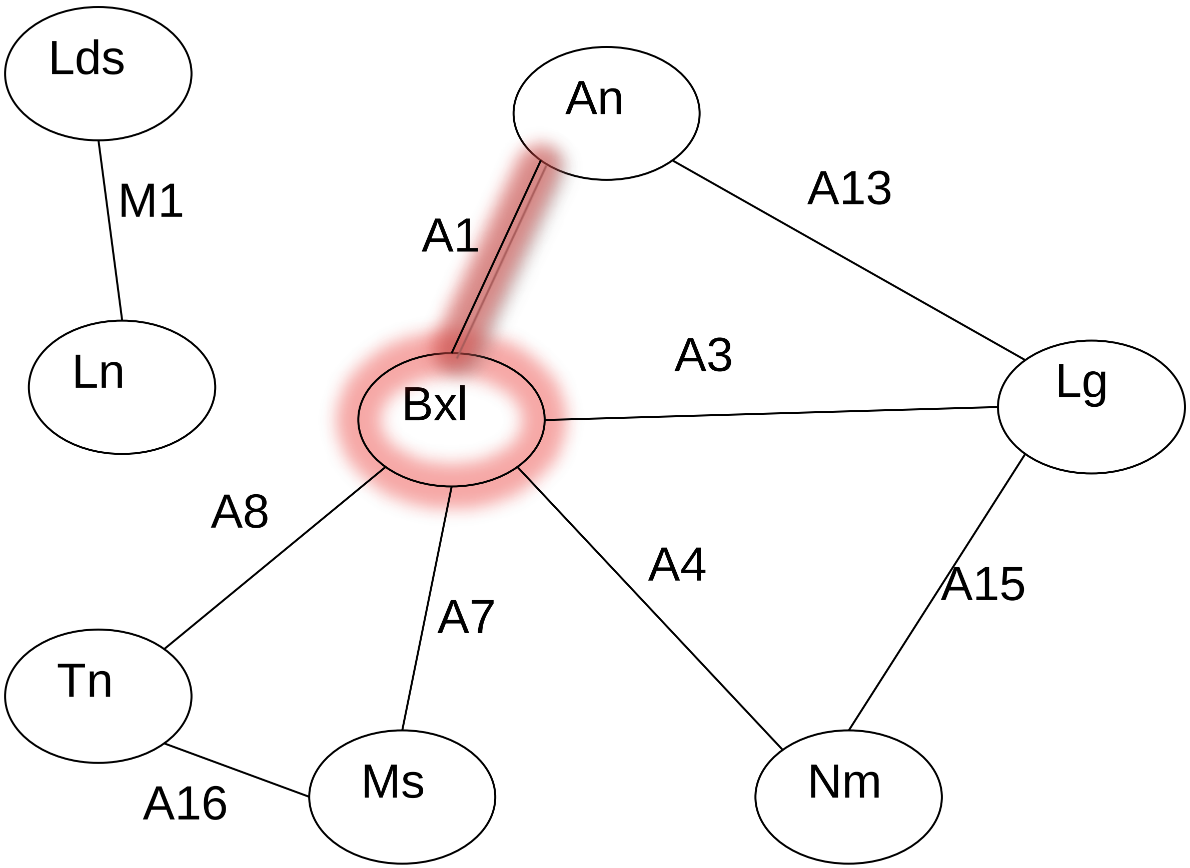
Degré d'un sommet



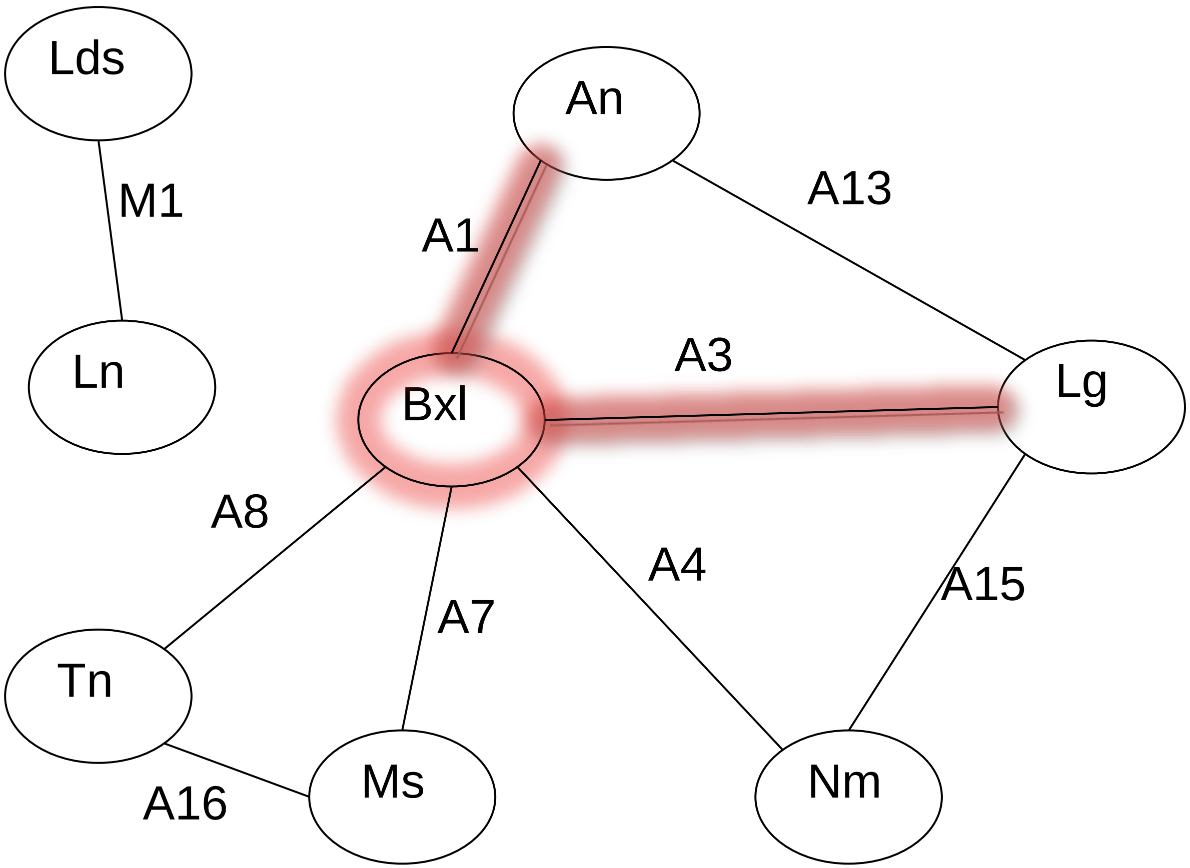
Degré d'un sommet



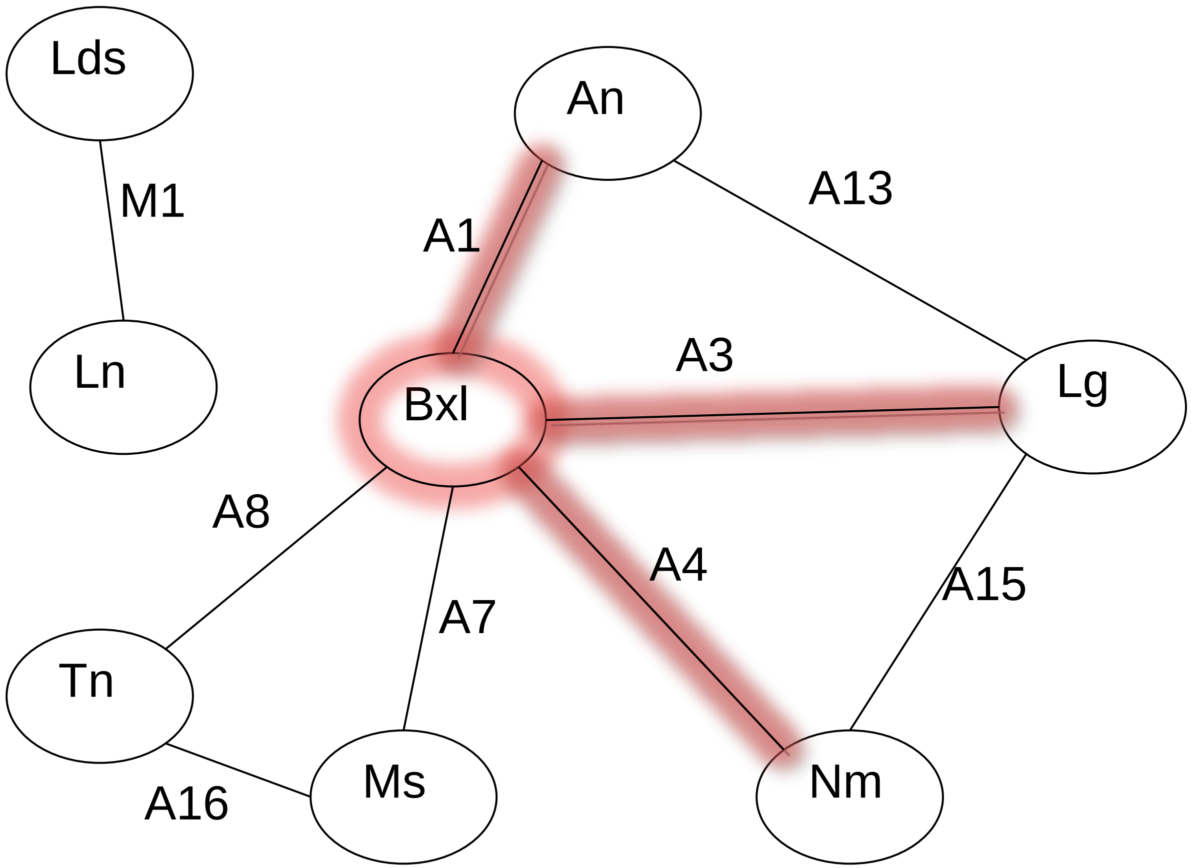
Degré d'un sommet



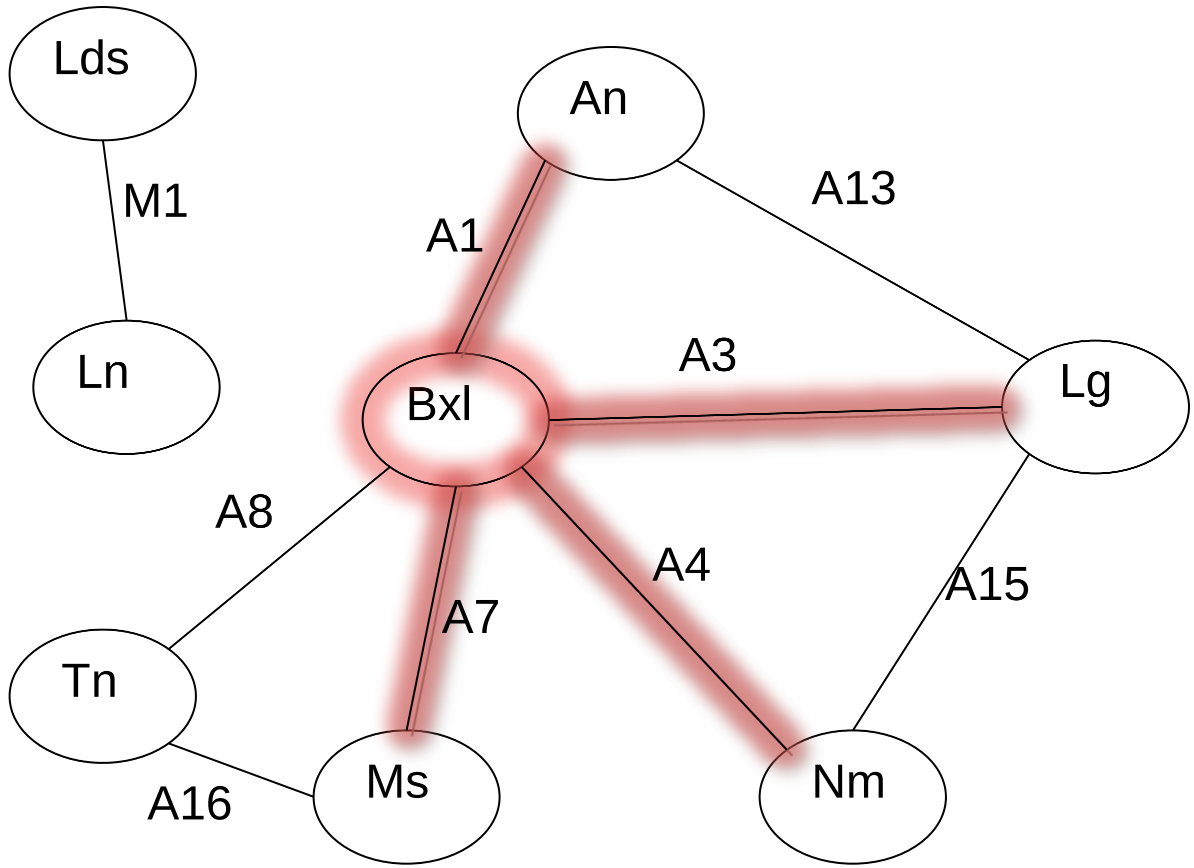
Degré d’un sommet



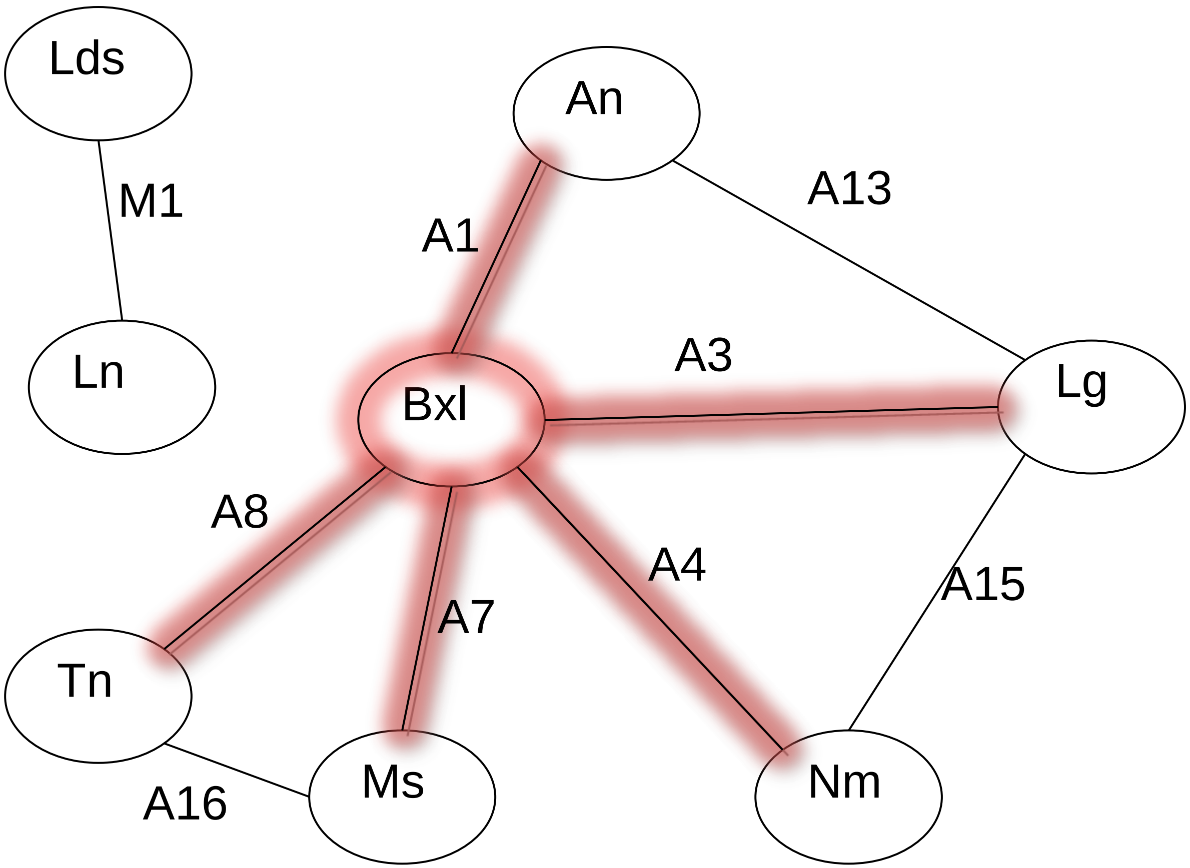
Degré d’un sommet



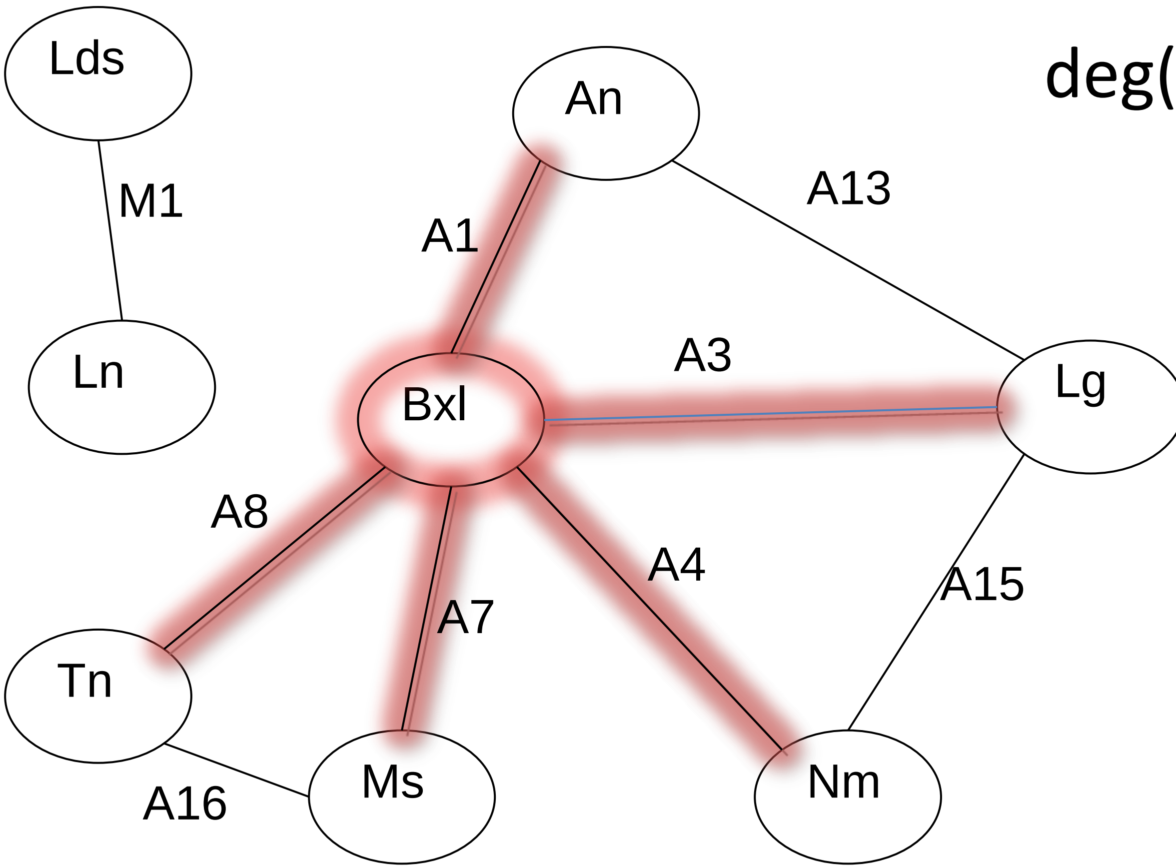
Degré d'un sommet



Degré d'un sommet

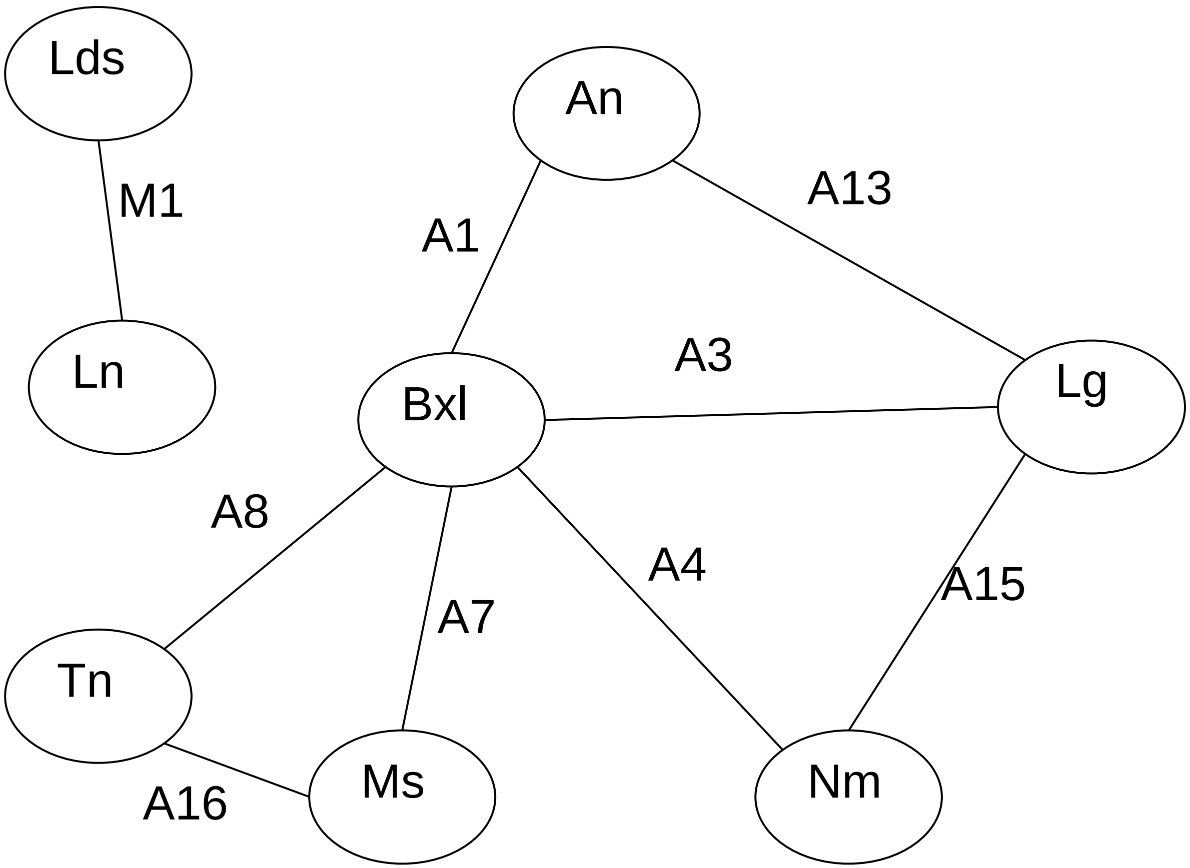


Degré d'un sommet



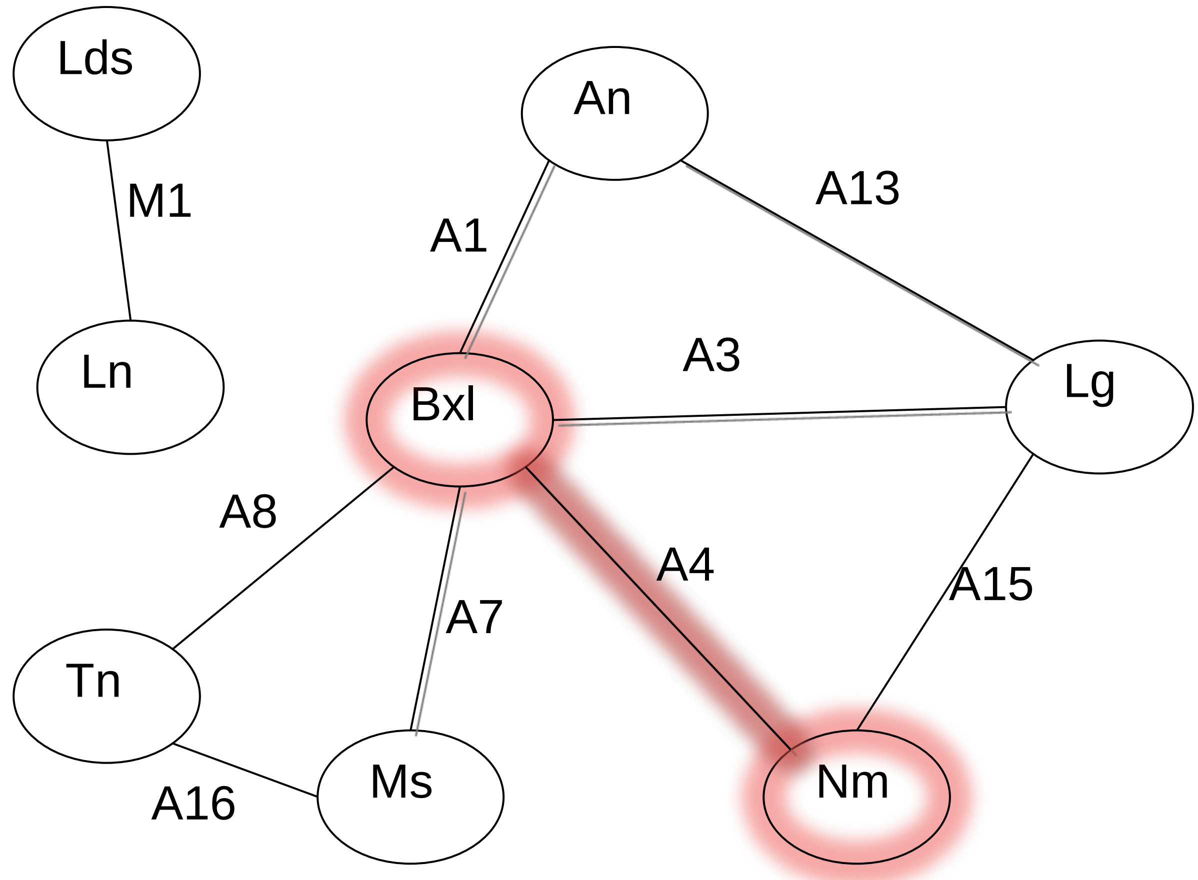
deg(Bxl) = 5

Chemin



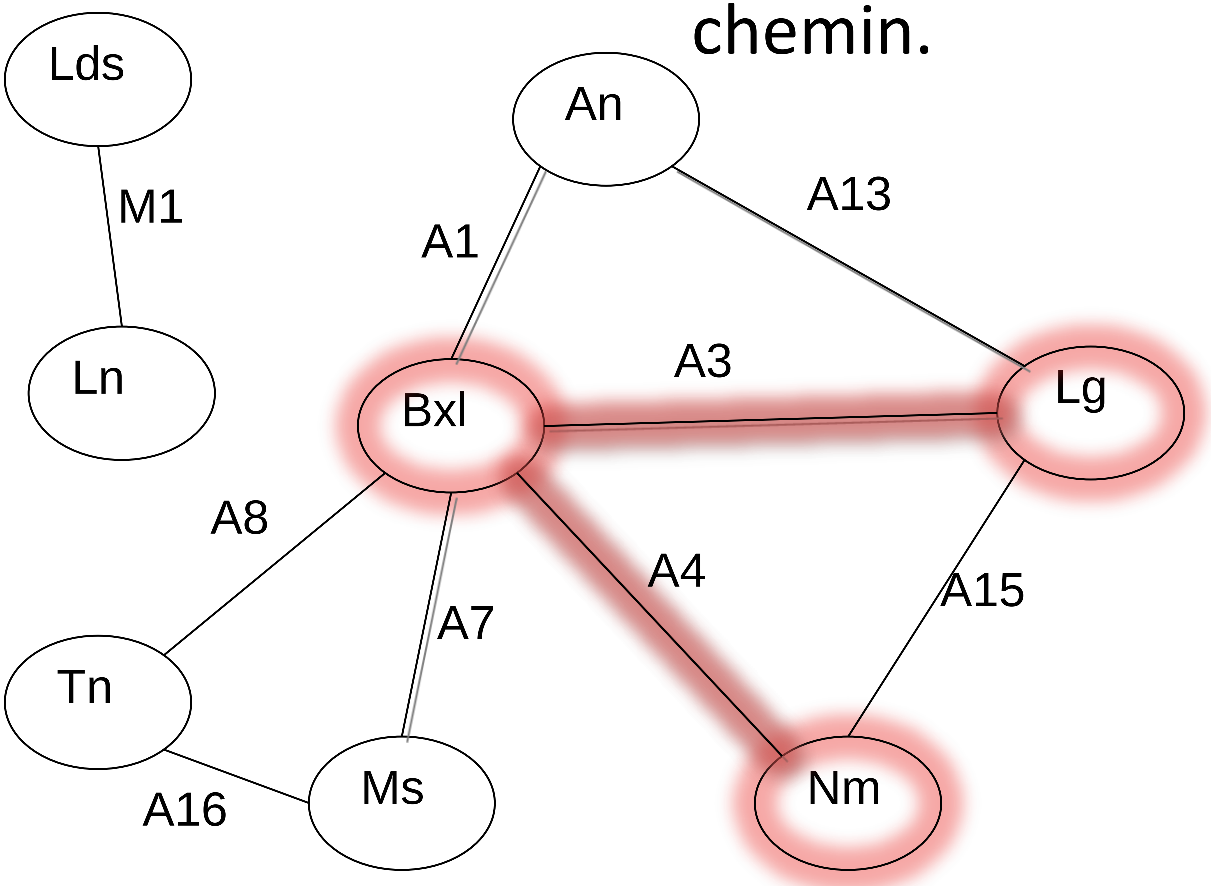
Chemin

(Nm,A4,Bxl) est un chemin.



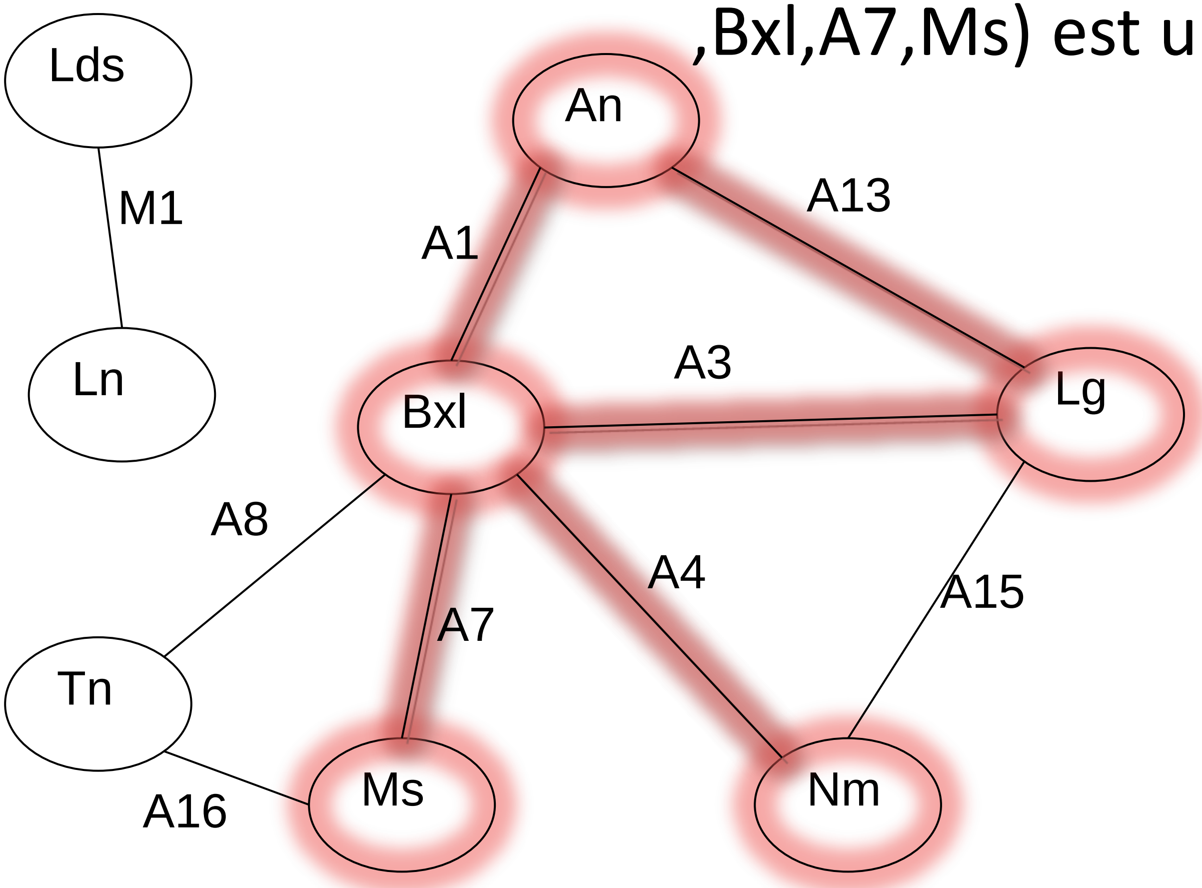
Chemin

(Nm,A4,Bxl,A3,Lg) est un chemin.



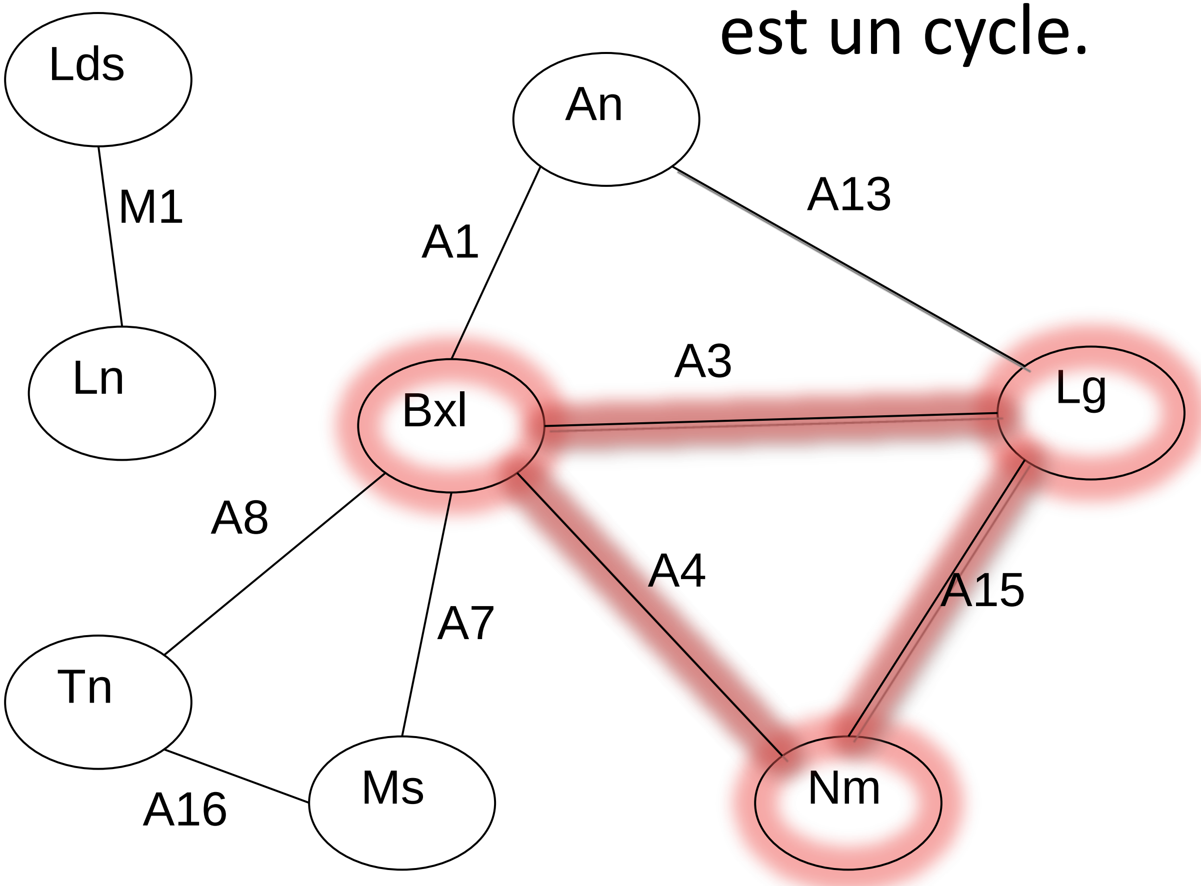
Chemin

(Nm,A4,Bxl,A3,Lg,A13,An,A1,Bxl,A7,Ms) est un chemin.

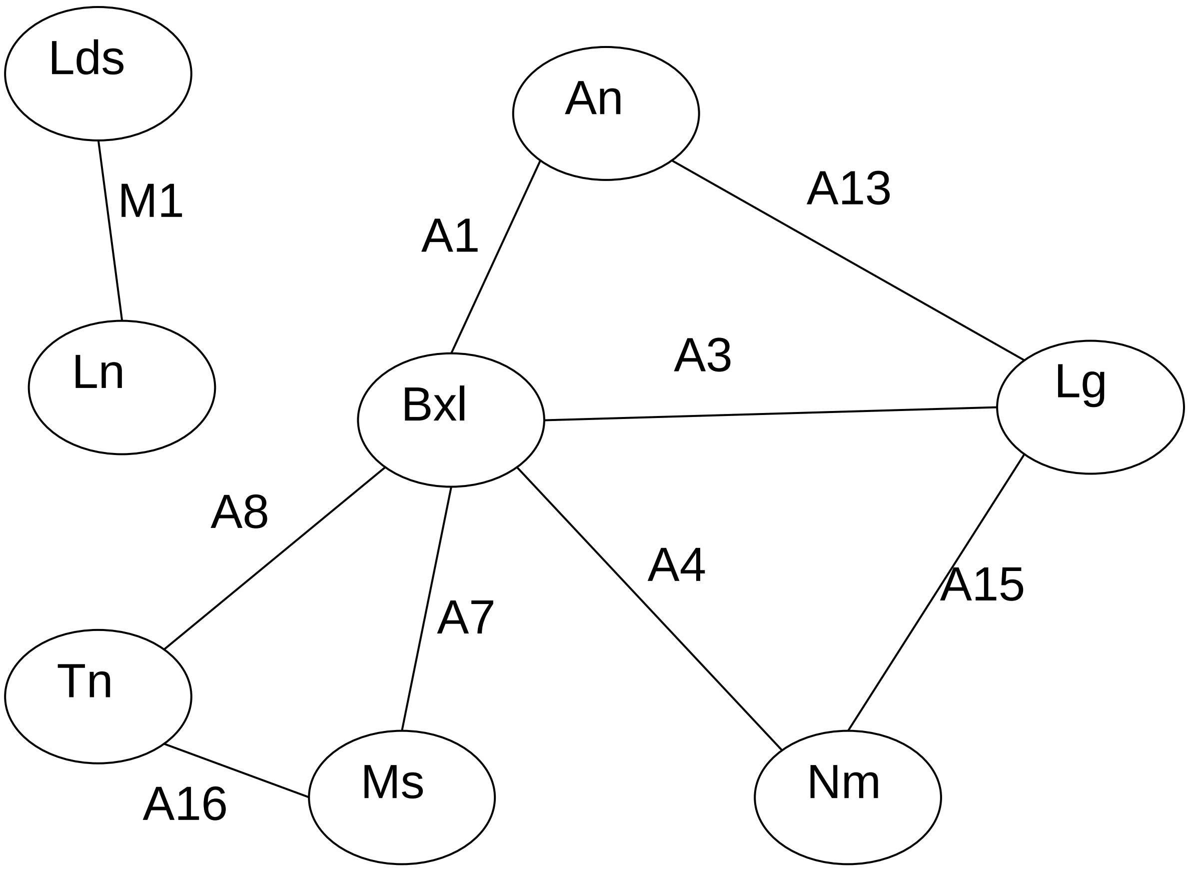


Chemin

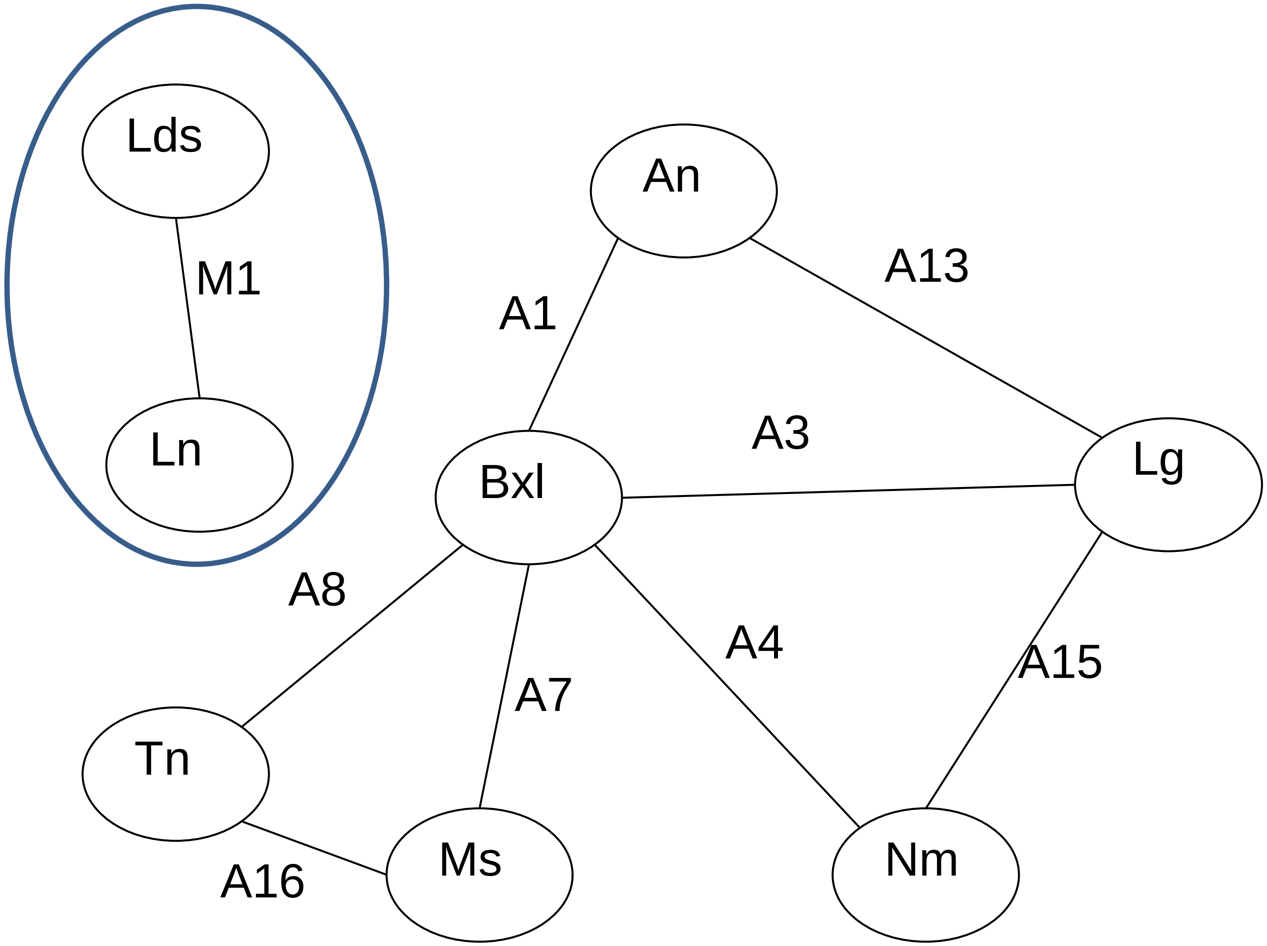
(Nm,A4,Bxl,A3,Lg,A15,Nm)
est un cycle.



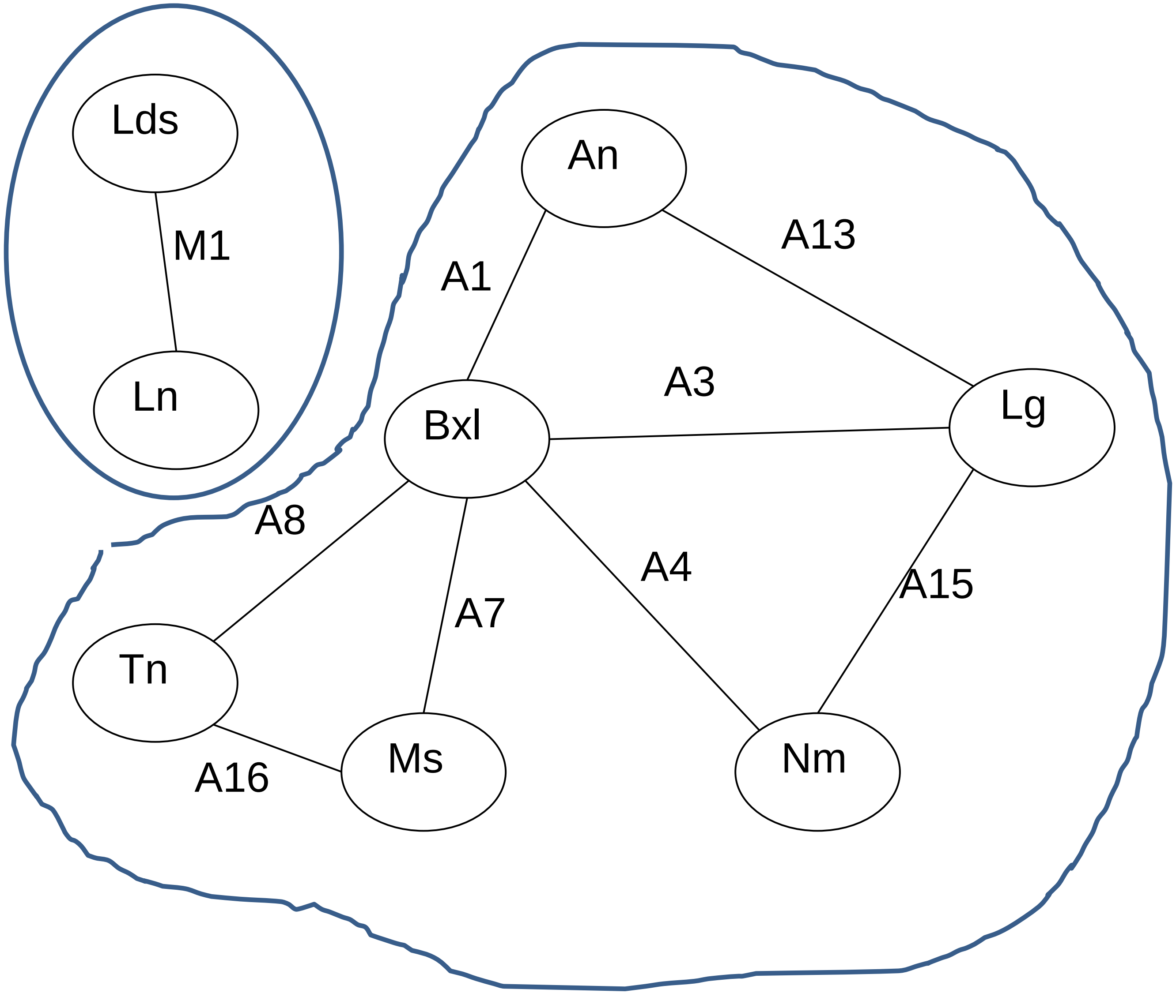
Composantes connexes



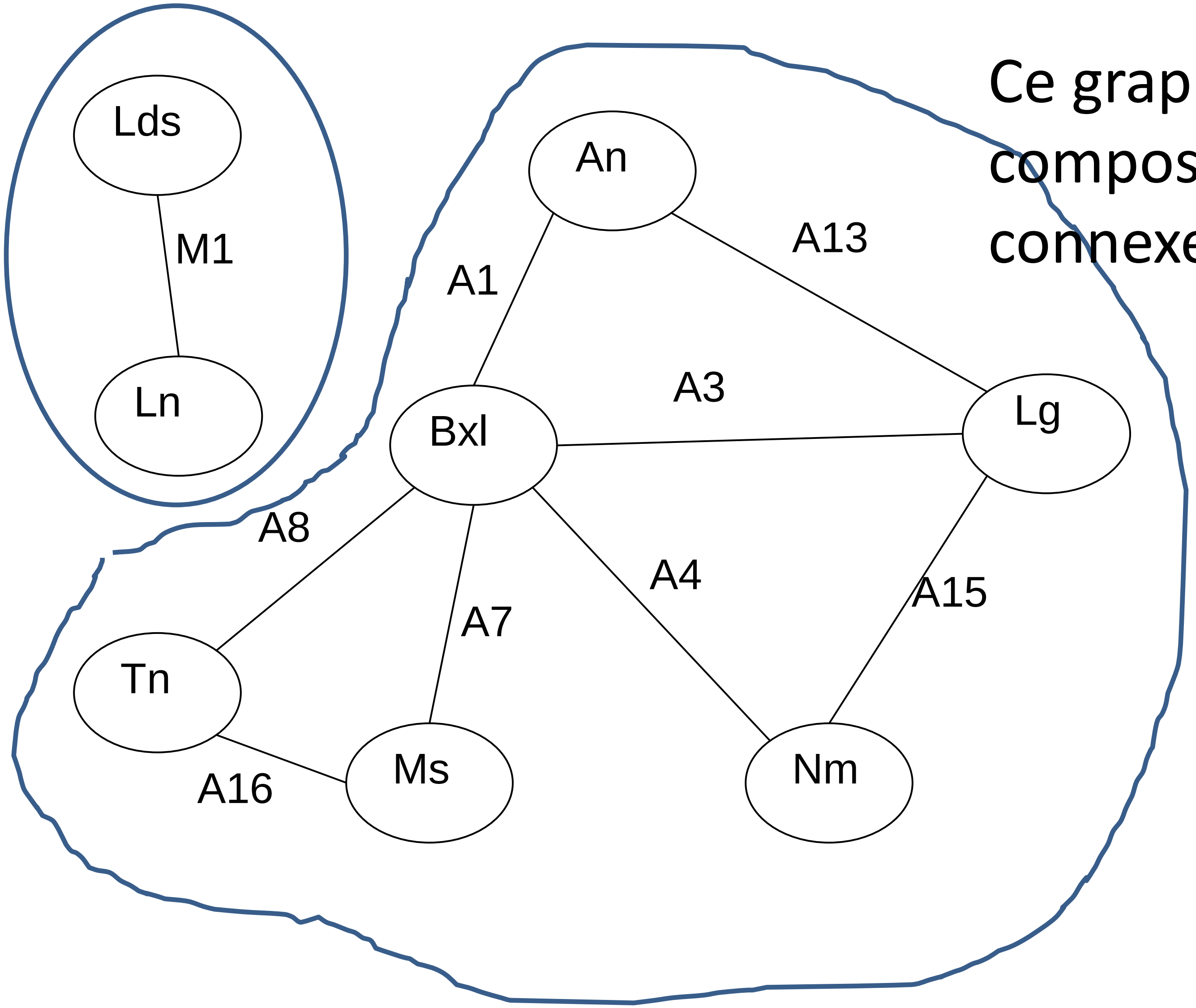
Composantes connexes



Composantes connexes

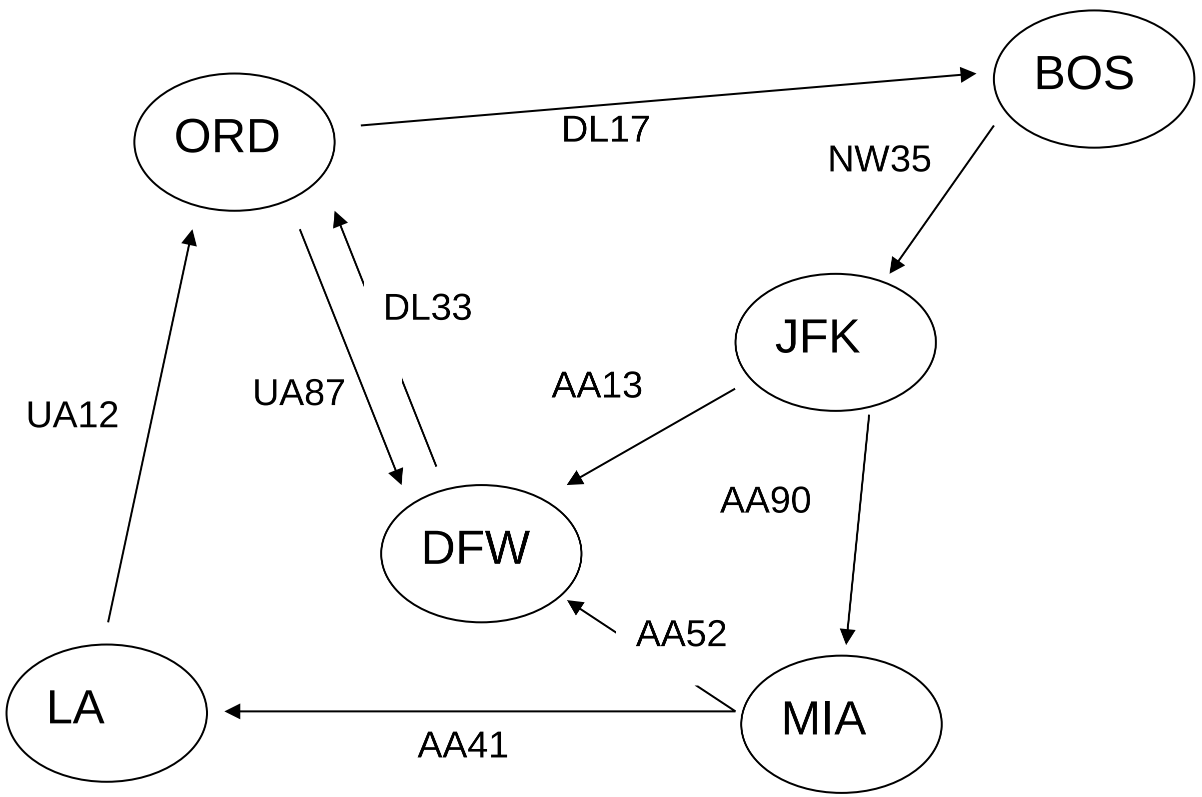


Composantes connexes

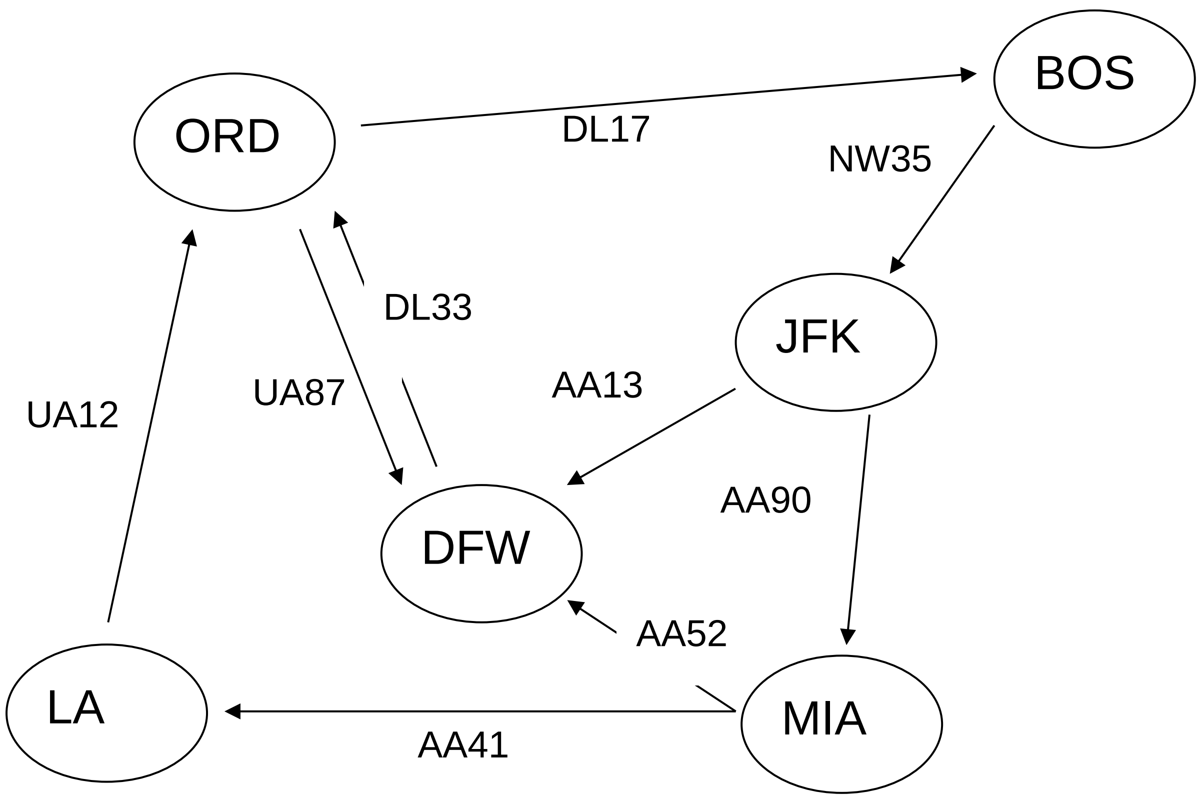


Ce graphe a deux composantes connexes.

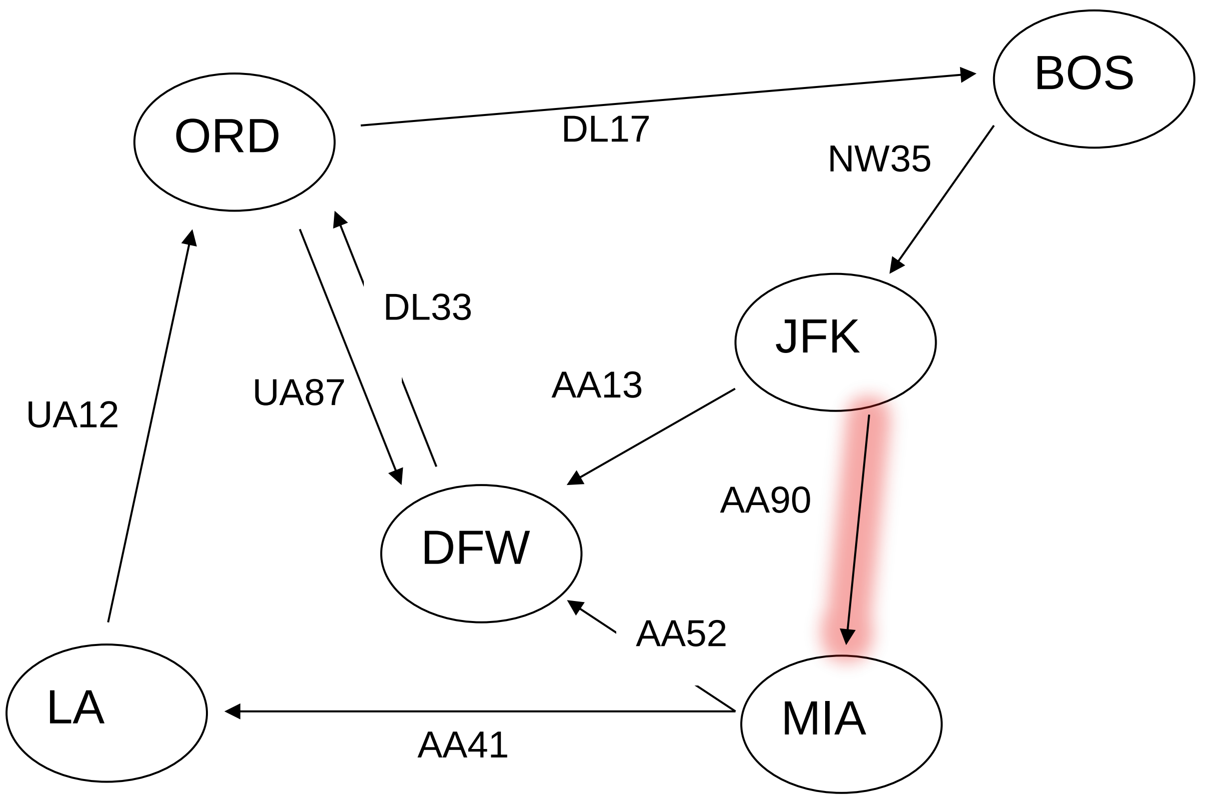
Exemple 2 : graphe dirigé



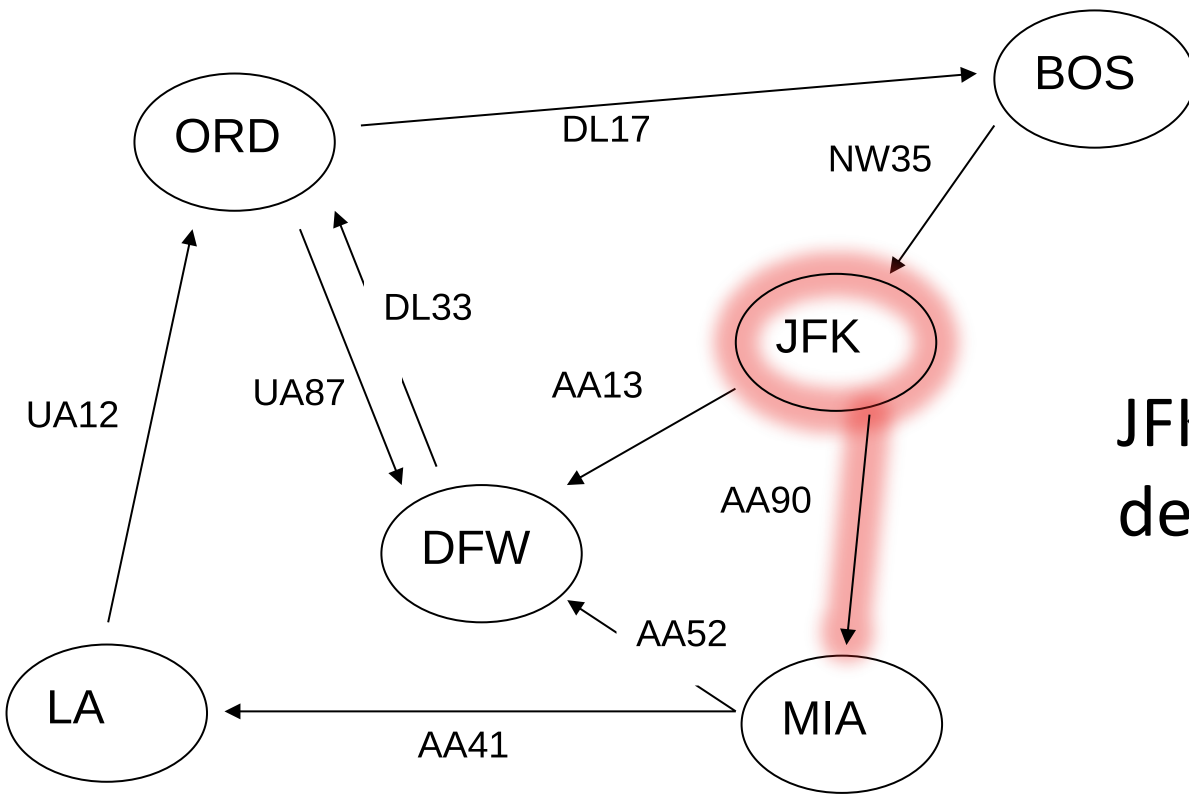
Origine


BOS
ORD
DL17
NW35
DL33
JFK
UA87
AA13
UA12
AA90
DFW
AA52
LA
MIA
AA41

Origine

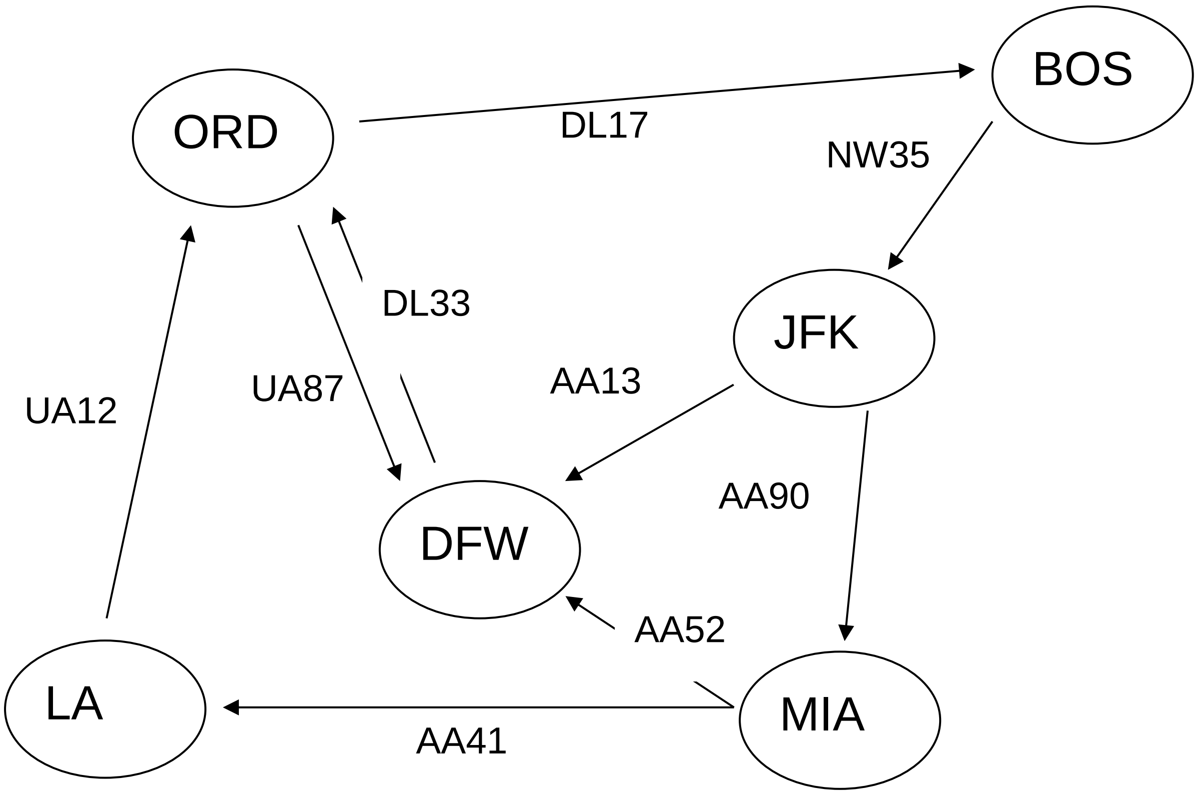

ORD
BOS
DL17
NW35
DL33
JFK
UA87
AA13
UA12
AA90
DFW
AA52
LA
MIA
AA41

Origine

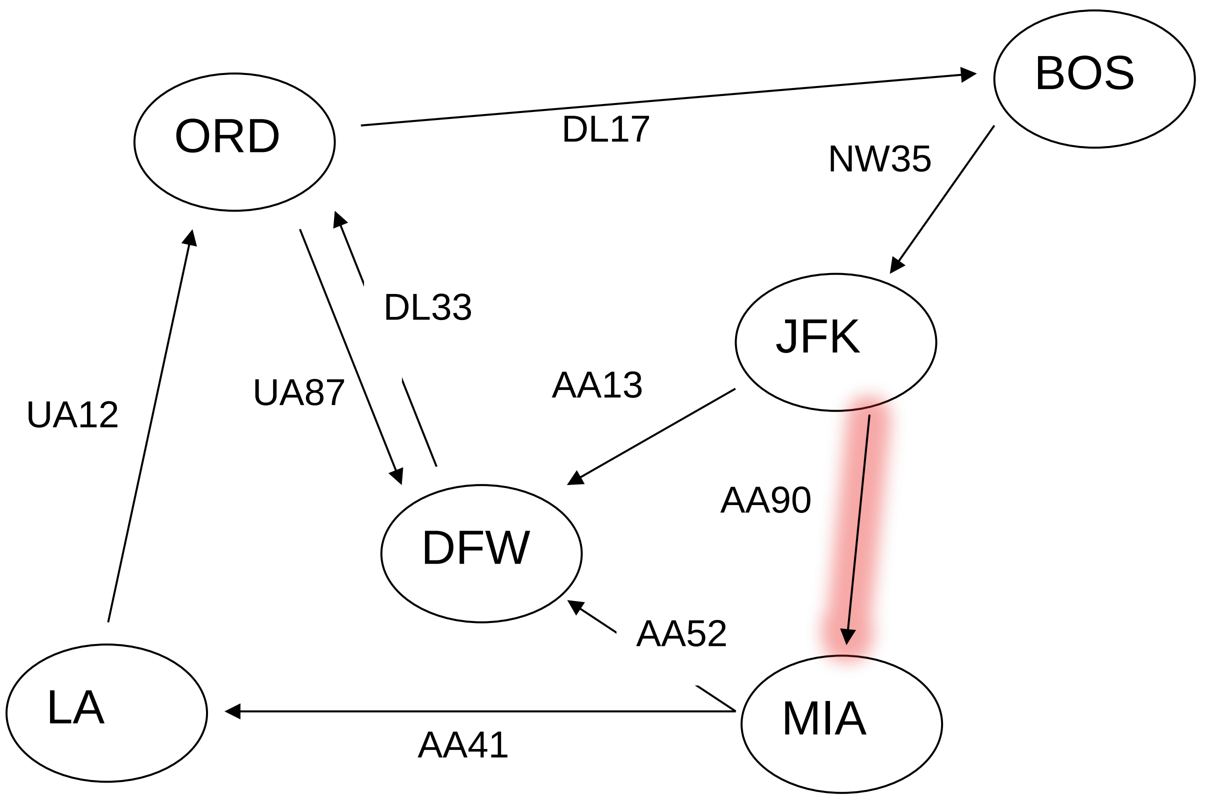


JFK est l'origine de l'arc AA90

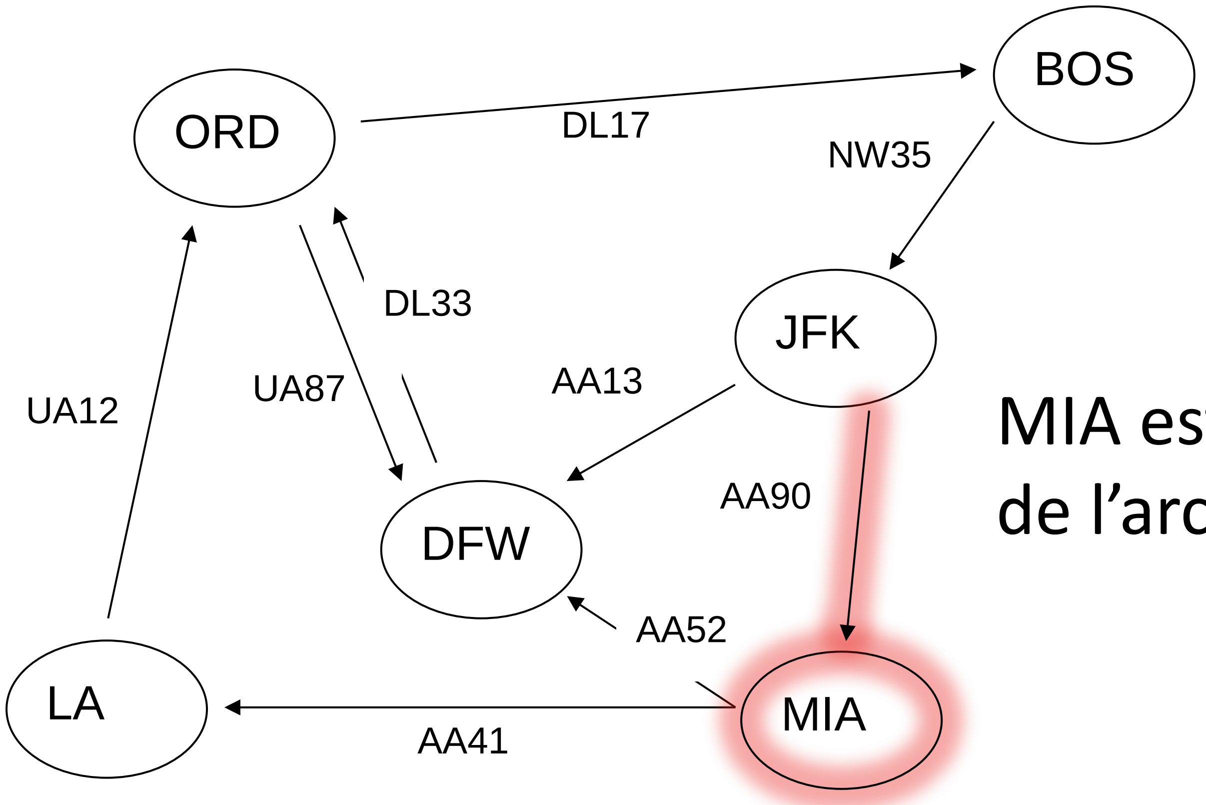
Destination



Destination

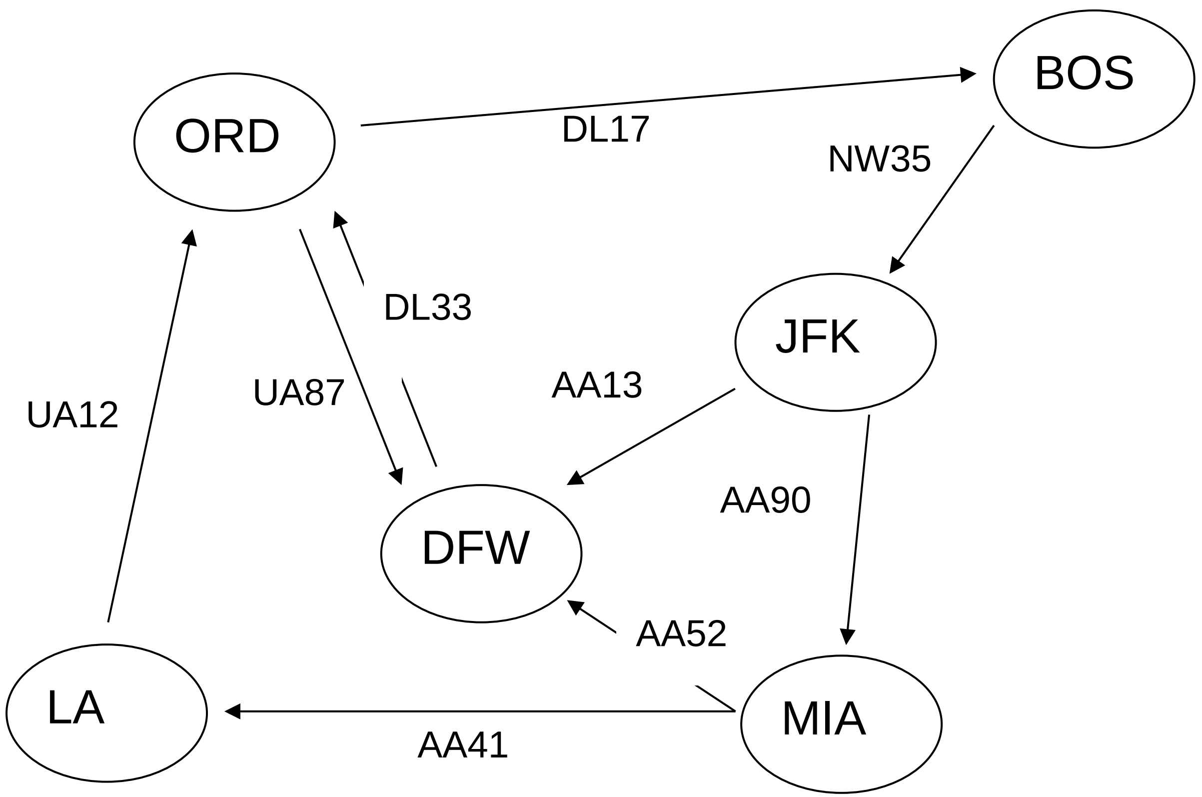


Destination

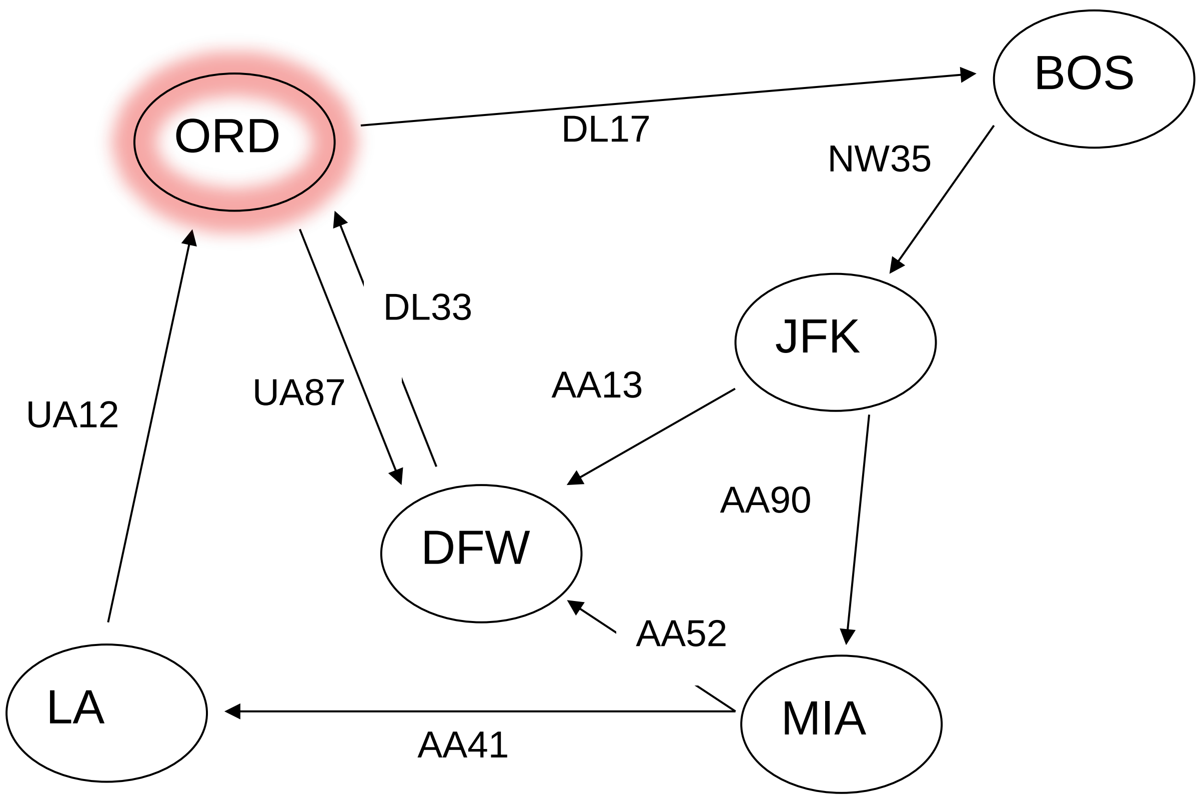


MIA est la destination de l’arc AA90

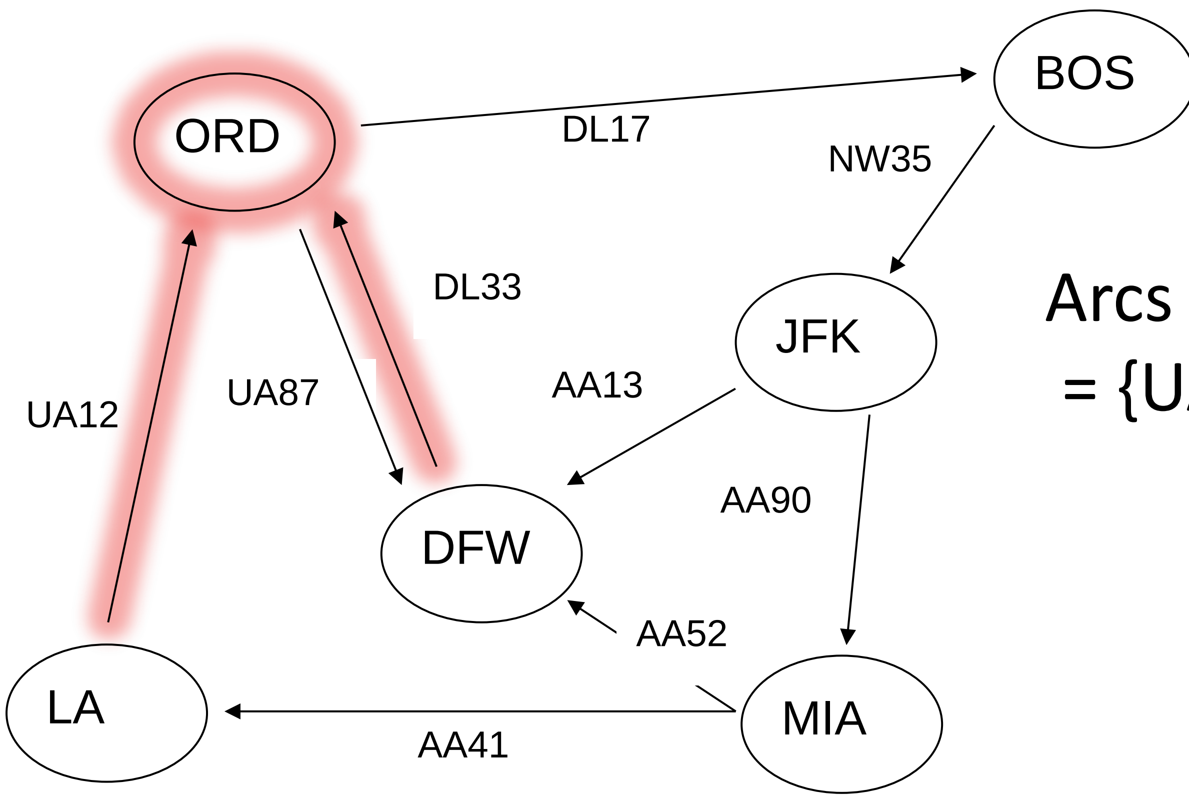
Arcs entrants



Arcs entrants

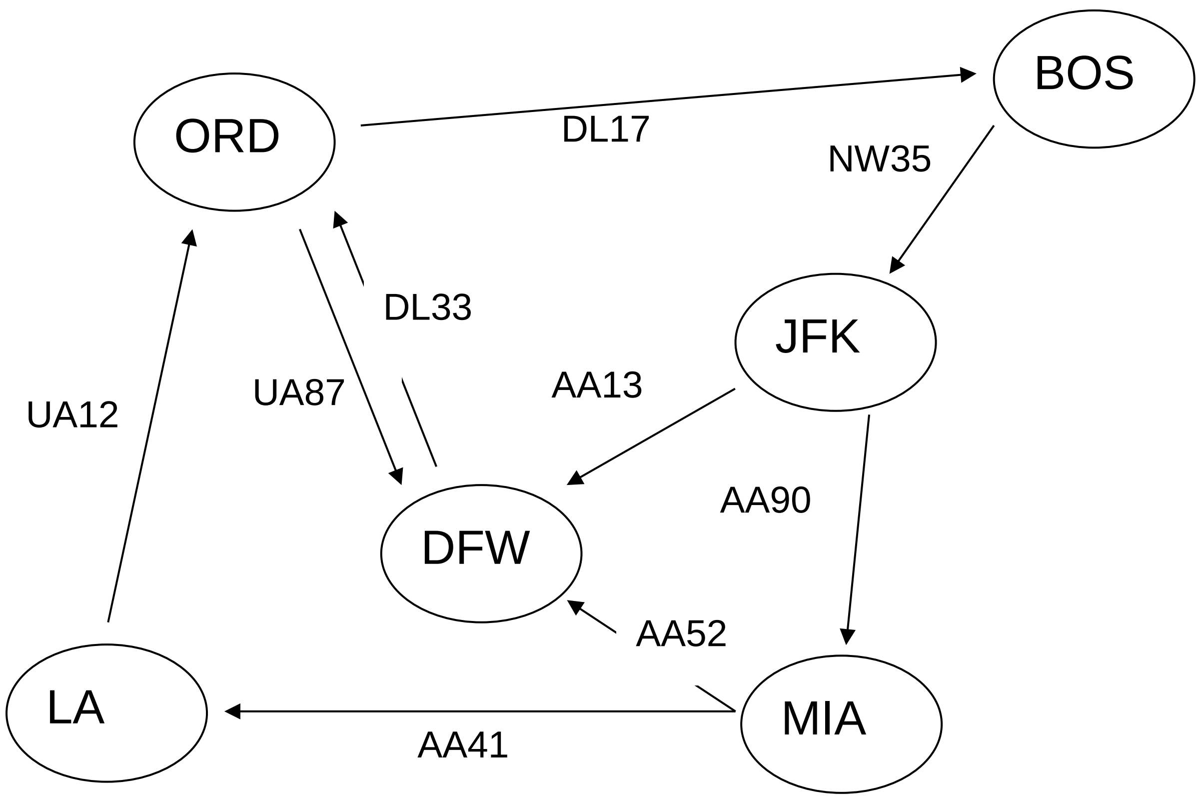


Arcs entrants

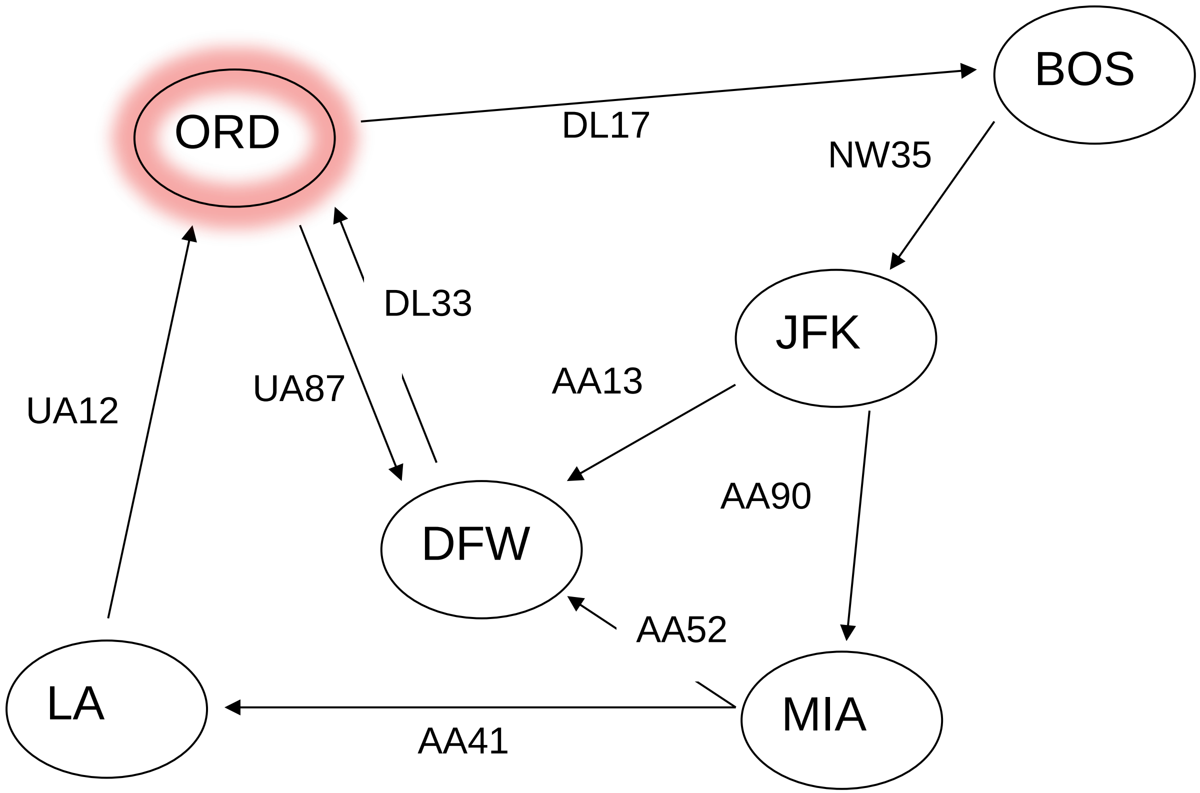


Arcs entrants de OR
= {UA12,DL33}

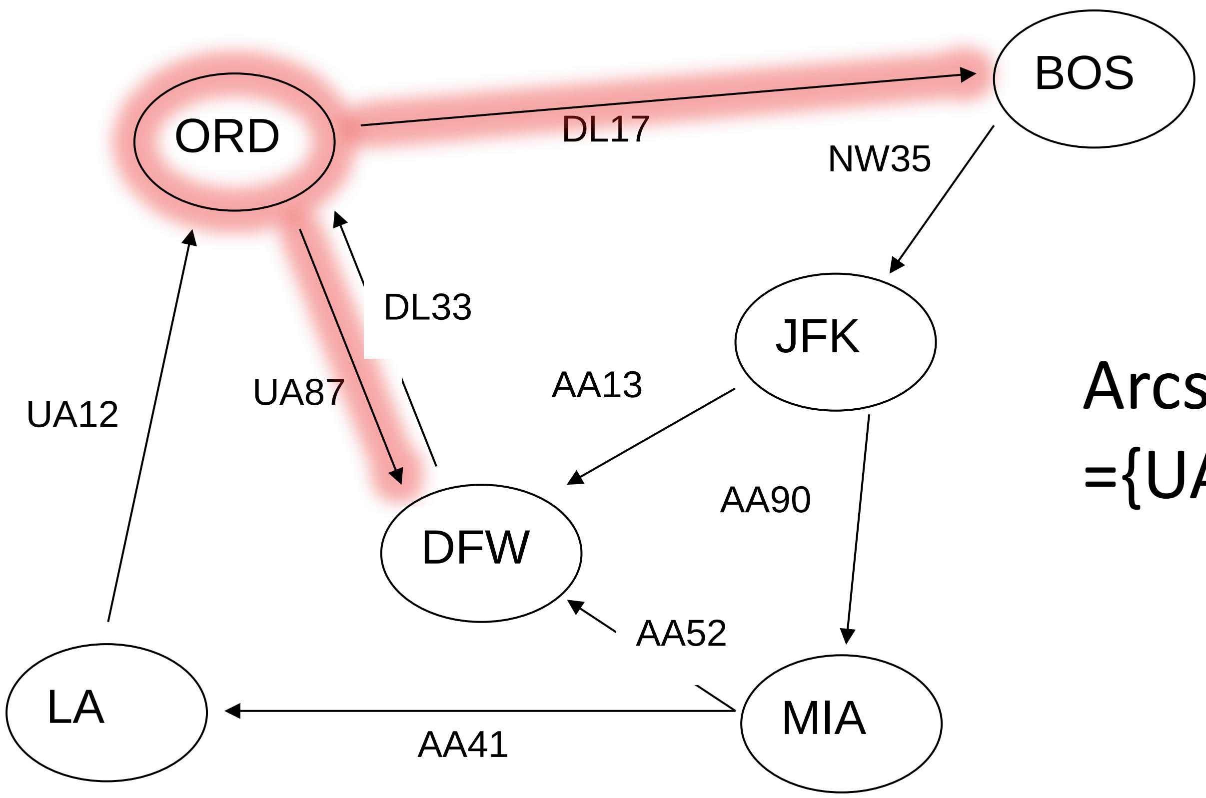
Arcs sortants



Arcs sortants

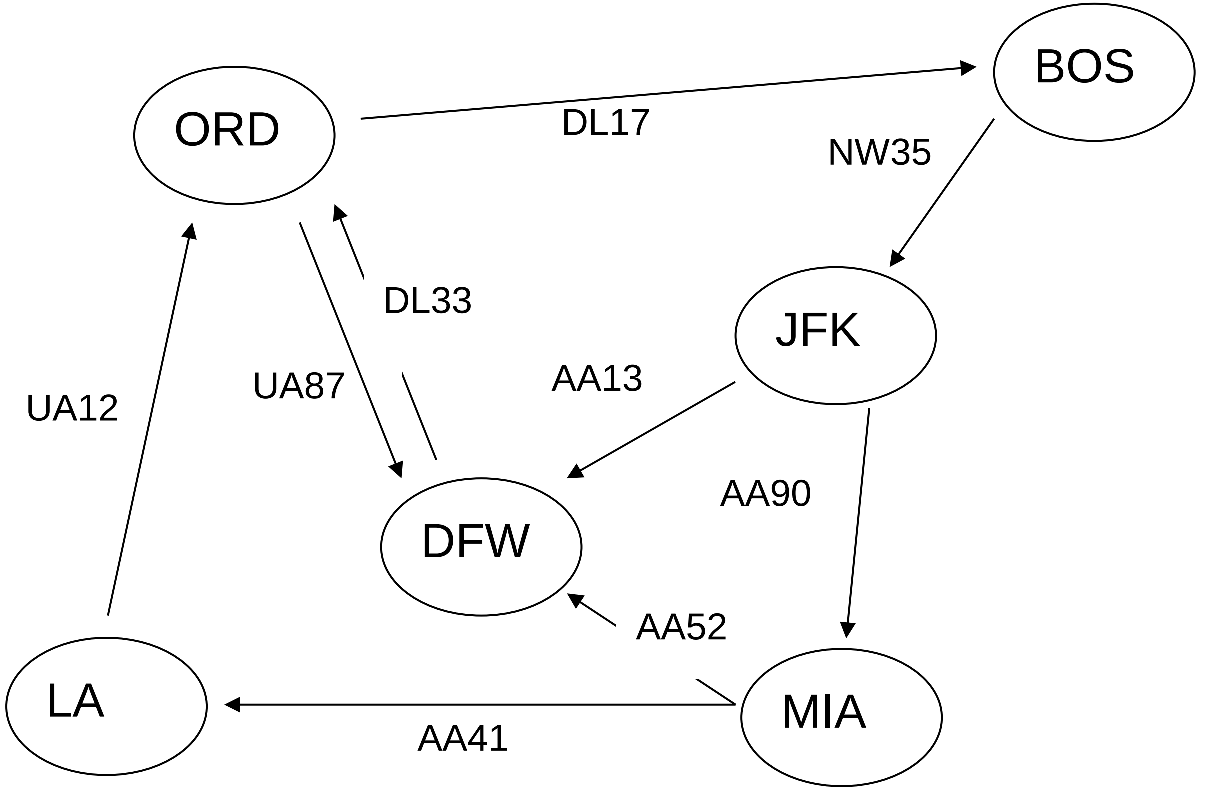


Arcs sortants



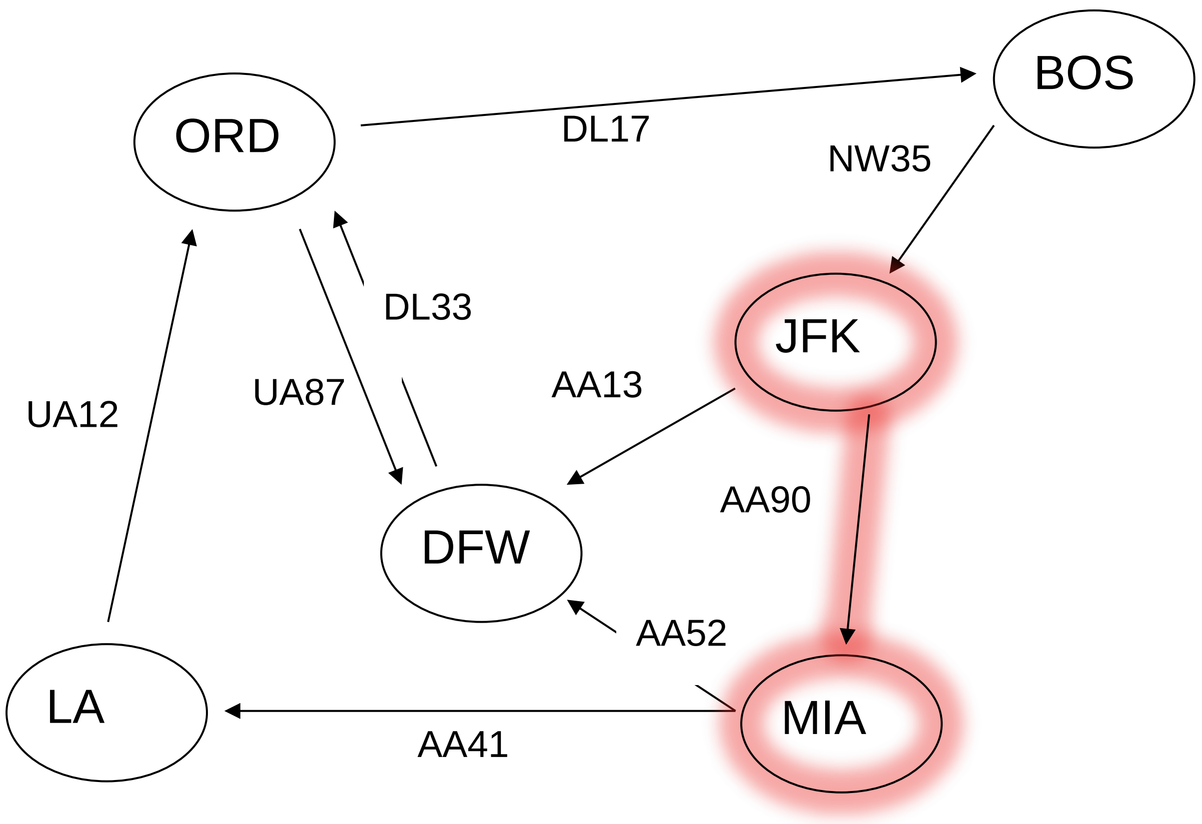
Arcs sortants de OR ={UA87,DL17}

Chemin



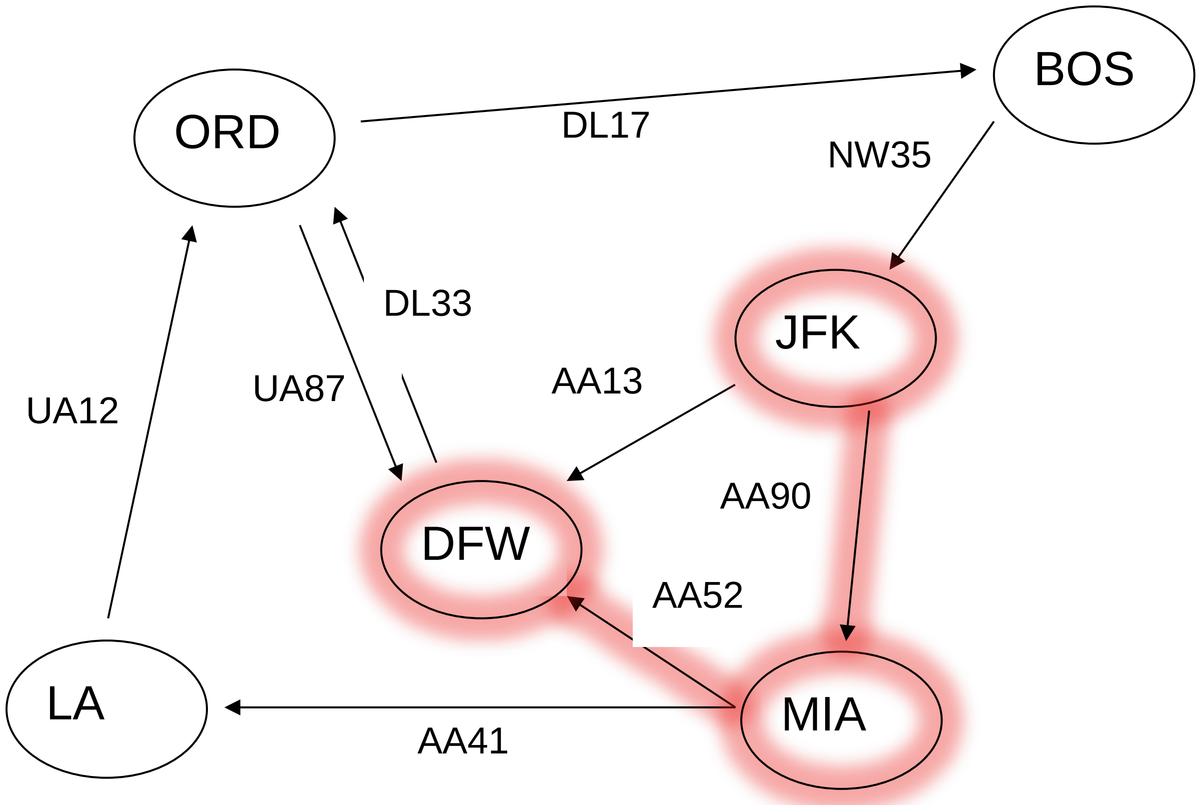
BOS
ORD
DL17
NW35
DL33
JFK
UA87
AA13
UA12
AA90
DFW
AA52
LA
MIA
AA41

Chemin



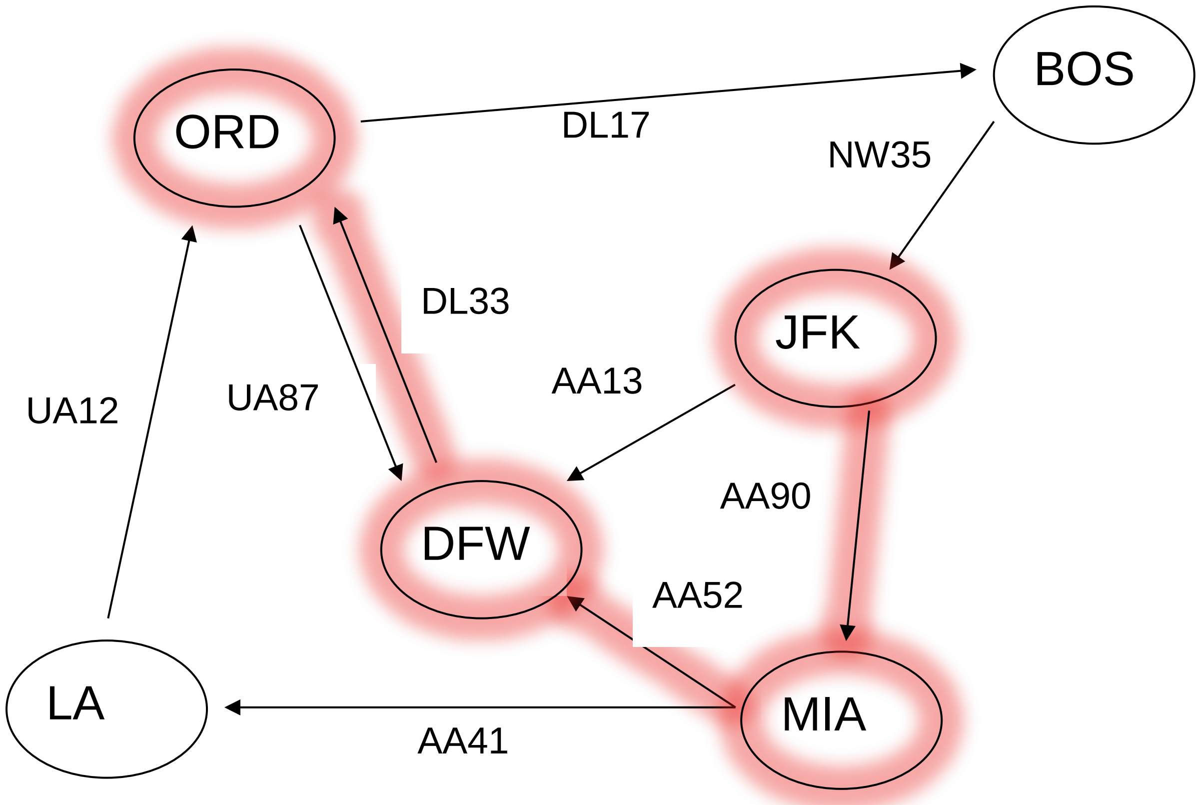
ORD
BOS
DL17
NW35
DL33
JFK
UA87
AA13
UA12
AA90
DFW
AA52
LA
MIA
AA41

Chemin



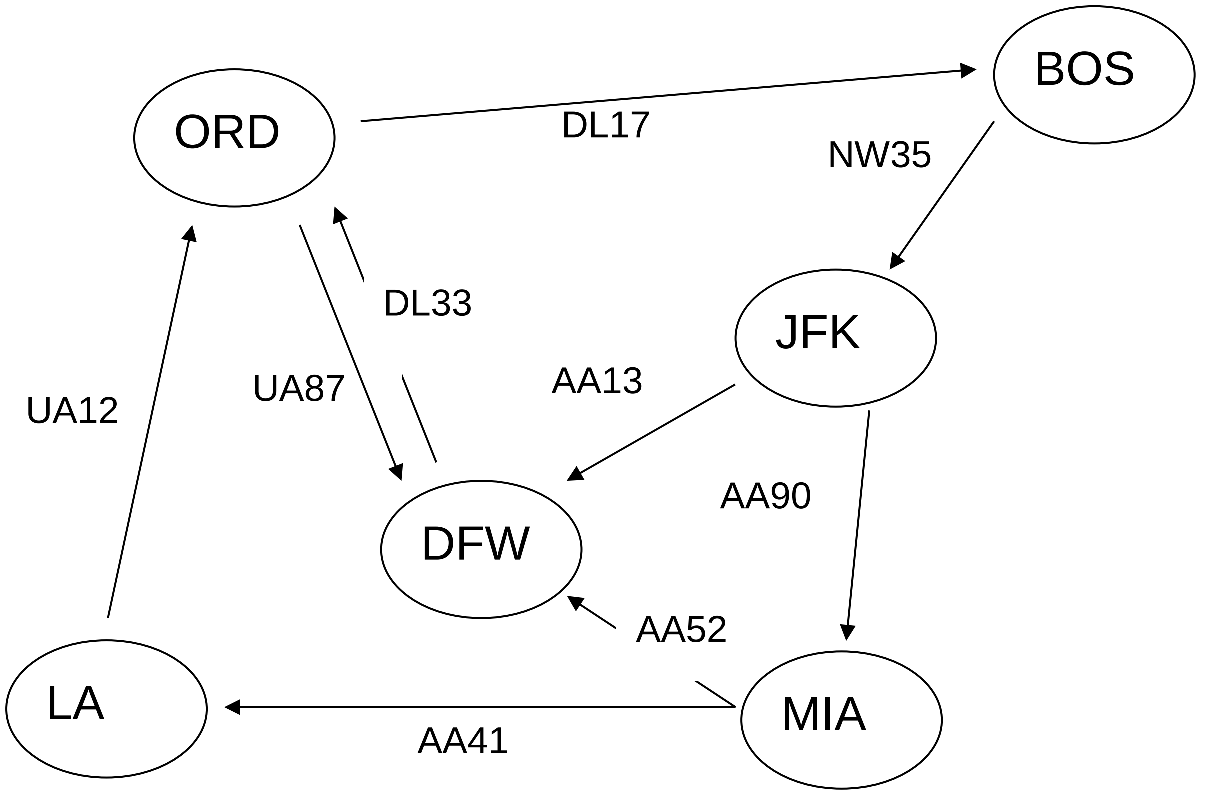
ORD
BOS
DL17
NW35
DL33
JFK
UA87
AA13
UA12
AA90
DFW
AA52
LA
MIA
AA41

Chemin



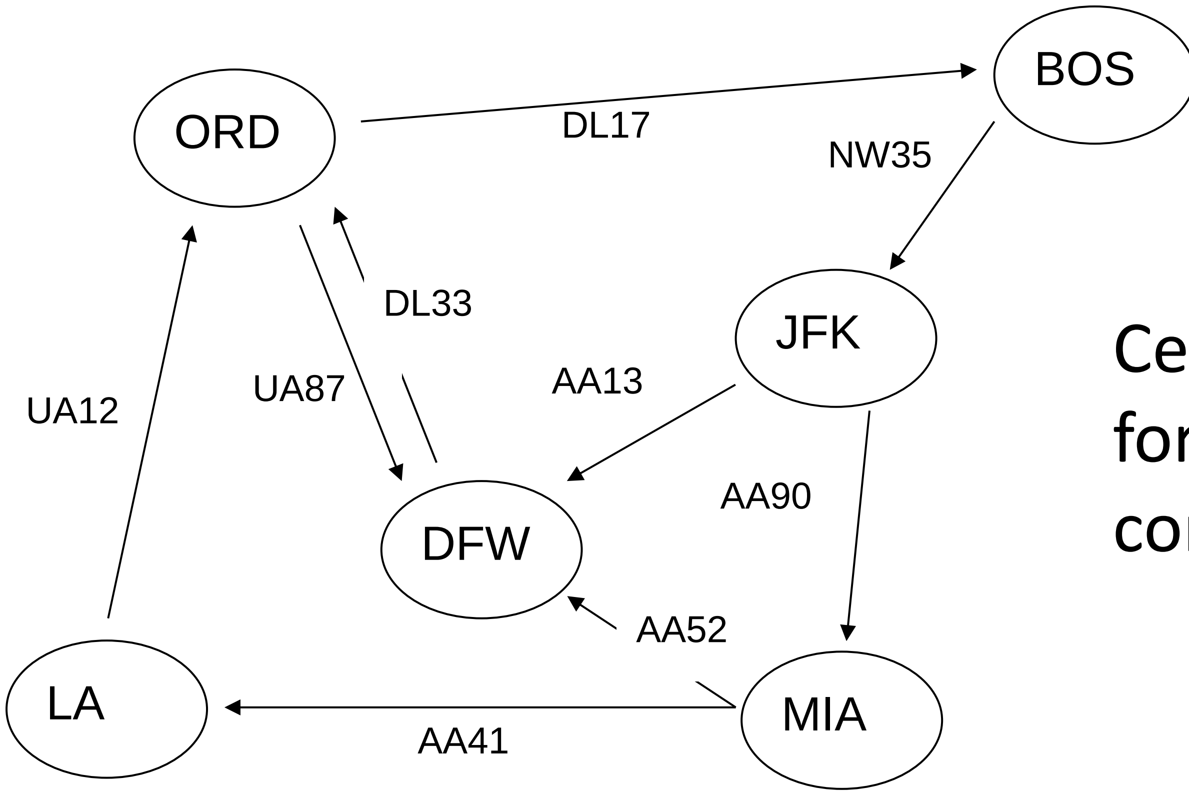
ORD
BOS
DL17
NW35
DL33
JFK
AA13
UA12
UA87
AA90
DFW
AA52
LA
MIA
AA41

Connexité



BOS
ORD
DL17
NW35
DL33
JFK
UA87
AA13
UA12
AA90
DFW
AA52
LA
MIA
AA41

Connexité



Ce graphe est fortement connexe.

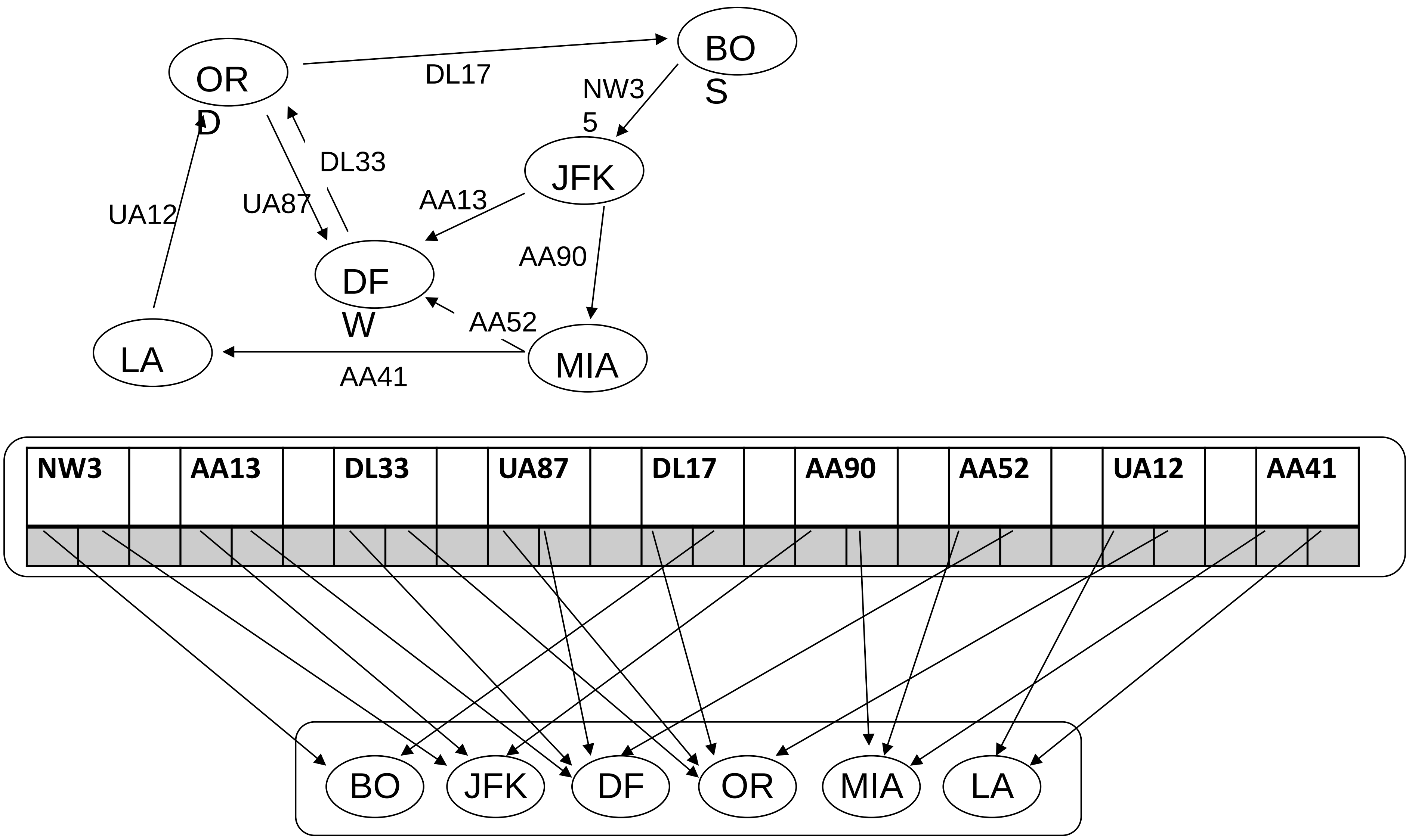
Exemple d’implémentation des graphes

- Classe Sommet
  - valeur du sommet
  - référence à l’endroit où ce sommet est stocké dans le conteneur des sommets (liste, ensemble, vecteur, …).
- Classe Arc
  - valeur de l’arc
  - référence vers le sommet origine
  - référence vers le sommet destination
  - référence à la position de cet arc dans le conteneur des arcs

3 exemples d’implémentation d’un graphe

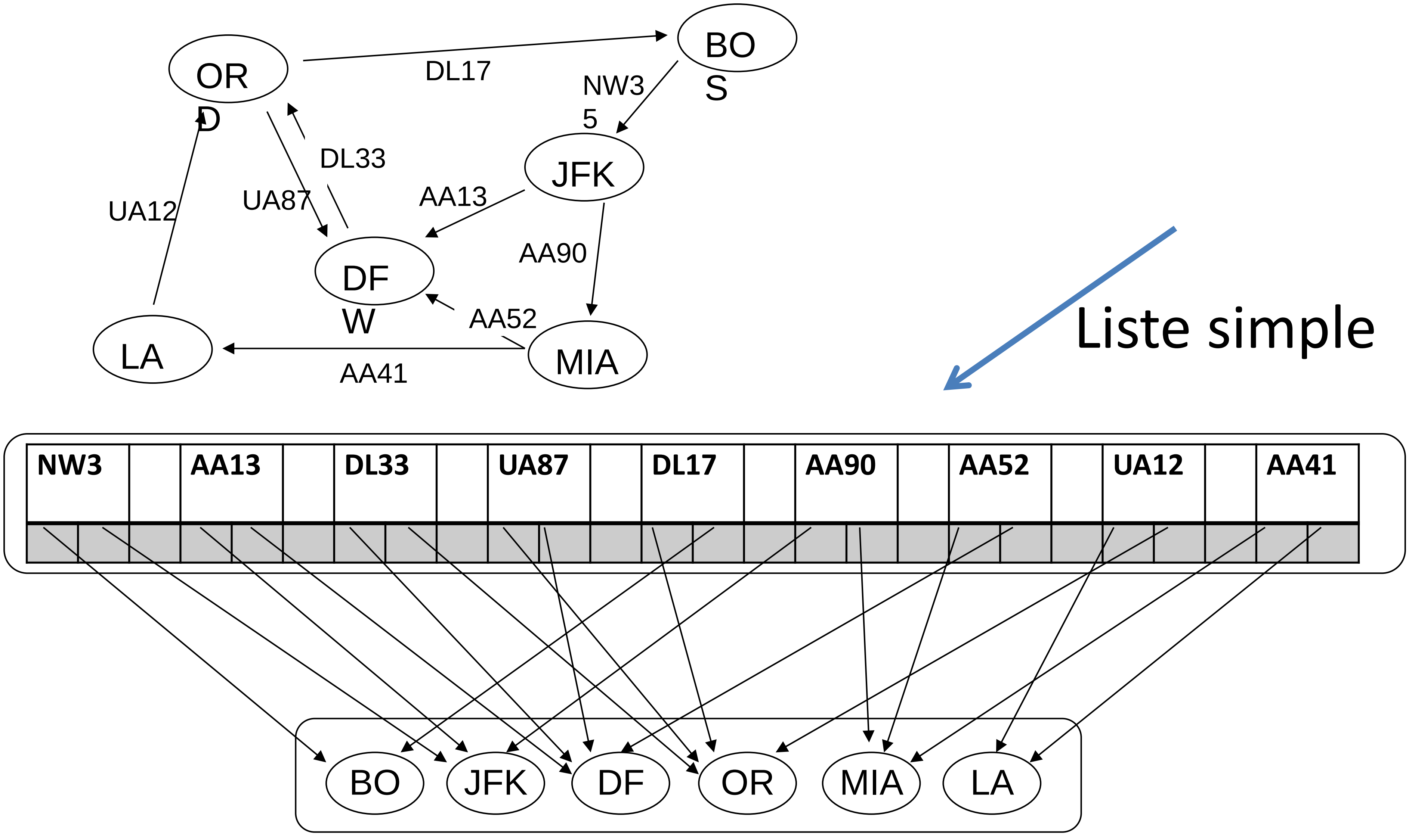
- Liste d’arcs
- Matrice d’adjacence
- Liste d’adjacence

Liste d'Arcs



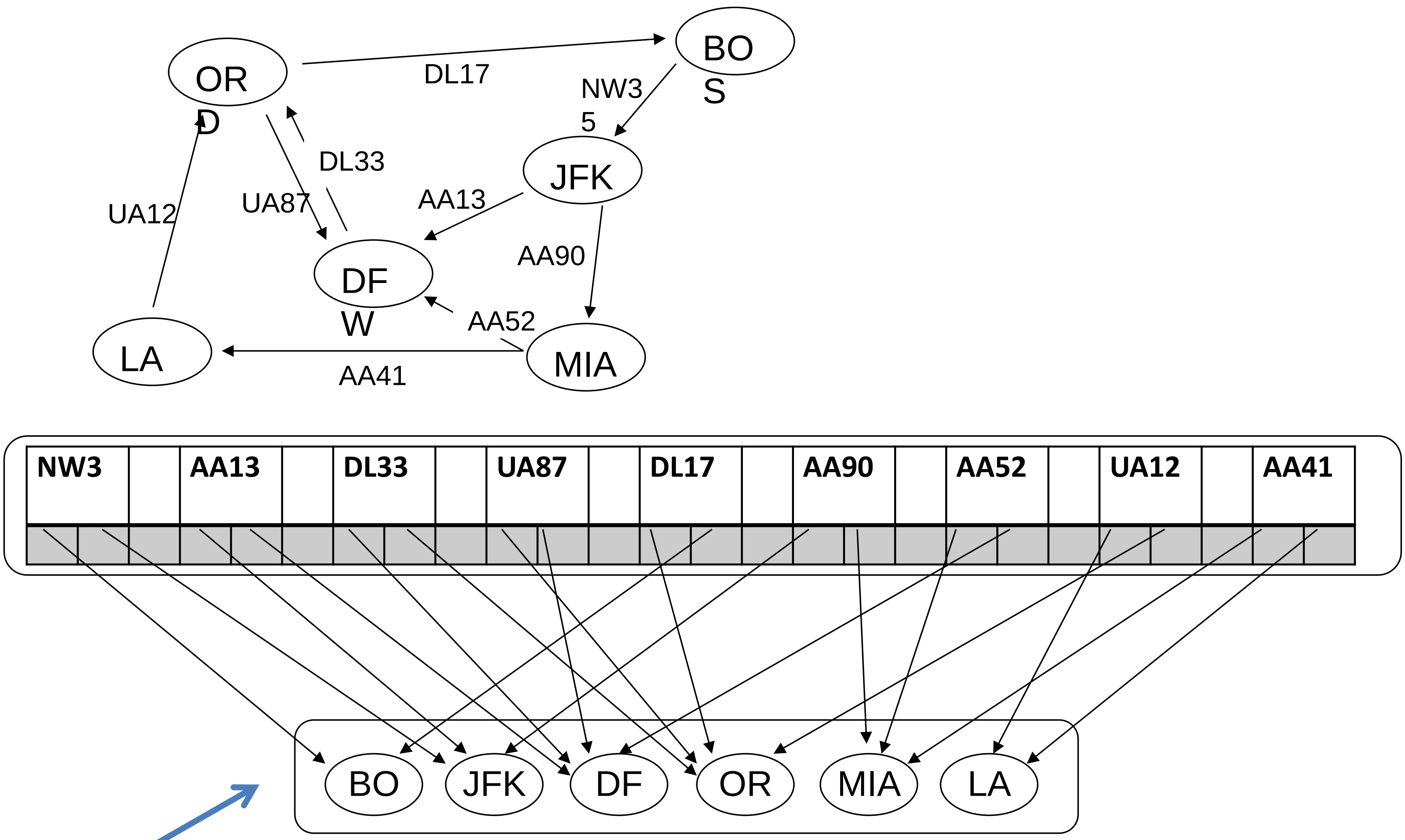
ORD
BOS
JFK
DFW
LA
MIA
DL17
NW35
DL33
UA87
UA12
AA13
AA90
AA52
AA41
NW3
AA13
DL33
UA87
DL17
AA90
AA52
UA12
AA41
BO
JFK
DF
OR
MIA
LA

Liste d'Arcs



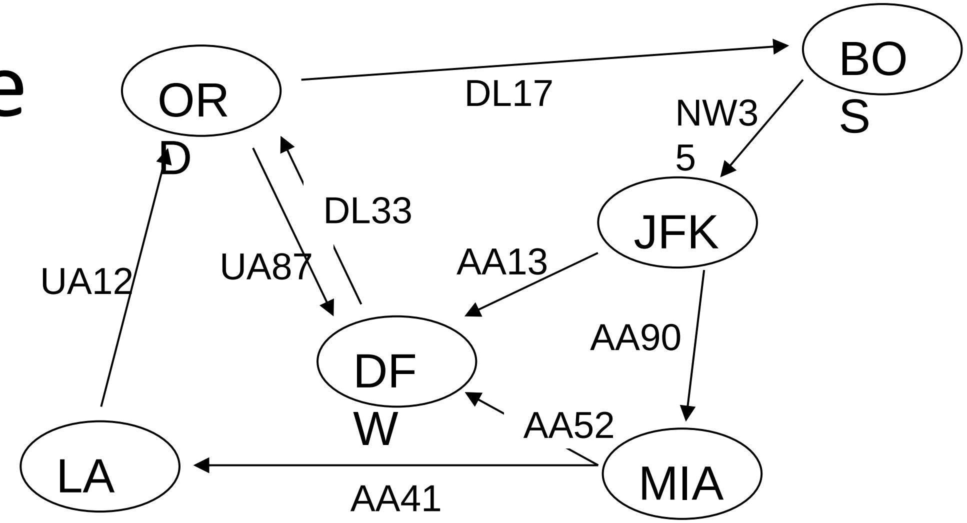
ORD
BOS
JFK
DFW
LA
MIA
DL17
NW35
DL33
UA87
UA12
AA13
AA90
AA52
AA41
Liste simple
NW3
AA13
DL33
UA87
DL17
AA90
AA52
UA12
AA41
BO
JFK
DF
OR
MIA
LA

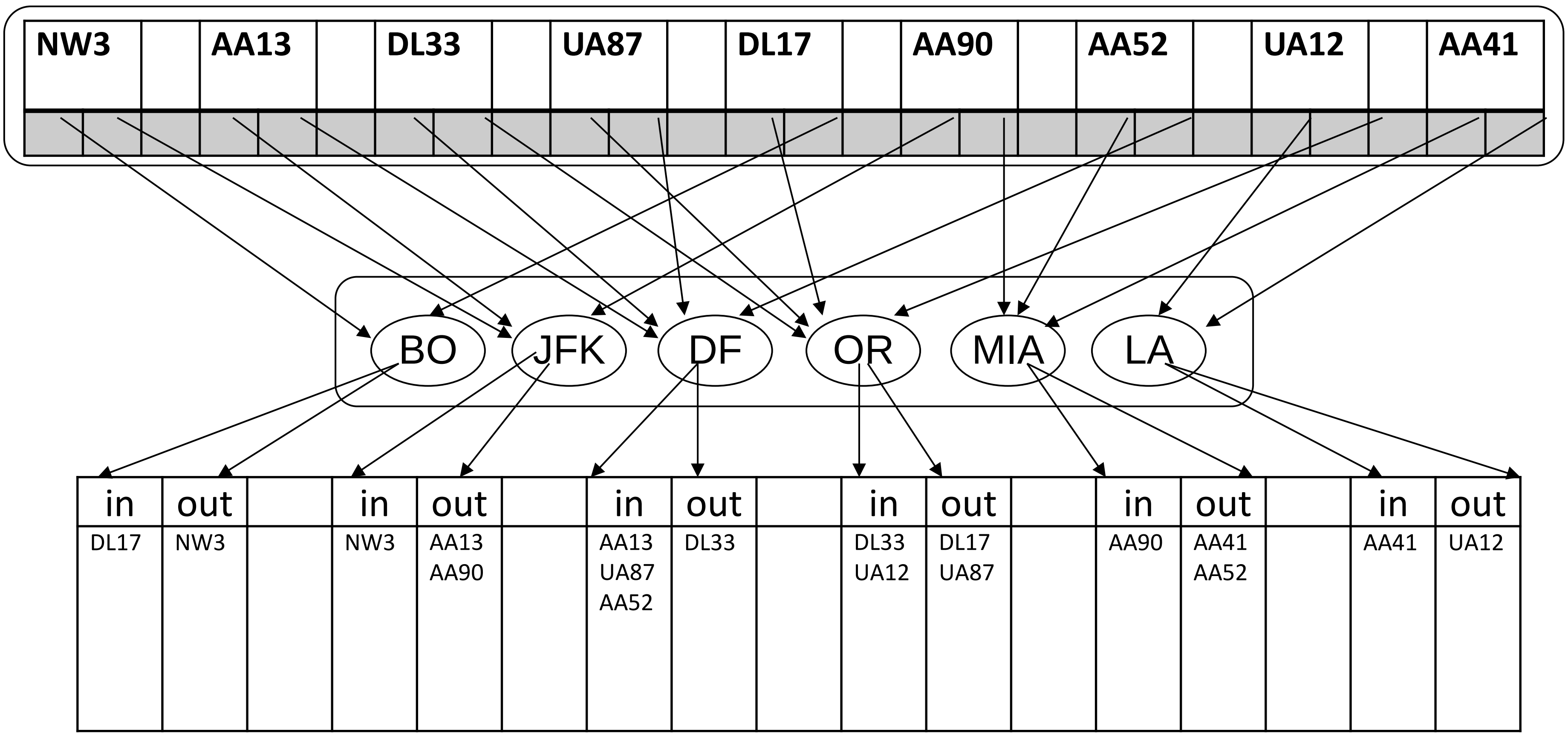
Liste d’Arcs



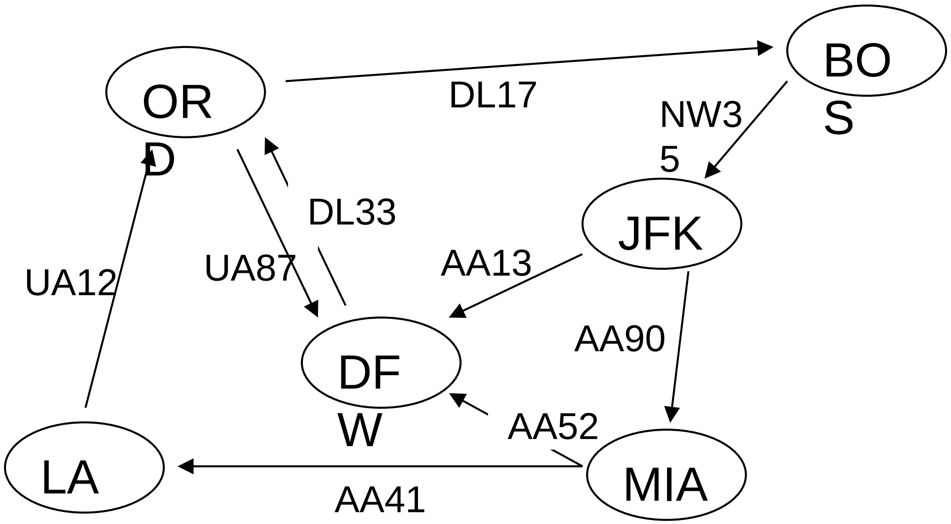
Liste simple ou …

Liste d’Adjacence





Matrice d’Adjacence



	0	1	2	3	4	5
0	-	DL17	-	-	UA87	-
1	-	-	-	NW3	-	-
2	UA12	-	-	-	-	-
3	-	-	-	-	AA13	AA90
4	DL33	-	-	-	-	-
5	-	-	AA41	-	AA52	-

0	1	2	3	4	5
OR	BO	LA	JFK	DF	MIA

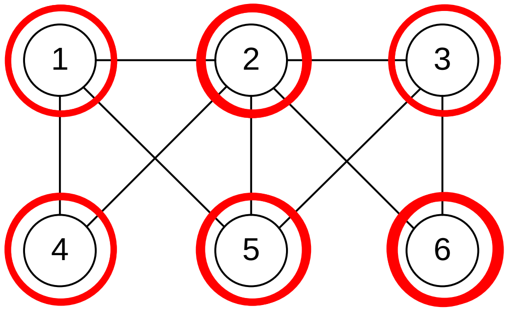
Algorithmes sur le graphes

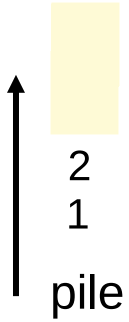
- Depth First Search (DFS)
- Breadth First Search (BFS)
- Algorithme du plus court chemin (Dijkstra)

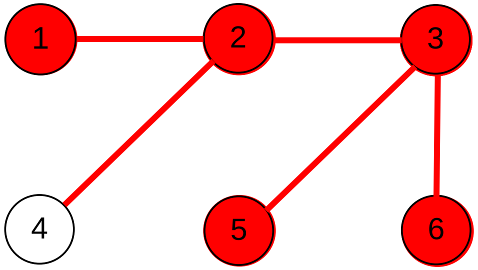
Depth First Search

Objectif : construire « en profondeur » un arbre couvrant
pour un graphe connexe.

Exemple







1° Fixer un sommet de départ (sommet courant)

2° Utiliser une « pile » auxiliaire

3° Sélectionner dans l'adjacence du sommet courant un nouveau sommet et le relier au sommet courant

4° Si c'est impossible, remonter au sommet précédent (dépiler)

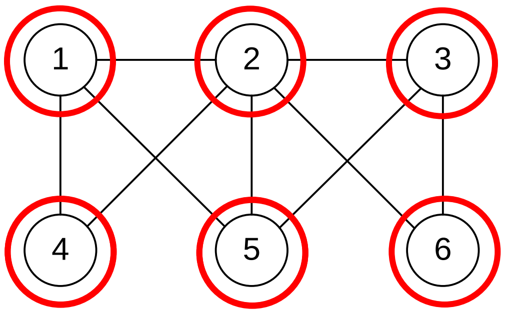
DFS

- Pour un graphe quelconque, le processus prend fin quand
  - On a capté tous les sommets
    - Le graphe est alors connexe
  - Ou lorsque la pile est vide
    - Le graphe est non connexe
    - On a construit un arbre couvrant pour la composante connexe du sommet de départ

Breadth-First Search

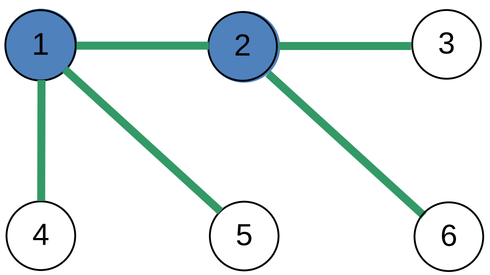
Objectif : construire « en largeur » un arbre couvrant
pour un graphe connexe.

Exemple



File

4 5 3 6



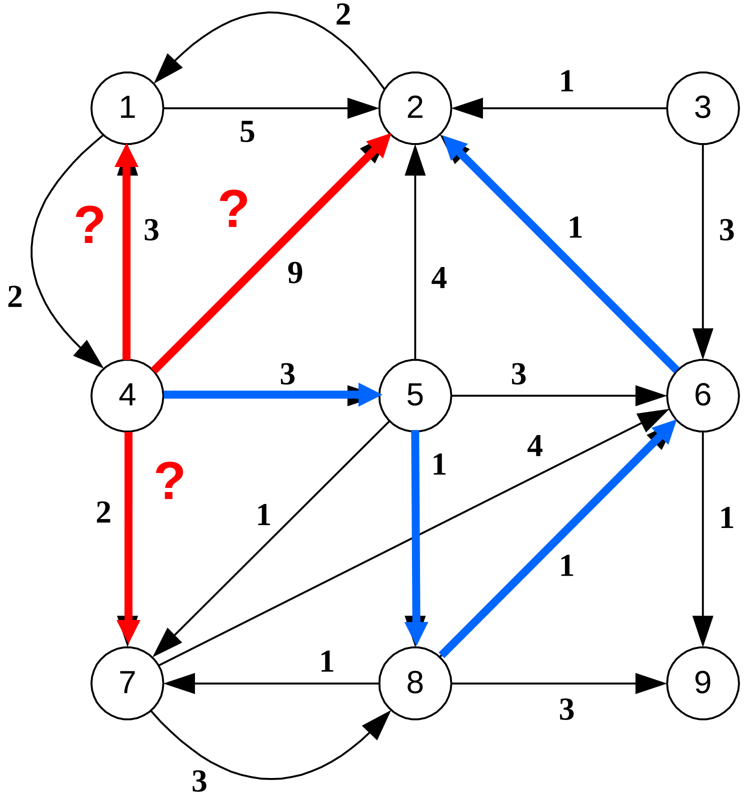
1° Fixer un sommet de départ (sommet courant)

2° Dans l'adjacence du sommet courant sélectionner tous les sommets non encore atteints, et les stocker dans une « file »

3° Le « premier » de file devient le nouveau courant

Algorithme du plus court chemin

Poids total 6



Rechercher le chemin
de **poids total minimum**,
d'un sommet d à un sommet a
dans un digraphe pondéré.

Par exemple, quel est le
« meilleur chemin »
de 4 à 2 ?

Comment choisir ?

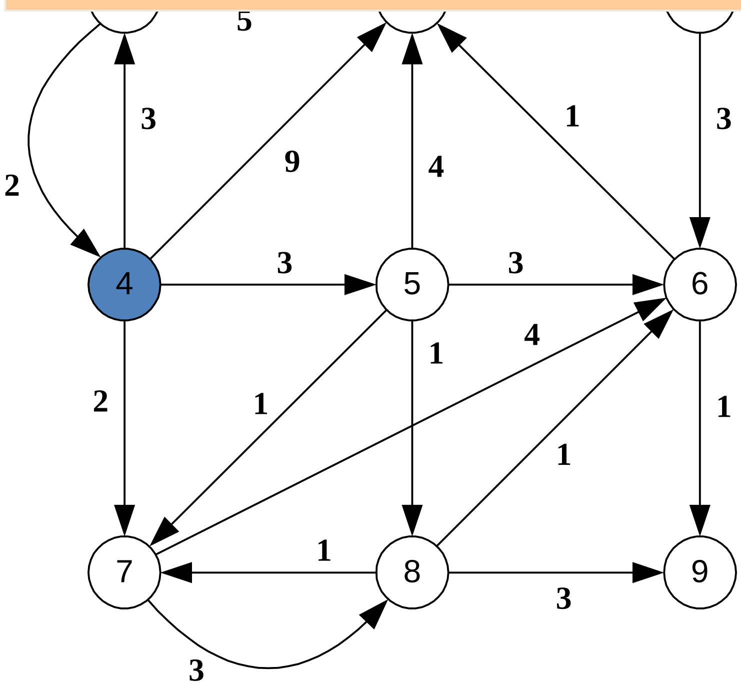
Comment concevoir un **algorithme**
permettant de découvrir
le « meilleur chemin »
d'un sommet à un autre ?

La réponse de Dijkstra

Dijkstra apporte une réponse à la question :

sommet de départ : 4

« quel est le meilleur chemin
d'un **sommet de départ fixé** à chacun des autres sommets ? »



donnent les poids des meilleurs chemins du sommet de départ vers chacun des sommets accessibles.

Etiquettes définitives :

3	**6**	**-**	**0**	**3**	**5**	**2**	**4**	**6**

I202B, Les arbres + la récursivité

J. Vander Meulen C. Damas

Mars 2017

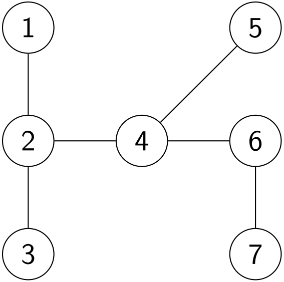
Qu'est ce qu'un arbre ?

Représentation informatique des arbres

Algorithmes

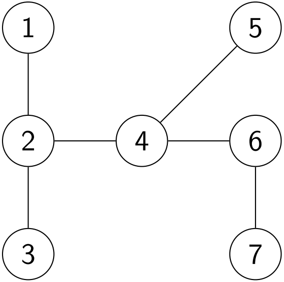
Un arbre (non orienté) est un graphe particulier

- Un graphe non vide, non orienté, acyclique et connexe.



Un arbre (non orienté) est un graphe particulier

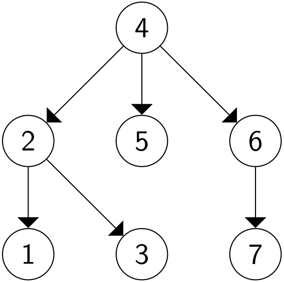
- Un graphe non vide, non orienté, acyclique et connexe.



- Un unique chemin d'un noeud à un autre

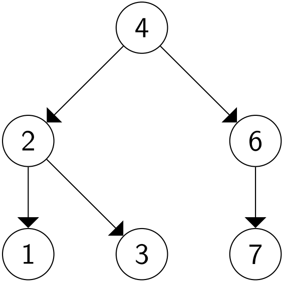
Un arbre (orienté) est aussi un graphe particulier

- Un graphe non vide, acyclique, ayant une unique racine et tel que tous les nœuds sauf la racine ont un unique parent

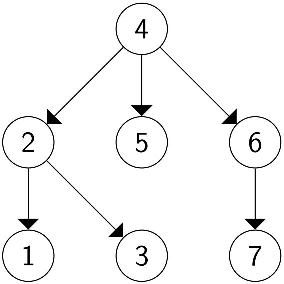


Un arbre binaire

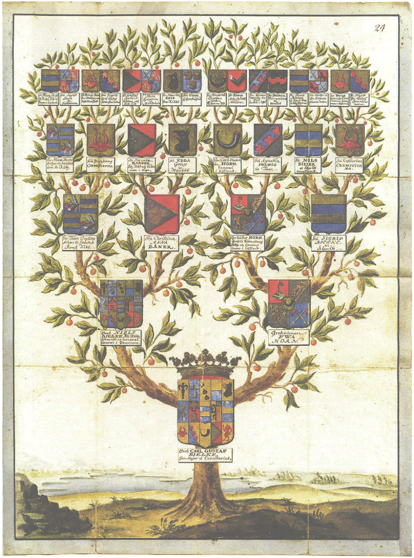
- Un arbre dont chaque noeud a au plus deux noeuds adjacents (souvent appelés fils gauche et fils droit)



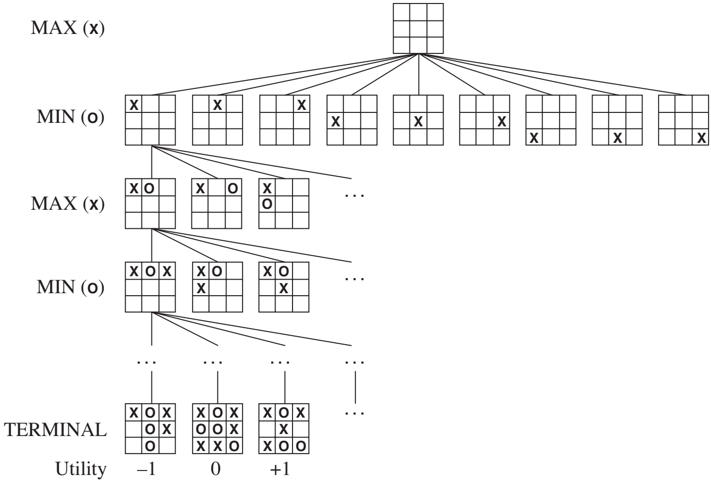
Tous les arbres ne sont pas binaires



Exemples 1 : les arbres généalogiques

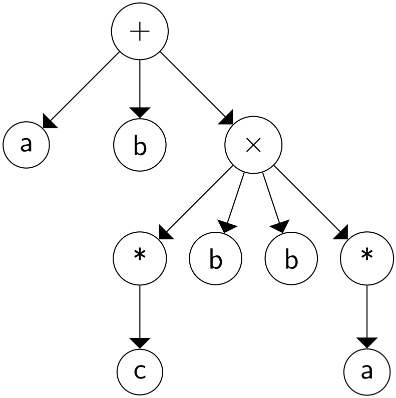


Exemples 2 : les arbres en IA

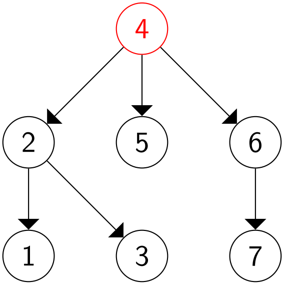


Exemples 3 : les arbres dans les compilateurs

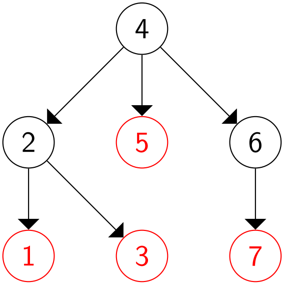
- Une expression : $a|b|c^*bba^*$



Terminologie : racine (anglais : root)

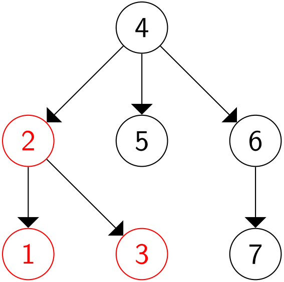


Terminologie : feuilles (anglais : leaves) (🍃)



Terminologie : parent–enfant (anglais : parent–children)

- 2 est le parent de 1 et 3
- 1 et 3 sont les enfants de 2



Une définition récursive des arbres

- Pour pouvoir écrire facilement des algorithmes récursifs sur les arbres, on va considérer une deuxième définition des arbres orientés

- Les deux définitions sont équivalentes

- L'ensemble des arbres qu'on peut définir avec les deux définitions sont les mêmes

Une définition récursive des arbres

- Dans un premier temps, on va définir les plus petits arbres possibles (les arbres composés d'un unique noeud)

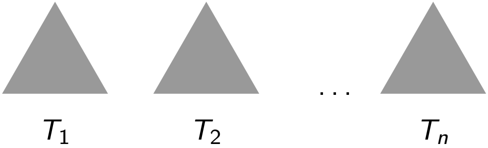
- Dans un second temps, on va définir des arbres de plus en plus grands (les arbres composés de plus d'un noeud)

Premier temps : soit un ensemble E, définition d'un arbre de hauteur 0

Une racine avec un label $e \in E$ est un arbre

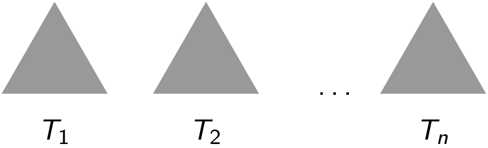
Second temps : soit un ensemble E, définition d'un arbre de hauteur > 0

- Supposons n arbres T_1, T_2, ..., T_n :

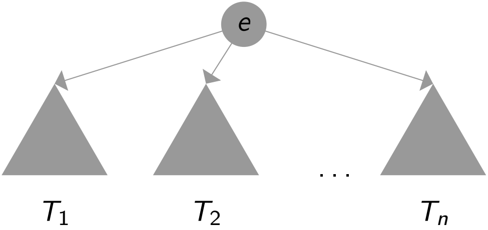


Second temps : soit un ensemble E, définition d'un arbre de hauteur > 0

- Supposons n arbres $T_1, T_2, \ldots, T_n$:

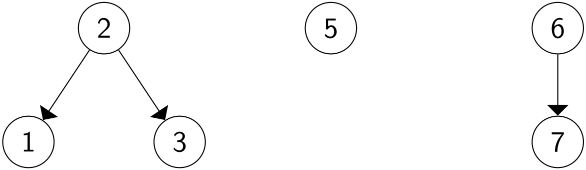


- Le graphe suivant, où $e \in E$, est aussi un arbre valide :



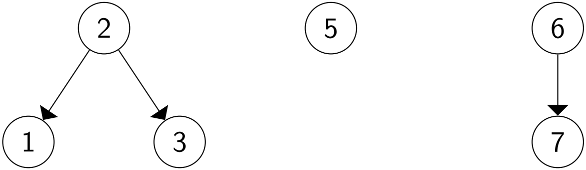
Second temps : exemple

- Supposons les 3 arbres :

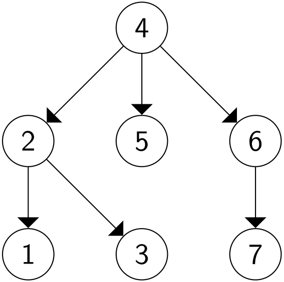


Second temps : exemple

- Supposons les 3 arbres :



- On peut construire l'arbre suivant :



Qu'est ce qu'un arbre ?

Représentation informatique des arbres

Algorithmes

Une représentation orientée POO
(application pratique de la définition récursive)

- Une classe Tree
- 3 attributs :
  - Une valeur
  - Une référence vers son parent
  - Un conteneur contenant des références vers ses fils
- 2 constructeurs :
  - Un destiné aux arbres de hauteur 0
  - Un autre destiné aux arbres de hauteur > 0

Classe Tree

```
public class Tree {
  private int value;
  private Tree parent;
  private Tree[] children;

  public Tree(int v, Tree[] chd) {
    value = v;
    children = chd;

    int i = 0;
    while(i != chd.length) {
      chd[i].parent = this;
      i++;
    }

    ...
  }
```

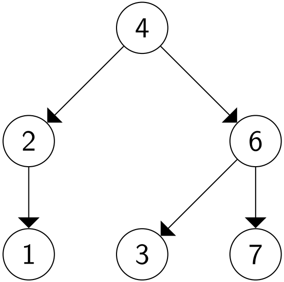
Classe Tree

```
public class Tree {
  ....

  public Tree(int v) {
    this(v, new Tree[0]);
  }

  ...
}
```

Créer un arbre

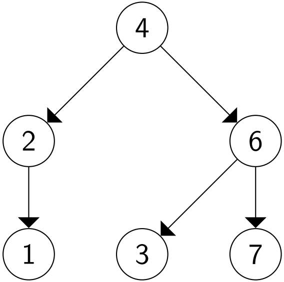
```
public class Main{
  public static void main(String[] args){
    Tree l1 = new Tree(1);
    Tree l3 = new Tree(3);
    Tree l5 = new Tree(5);
    Tree l7 = new Tree(7);

    Tree t2 = new Tree(2, new Tree[]{l1, l3});
    Tree t6 = new Tree(6, new Tree[]{l7});

    Tree t4 = new Tree(4, new Tree[]{t2, l5, t6});
  }
}
```

Une représentation d'arbres binaires avec un tableau



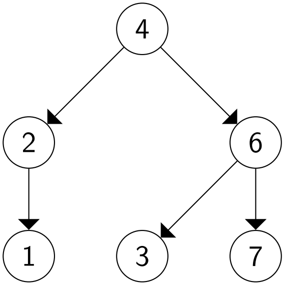
valeur	4	2	1	6	3	7	0
fils gauche	1	2	2	4	4	5	1
fils droit	3	1	2	5	4	5	2
	0	1	2	3	4	5	

Une représentation d'arbres binaires avec un tableau



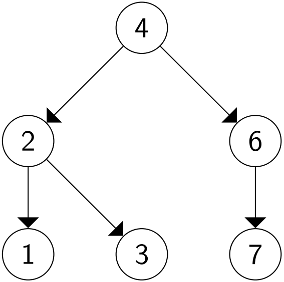
valeur	4	2	1	6	3	7	0
fils gauche	1	2	2	4	4	5	1
fils droit	3	1	2	5	4	5	2
	0	1	2	3	4	5	

Une représentation d'arbres binaires avec un tableau



valeur	4	2	1	6	3	7	0
fils gauche	1	2	2	4	4	5	1
fils droit	3	1	2	5	4	5	2
	0	1	2	3	4	5	

Une représentation d'arbres binaires complets avec un tableau



valeur	4	2	6	1	3	7
	0	1	2	3	4	5

Qu'est ce qu'un arbre ?

Représentation informatique des arbres

Algorithmes

Les algorithmes récursifs : intuitions

Pour résoudre un problème, on considère généralement différents cas :

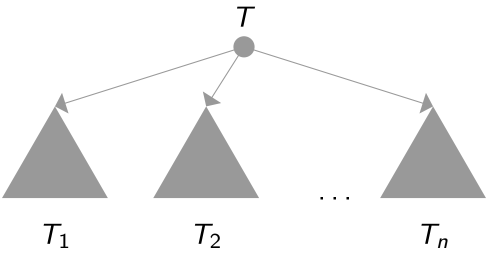
- cas simples :
  - par exemple, les arbres de hauteur 0
  - il existe généralement un algo trivial pour ces instances

Les algorithmes récursifs : intuitions

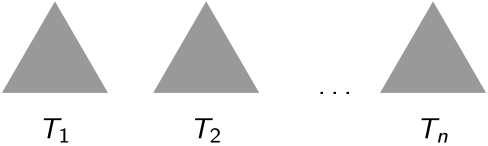
Pour résoudre un problème, on considère généralement différents cas :

- cas simples :
  - par exemple, les arbres de hauteur 0
  - il existe généralement un algo trivial pour ces instances
- cas plus complexes :
  - par exemple, les arbres de hauteur > 0
  - avant de traiter ces instances, on va considérer des instances plus petites.

Avant de traiter tout cet arbre



On va traiter uniquement ses sous-arbres (de manière récursive)



Remarque : les sous-arbres T_1, T_2, ..., T_n sont plus petits que l'arbre initial

Un 1er exemple : #🍃 d'un arbre

- Écrire un algorithme récursif qui renvoie le #🍃 d'un arbre

```
public static int nbrLeaves(Tree t)
```

Un 1er exemple : #🍃 d'un arbre

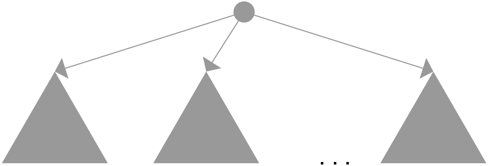
- Écrire un algorithme récursif qui renvoie le #🍃 d'un arbre

```
public static int nbrLeaves(Tree t)
```

- Cas simple : le #🍃 d'une 🍃 $= 1$

$\#$🍃 d'un arbre de hauteur > 0

$$\#🍃(T) =$$



$$\#🍃(T_1) + \#🍃(T_2) + \quad \cdots + \#🍃(T_n)$$

: code Java

```java
public static int nbrLeaves(Tree t) {
  int r;
  if (t.children.length == 0) {
    r = 1;
  } else {
    r = 0;
    int i = 0;
    while (i != t.children.length) {
      r += nbrLeaves(t.children[i]);
      i++;
    }
  }
  return r;
}
```

Un 2e exemple : aplanir les 🍃 d'un arbre

- Écrire un algorithme récursif qui renvoie un tableau contenant les 🍃 d'un arbre

```
public static Tree[] flattenLeaves(Tree t)
```

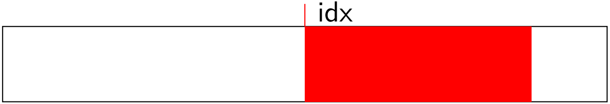
Un 2e exemple : aplanir les 🍃 d'un arbre

- Écrire un algorithme récursif qui renvoie un tableau contenant les 🍃 d'un arbre

```
public static Tree[] flattenLeaves(Tree t)
```

- Pour résoudre ce prb on va construire un algorithme plus général
  - On place les 🍃 d'un arbre dans une partie de tableau

```
static int flattenLeaves(Tree t, Tree[] a, int idx)
```



Un 2e exemple : code Java

```java
static int flattenLeaves(Tree t, Tree[] a, int idx) {
  int r;
  if (t.children.length == 0) {
    a[idx] = t;
    r = 1;
  } else {
    r = 0;
    int i = 0;
    while (i != t.children.length) {
      r += flattenLeaves(t.children[i], a, idx + r);
      i++;
    }
  }
  return r;
}
```

Un 2e exemple : code Java

```
static int flattenLeaves(Tree t) {
  int nl = nbrLeaves(t);
  Tree[] r = new Tree[nl];
  flattenLeaves(t, r, 0);
  return r;
}
```

Un 3e exemple : imprimer le chemin vers la racine

- Tous les algorithmes traitant des arbres ne sont pas obligatoirement récursifs

```
public static void pathV1(Tree t) {
  System.out.println(t.value);
  if (t.parent != null) {
    pathV1(t.parent);
  }
}

public static void pathV2(Tree t) {
  while(t != null) {
    System.out.println(t.value);
    t = t.parent;
  }
}
```

I202B, Union-find

J. Vander Meulen C. Damas

Mars 2017

Description de la structure

Représentation

Algorithmes

Application

Représenter des partitions d'un intervalle $E = [0 \ldots n[$

- Soit un intervalle fini E

$$E = \{0, 1, 2, 3, 4, 5, 6, 7\}$$

Représenter des partitions d'un intervalle $E = [0 \ldots n[$

- Soit un intervalle fini E

$$E = \{0, 1, 2, 3, 4, 5, 6, 7\}$$

- On veut représenter et manipuler des partitions de E

$$\boxed{0 \quad 3 \quad 6 \quad 7} \quad \boxed{2} \quad \boxed{1 \quad 4 \quad 5}$$

À partir d'une partition initiale de E

- $\Big\{\{e\} \mid e \in E\Big\}$
- $\Big\{\{0\},\{3\},\{6\},\{7\},\{2\},\{1\},\{4\},\{5\}\Big\}$
- C'est la relation identité abordée durant le cours de mathématiques du bloc 1.

On veut pouvoir

- Fusionner deux parties de la partition

$$\Big\{\{0\},\{3\},\{6\},\{7\},\{2\},\{1\},\{4\},\{5\}\Big\}$$

On veut pouvoir

- Fusionner deux parties de la partition

$$\Big\{\{0, 3\}, \{6\}, \{7\}, \{2\}, \{1\}, \{4\}, \{5\}\Big\}$$

On veut pouvoir

- Fusionner deux parties de la partition

$$\Big\{\{0,3\},\{6,7\},\{2\},\{1\},\{4\},\{5\}\Big\}$$

On veut pouvoir

- Fusionner deux parties de la partition

$$\Big\{\{0,3,6,7\},\{2\},\{1\},\{4\},\{5\}\Big\}$$

On veut pouvoir

- Fusionner deux parties de la partition

$$\Big\{\{0,3,6,7\},\{2\},\{1\},\{4,5\}\Big\}$$

On veut pouvoir

- Fusionner deux parties de la partition

$$\Big\{\{0,3,6,7\},\{2\},{\color{red}\{1,4,5\}}\Big\}$$

On veut pouvoir

- Fusionner deux parties de la partition

$$\Big\{\{0,3,6,7\},\{2\},\{1,4,5\}\Big\}$$

- Répondre à la question :

Deux éléments appartiennent-ils à la même partie ?

$$\Big\{\{0,3,{\color{red}6},7\},\{2\},\{1,{\color{red}4},5\}\Big\} \Rightarrow \text{Non}$$

On veut pouvoir

- Fusionner deux parties de la partition

$$\Big\{\{0,3,6,7\},\{2\},\{1,4,5\}\Big\}$$

- Répondre à la question :

Deux éléments appartiennent-ils à la même partie ?

$$\Big\{\{\textcolor{red}{0},3,\textcolor{red}{6},7\},\{2\},\{1,4,5\}\Big\} \Rightarrow \text{Oui}$$

On veut pouvoir

- Fusionner deux parties de la partition

$$\Big\{\{0,3,6,7\},\{2\},\{1,4,5\}\Big\}$$

- Répondre à la question :

  Deux éléments appartiennent-ils à la même partie ?

$$\Big\{\{0,3,6,7\},\{2\},\{1,4,5\}\Big\} \Rightarrow \text{Oui}$$

- Compter le nombre d'éléments d'une partie

$$\#\{0,3,6,7\} = 4$$

Les représentants des parties

Pour des raisons algorithmiques :

chaque partie est associée à un représentant

$$\Big\{\{0, 3, \textcolor{red}{6}, 7\}, \{\textcolor{red}{2}\}, \{\textcolor{red}{1}, 4, 5\}\Big\}$$

Description de la structure

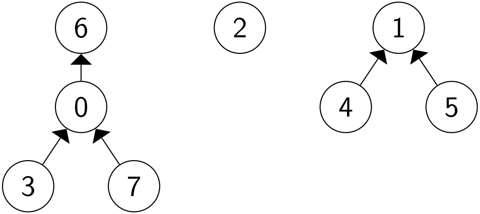
Représentation

Algorithmes

Application

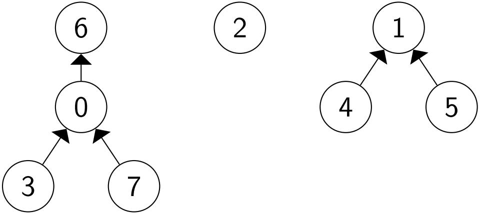
Une partition d'un intervalle E est représentée abstraitement par une forêt

$$\Big\{\{0,3,6,7\},\{2\},\{1,4,5\}\Big\}$$



Une partition d'un intervalle E est représentée concrètement par une tableau

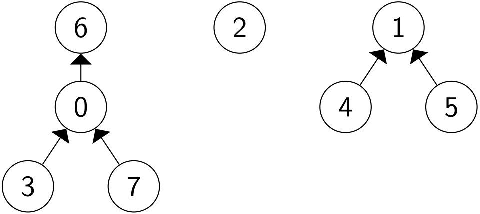
$$\Big\{\{0,3,6,7\},\{2\},\{1,4,5\}\Big\}$$



# du sous-arbre	3	3	1	1	1	1	4	1	0
indice parent	6	1	2	0	1	1	6	0	1
	0	1	2	3	4	5	6	7	

Une partition d'un intervalle E est représentée concrètement par une tableau

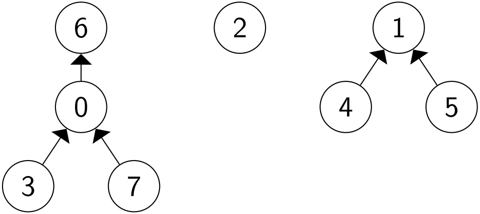
$$\big\{\{0,3,6,7\},\{2\},\{1,4,5\}\big\}$$



# du sous-arbre	3	3	1	1	1	1	4	1	0
indice parent	6	1	2	0	1	1	6	0	1
	0	1	2	3	4	5	6	7	

Une partition d'un intervalle E est représentée concrètement par une tableau

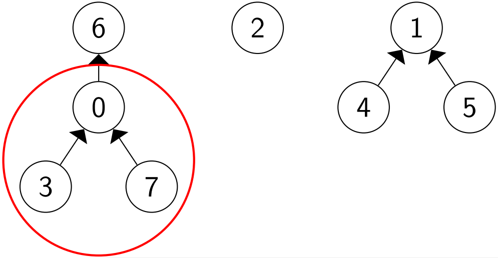
$$\Big\{\{0,3,6,7\},\{2\},\{1,4,5\}\Big\}$$



# du sous-arbre	3	3	1	1	1	1	4	1	0
indice parent	6	1	2	0	1	1	6	0	1
	0	1	2	3	4	5	6	7	

Une partition d'un intervalle E est représentée concrètement par une tableau

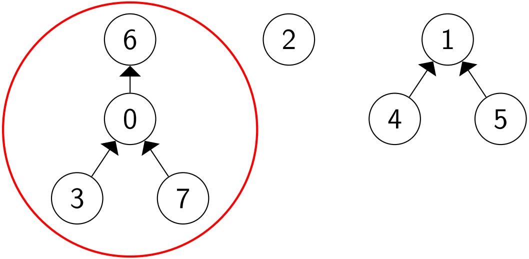
$$\Big\{\{0,3,6,7\},\{2\},\{1,4,5\}\Big\}$$



# du sous-arbre	3	3	1	1	1	1	4	1	0
indice parent	6	1	2	0	1	1	6	0	1
	0	1	2	3	4	5	6	7	

Une partition d'un intervalle E est représentée concrètement par une tableau

$$\Big\{\{0,3,6,7\},\{2\},\{1,4,5\}\Big\}$$

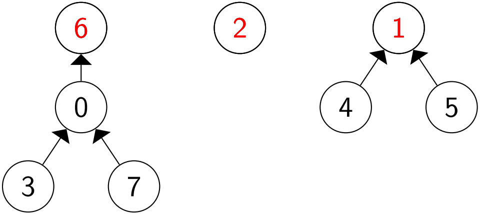


	0	1	2	3	4	5	6	7	
# du sous-arbre	3	3	1	1	1	1	4	1	0
indice parent	6	1	2	0	1	1	6	0	1

Une partition d'un intervalle E est représentée abstraitement par une forêt

- Les racines des arbres sont les représentants des $\neq$ parties

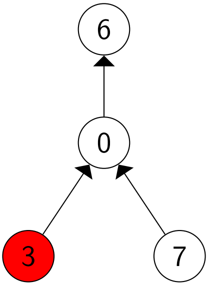
- $\{\{0,3,6,7\},\{2\},\{1,4,5\}\}$



# du sous-arbre	3	3	1	1	1	1	4	1	0
indice parent	6	1	2	0	1	1	6	0	1
	0	1	2	3	4	5	6	7	

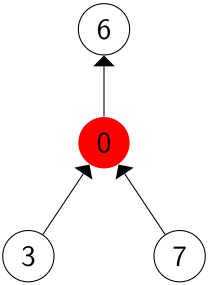
Trouver la racine d'un arbre

Une première opération interne, à partir d'un noeud de départ, on remonte jusqu'à la racine.



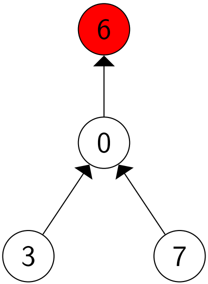
Trouver la racine d'un arbre

Une première opération interne, à partir d'un noeud de départ, on remonte jusqu'à la racine.



Trouver la racine d'un arbre

Une première opération interne, à partir d'un noeud de départ, on remonte jusqu'à la racine.



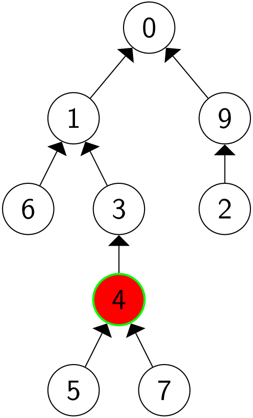
Trouver le représentant de la partie d'un nombre n (find)

- 1) Trouver la racine R de l'arbre auquel le noeud N associé au nombre n appartient.

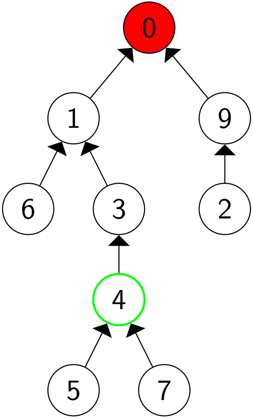
  Le nombre associé à la racine R est le représentant de la partie du nombre n

- 2) Compresser le chemin allant de la racine de R à N

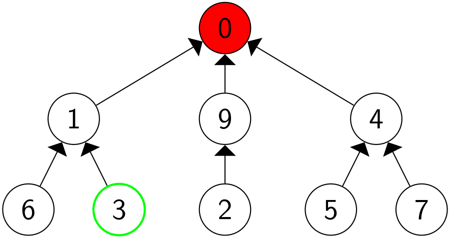
Trouver le représentant de la partie du nombre 4



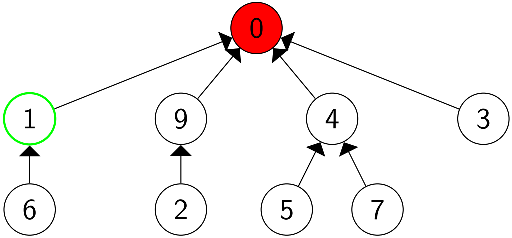
Trouver le représentant de la partie du nombre 4



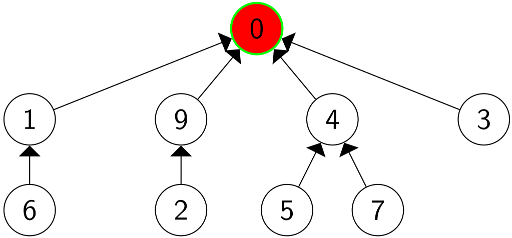
Trouver le représentant de la partie du nombre 4



Trouver le représentant de la partie du nombre 4



Trouver le représentant de la partie du nombre 4



Pour compter le nombre d'éléments d'une partie P à laquelle un nombre n appartient

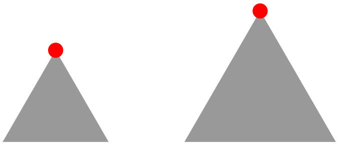
- $\#\{0, 3, 6, 7\} = 4$
- On trouve le représentant de la partie P (find)
- Ensuite, on trouve trivialement la taille de cette partie dans le tableau qui représente l'arbre de la partie P

Pour vérifier si deux nombres n_1 et n_2 appartiennent à la même partie

- $\Big\{\{0, 3, 6, 7\}, \{2\}, \{1, 4, 5\}\Big\} \Rightarrow \mathrm{Non}$
- $\Big\{\{0, 3, 6, 7\}, \{2\}, \{1, 4, 5\}\Big\} \Rightarrow \mathrm{Oui}$
- On trouve les représentants des deux parties de n_1 et de n_2
- Ensuite, on vérifie que ces deux représentants sont les mêmes

Pour fusionner deux parties P_1 et P_2 de la partition (union)

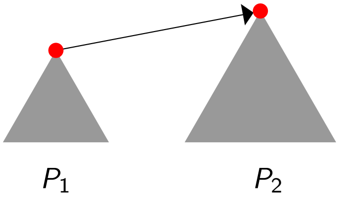
- On trouve les représentants des deux parties P_1 et P_2



P_1 P_2

Pour fusionner deux parties P_1 et P_2 de la partition (union)

- On trouve les représentants des deux parties P_1 et P_2
- On « attache » la plus petite partie à la plus grande



Complexité

Si $E = [0..n[$ et que l'on effectue m opérations find ou union :

$$\mathcal{O}(\alpha(n)m)$$

où

- α est la fonction inverse de la fonction d'Ackermann
- α est une fonction qui croît extrêmement lentement

Complexité

Si $E = [0..n[$ et que l'on effectue m opérations find ou union :

$$\mathcal{O}(\alpha(n)m)$$

où

- α est la fonction inverse de la fonction d'Ackermann
- α est une fonction qui croît extrêmement lentement

En pratique

$$\approx \mathcal{O}(m)$$

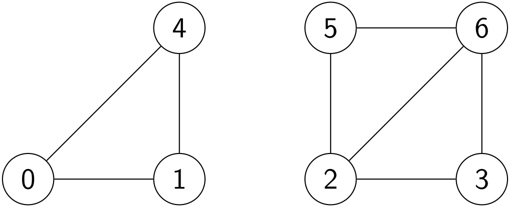
Description de la structure

Représentation

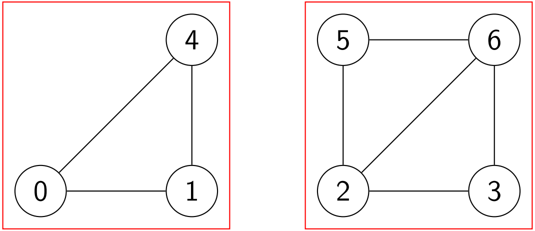
Algorithmes

Application

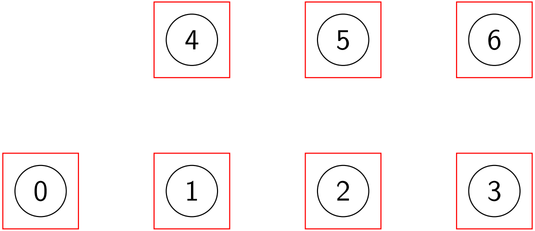
Composantes fortement connexes d'un graphe



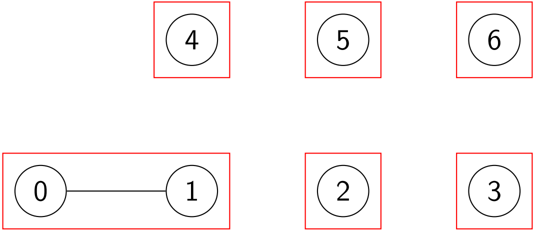
Composantes fortement connexes d'un graphe



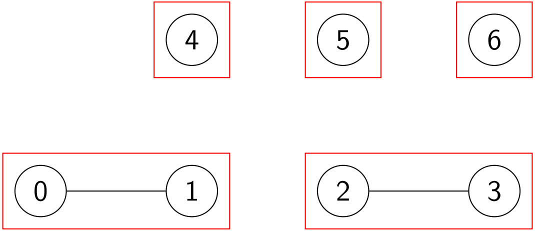
Composantes connexes d'un graphe



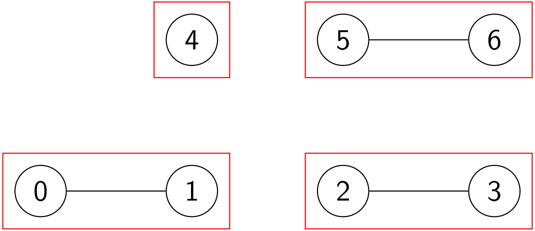
Composantes connexes d'un graphe



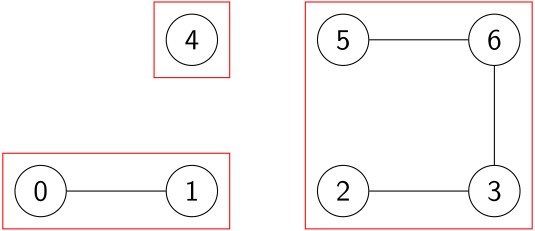
Composantes connexes d'un graphe



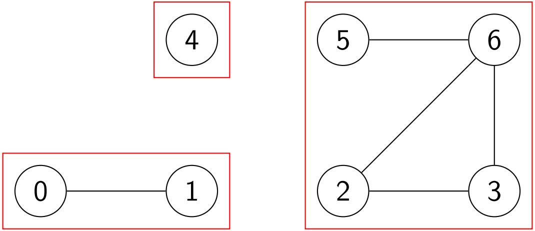
Composantes connexes d'un graphe



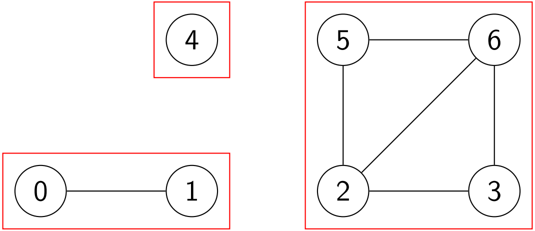
Composantes connexes d'un graphe



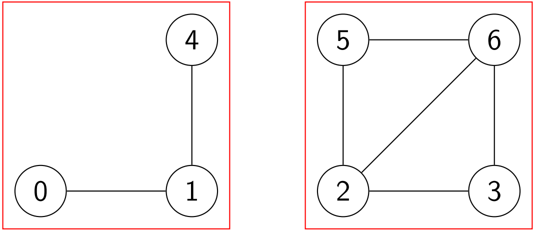
Composantes connexes d'un graphe



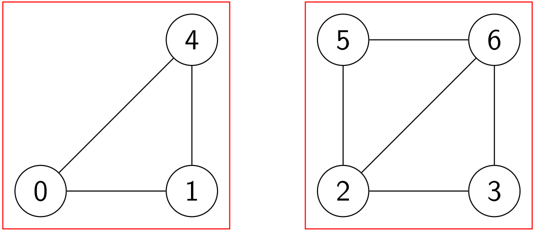
Composantes connexes d'un graphe



Composantes connexes d'un graphe



Composantes connexes d'un graphe



Codes de Huffman

Méthode de compression des fichiers

BUT : coder les "lettres" de plus grandes fréquences sur moins de bits que celles de fréquences plus petites

Mauvais exemple

CAFE
010111

Quel est le problème ?

A	0
B	10
C	01
D	00
E	1
F	11

Mauvais exemple

CAFE
010111

Quel est le problème ?

A	0
B	10
C	01
D	00
E	1
F	11

LE HIC : savoir où un code se termine et où un nouveau code commence

Mauvais exemple

CAFE
010111

Quel est le problème ?

A	0
B	10
C	01
D	00
E	1
F	11

LE HIC : savoir où un code se termine et où un nouveau code commence

SOLUTION : le codage d'une "lettre" n'est jamais un préfixe du codage d'une autre "lettre"

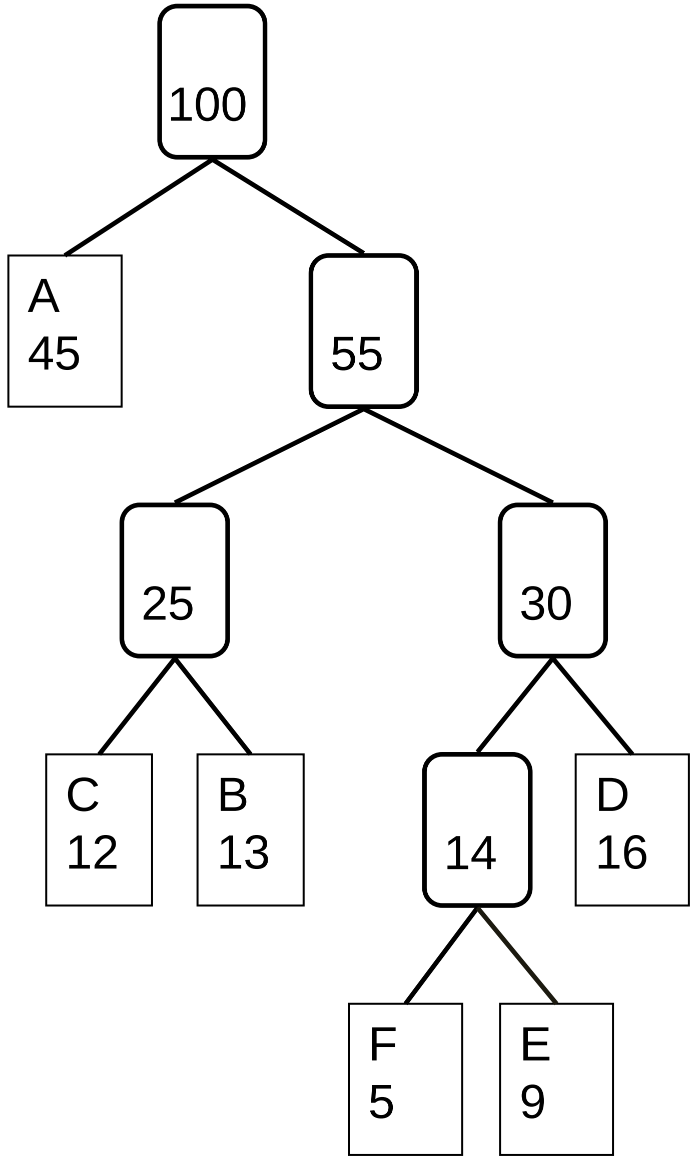
L'algorithme de Huffman construit un arbre binaire représentant le codage de chaque « lettre »

Exemple:

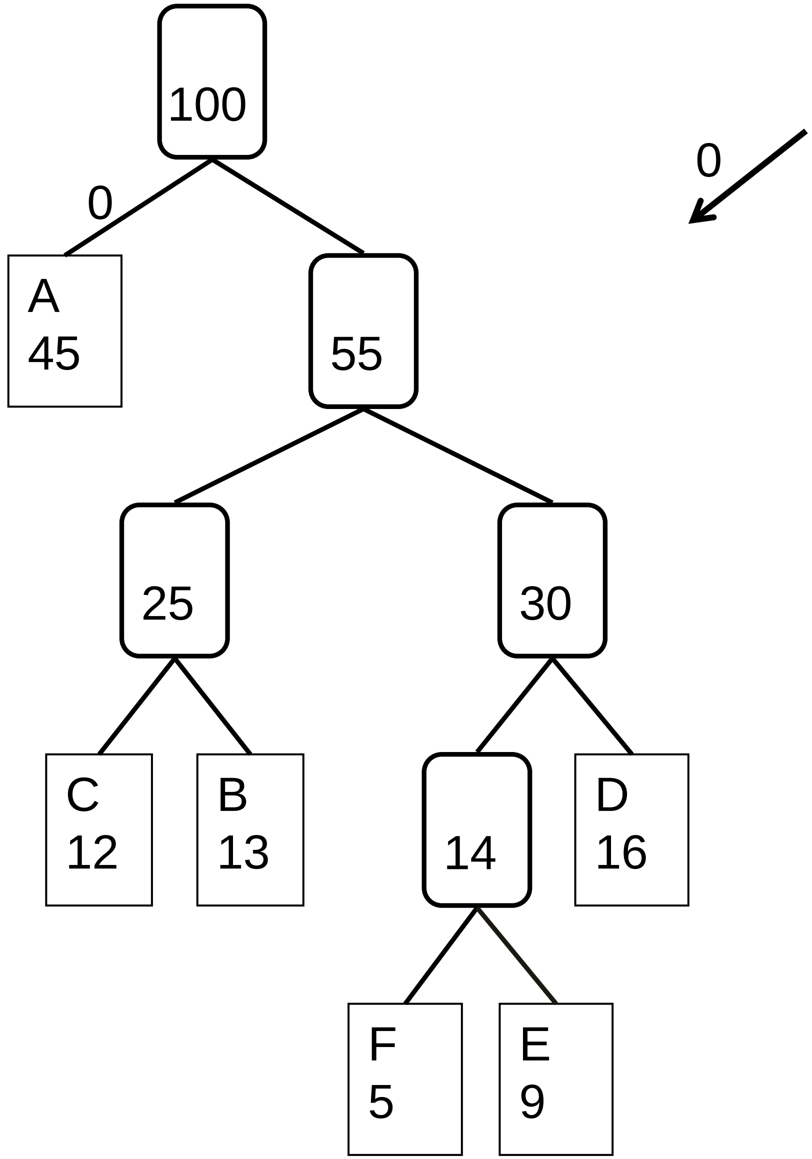
Supposons que notre texte ne contient que les lettres A, B, C, D, E et F et qu'on a la map des fréquences suivantes :

A	B	C	D	E	F
45	13	12	16	9	5

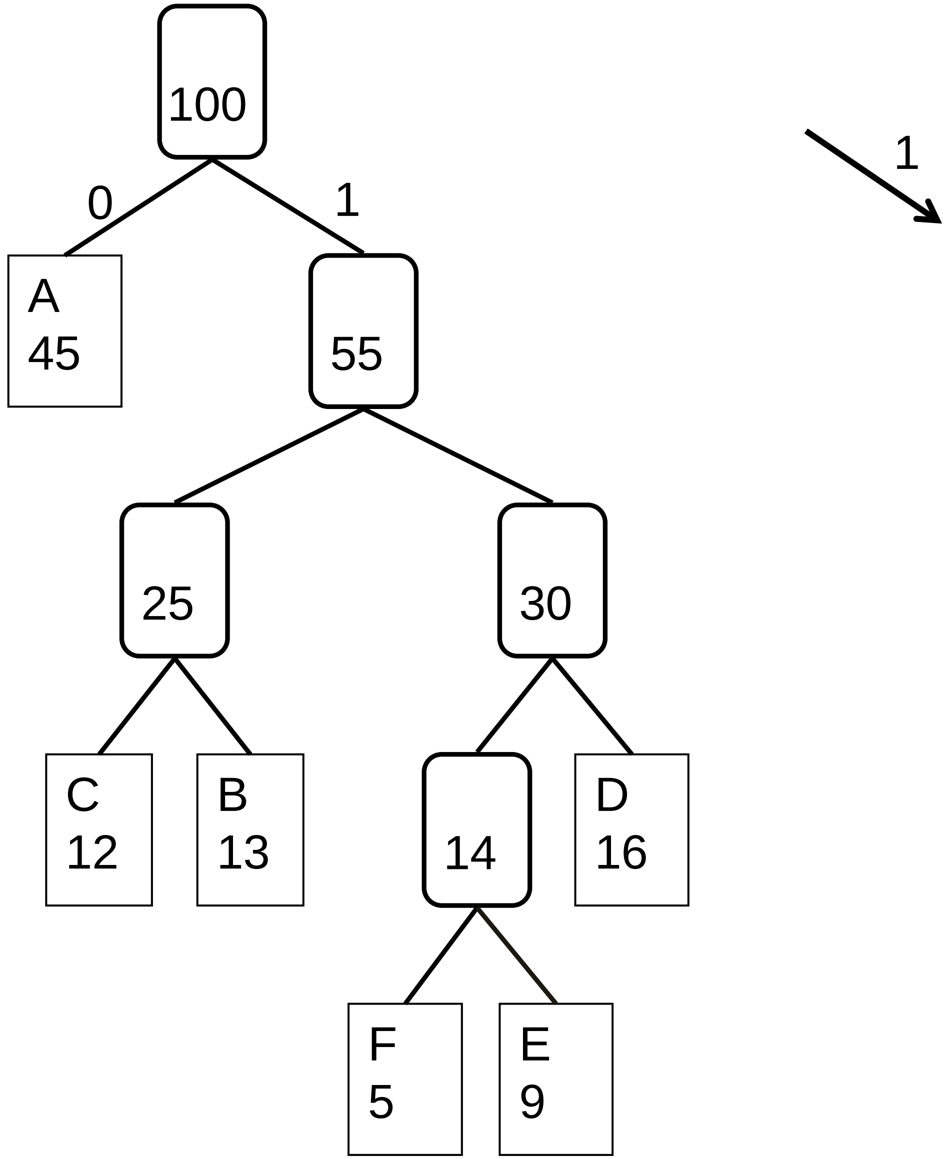
Voici l'arbre binaire construit par l'algorithme :



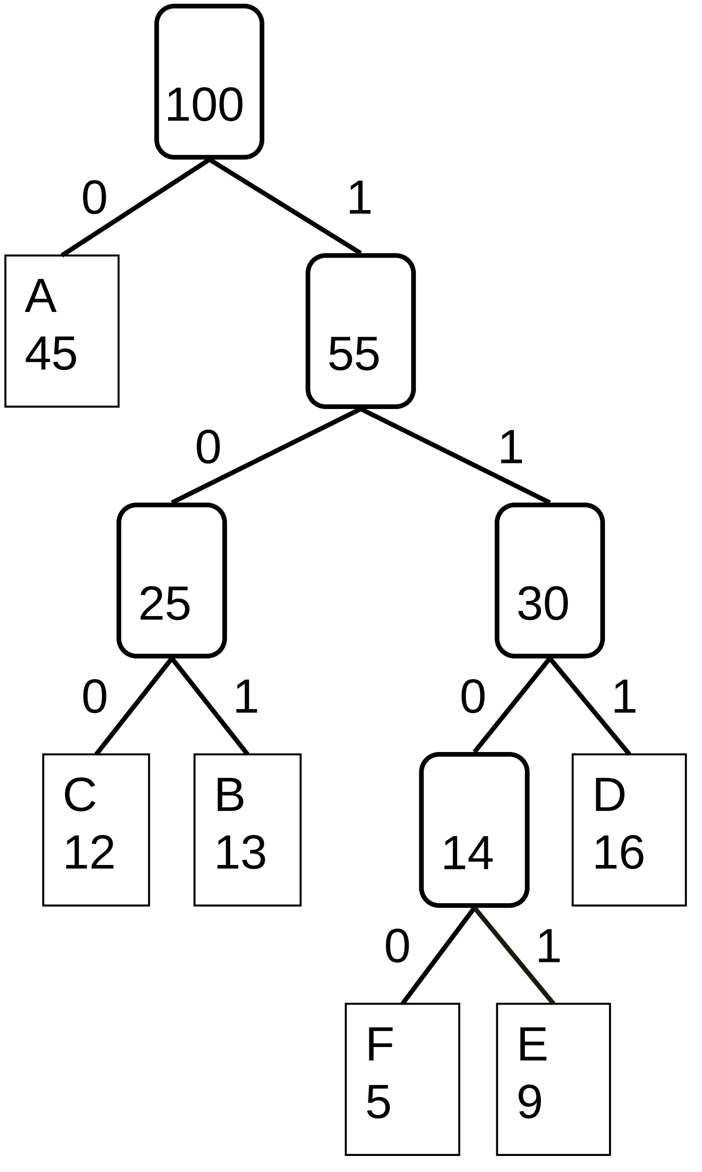
Voici l'arbre binaire construit par l'algorithme :



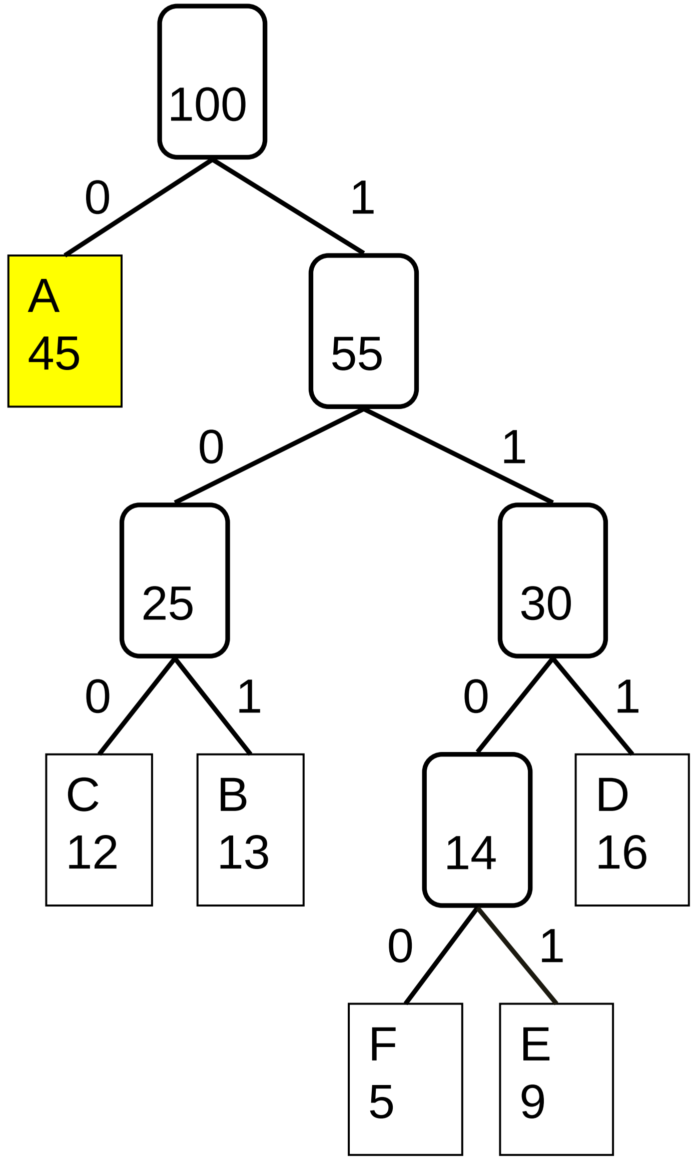
Voici l’arbre binaire construit par l’algorithme :



Voici l’arbre binaire construit par l’algorithme :

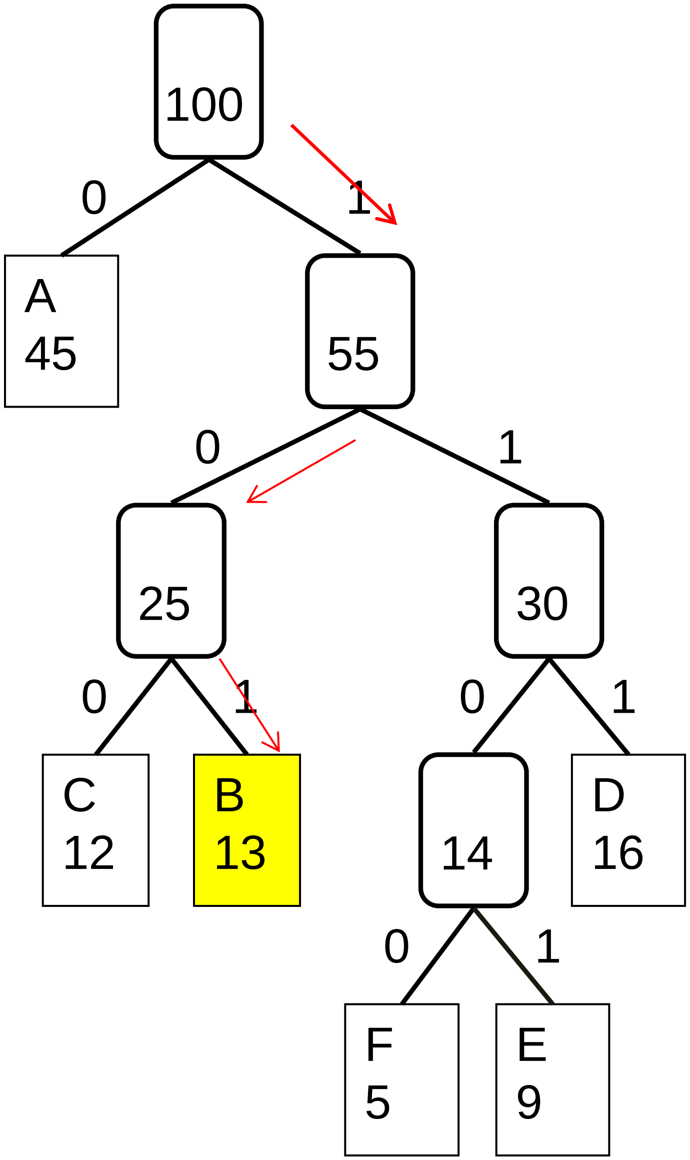


CODAGE :



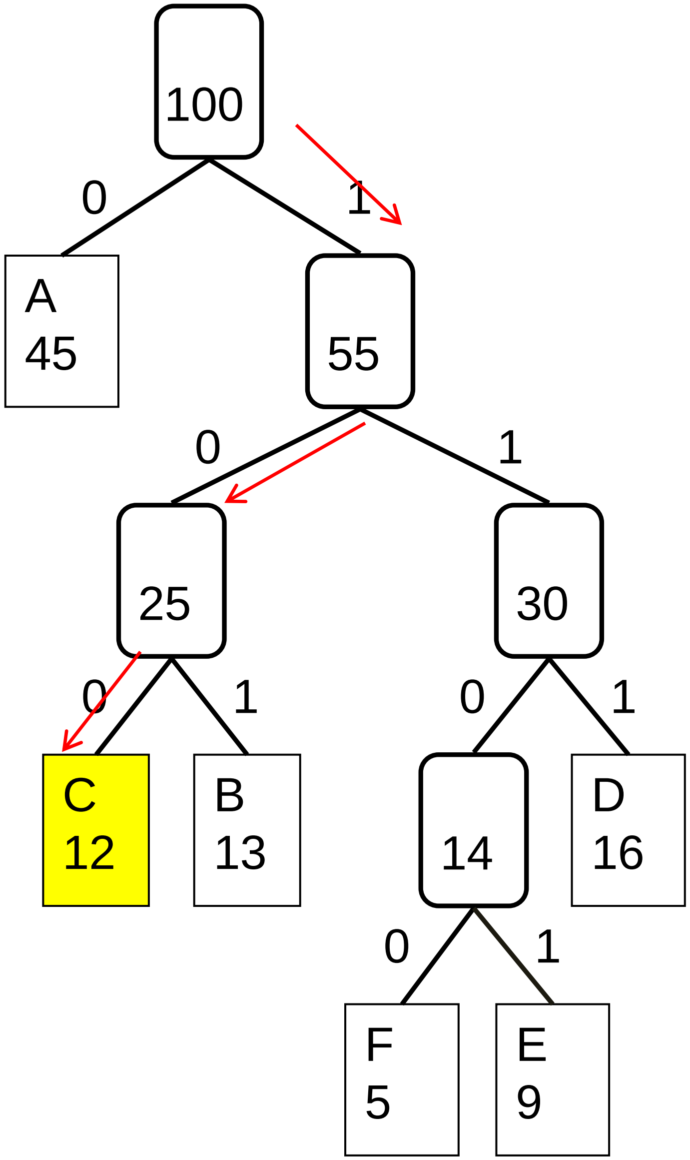
A 0

CODAGE :



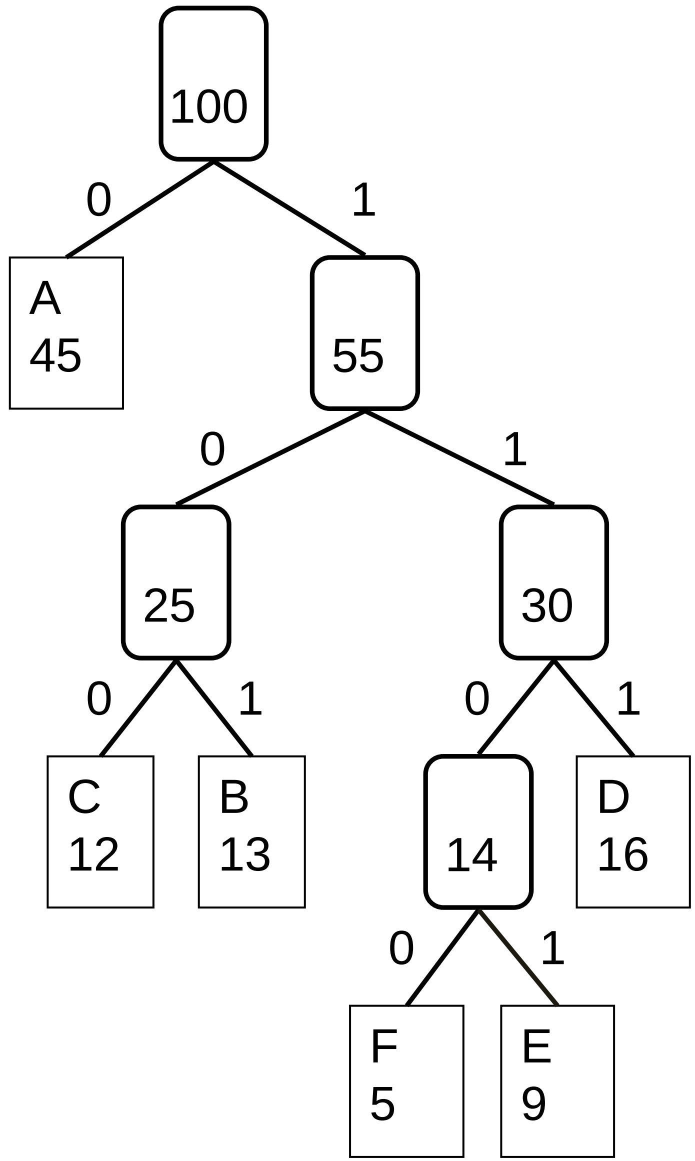
A	0
B	101

CODAGE :



A	0
B	101
C	100

CODAGE :



A	0
B	101
C	100
D	111
E	1101
F	1100

CODAGE :

A	0
B	101
C	100
D	111
E	1101
F	1100

CAFE

CODAGE :

A	0
B	101
C	100
D	111
E	1101
F	1100

CAFE

100

CODAGE :

A	0
B	101
C	100
D	111
E	1101
F	1100

CAFE

1000

CODAGE :

A	0
B	101
C	100
D	111
E	1101
F	1100

CAFE

10001100

CODAGE :

A	0
B	101
C	100
D	111
E	1101
F	1100

CAFE

10001100 1101

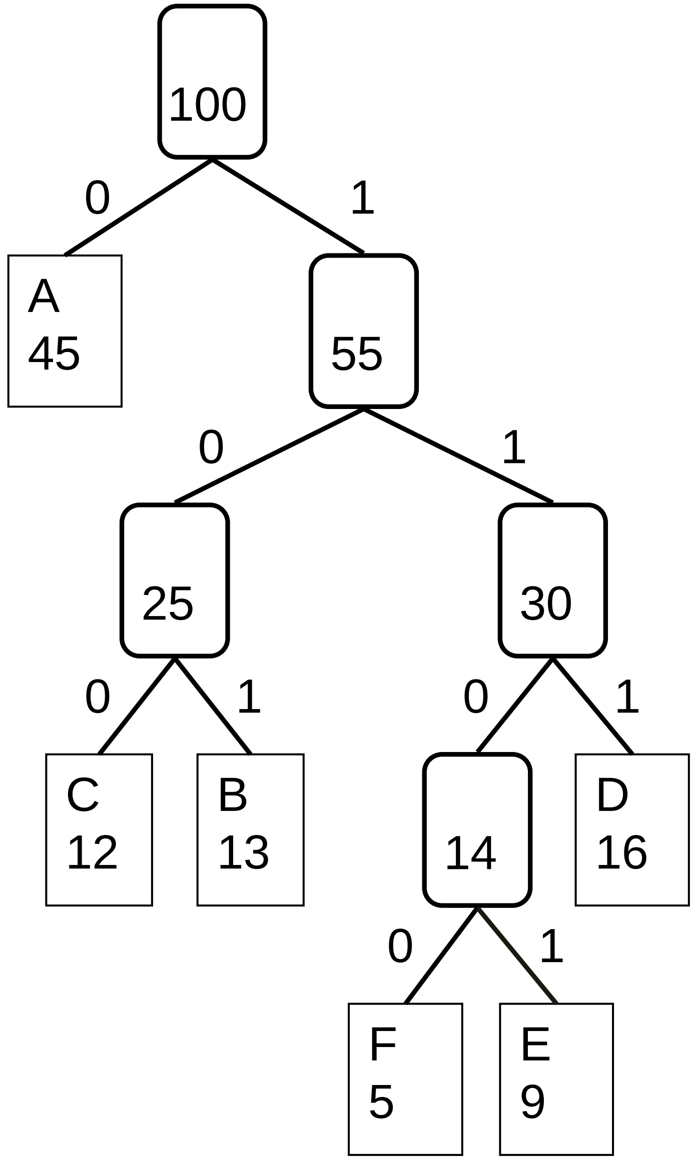
CODAGE :

A	0
B	101
C	100
D	111
E	1101
F	1100

CAFE

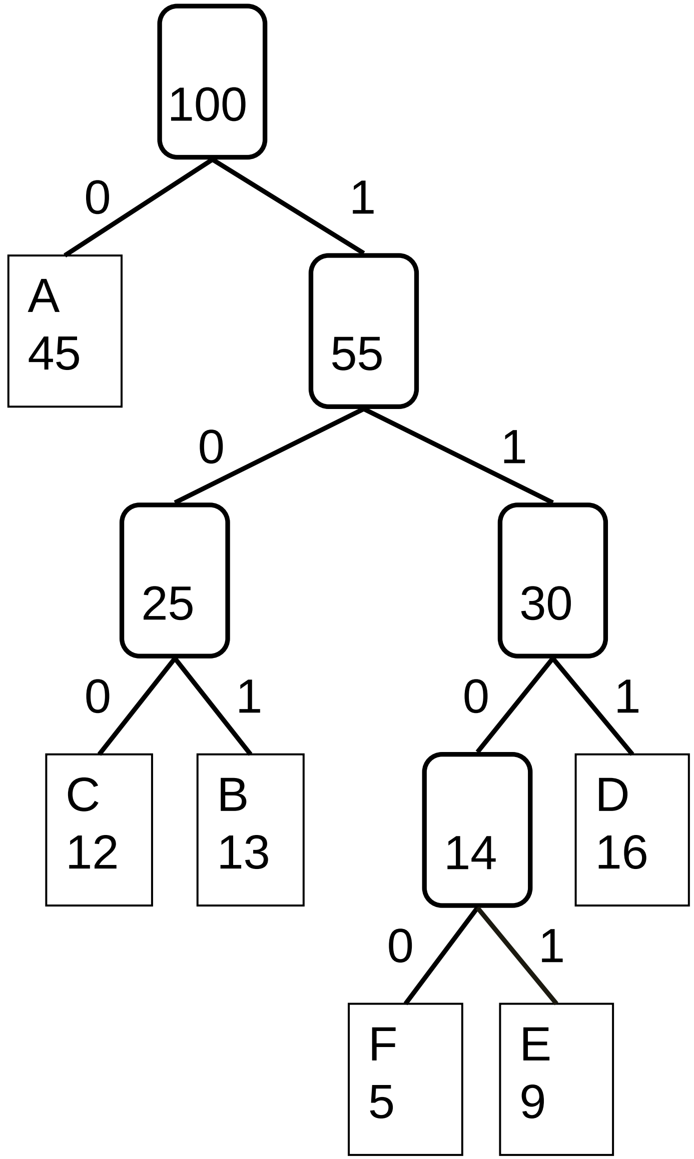
100011001101

DECODAGE :



A	0
B	101
C	100
D	111
E	1101
F	1100

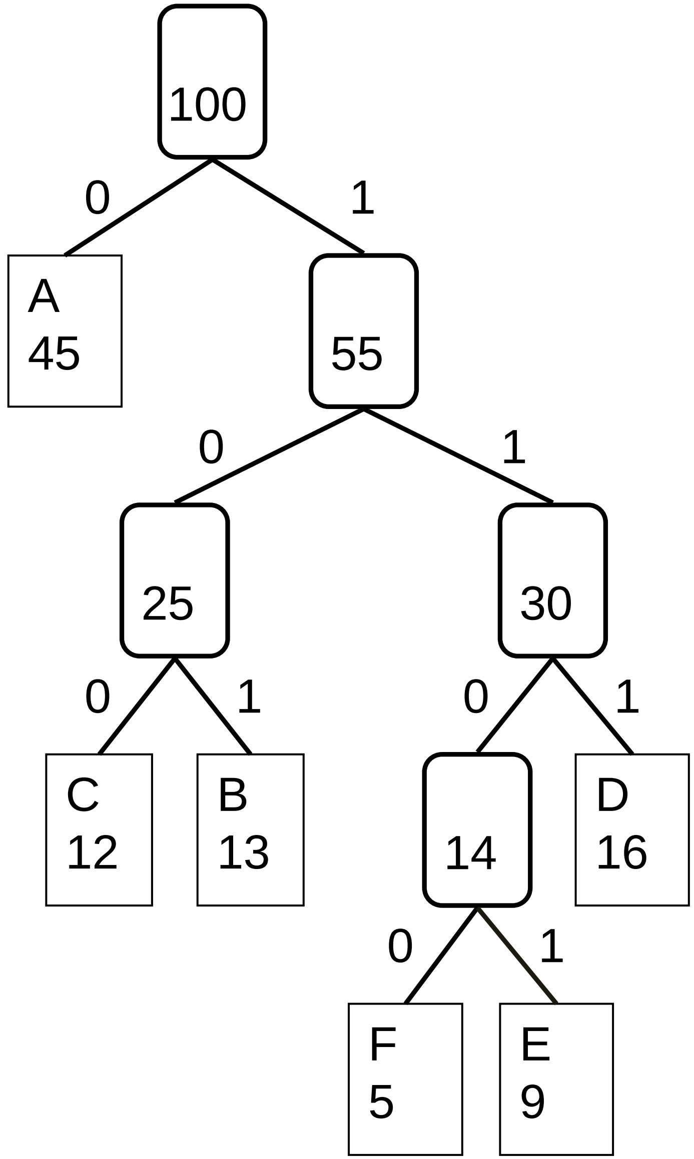
DECODAGE :



A	0
B	101
C	100
D	111
E	1101
F	1100

11111011000

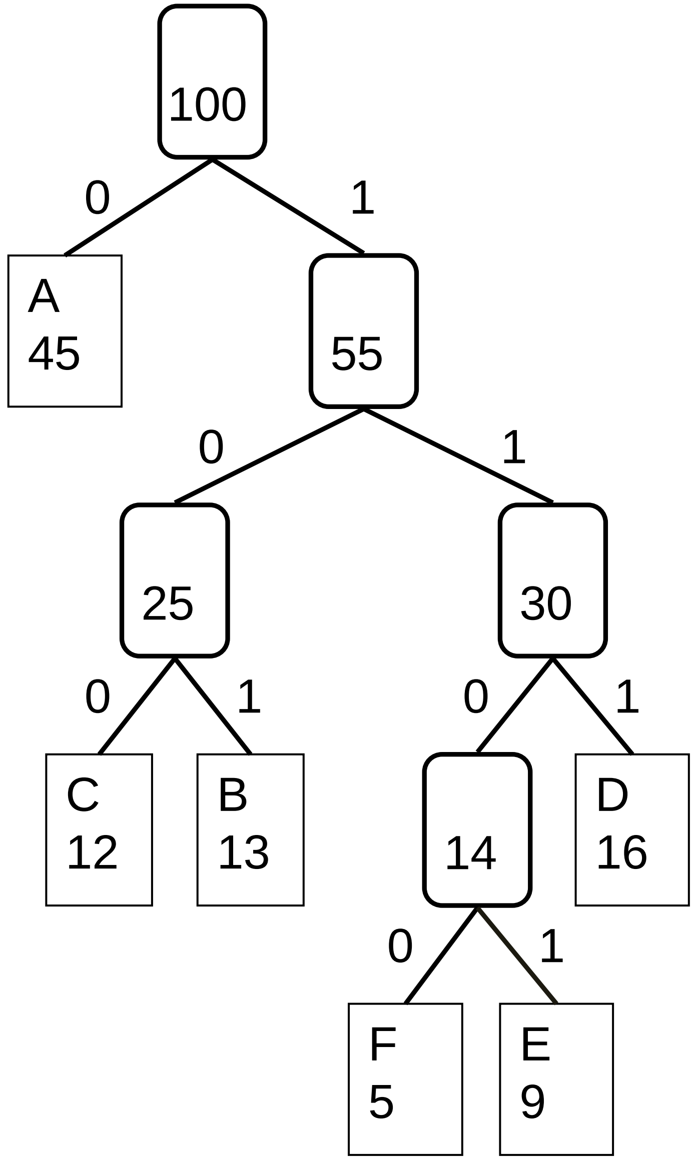
DECODAGE :



A	0
B	101
C	100
D	111
E	1101
F	1100

11111011000

DECODAGE :



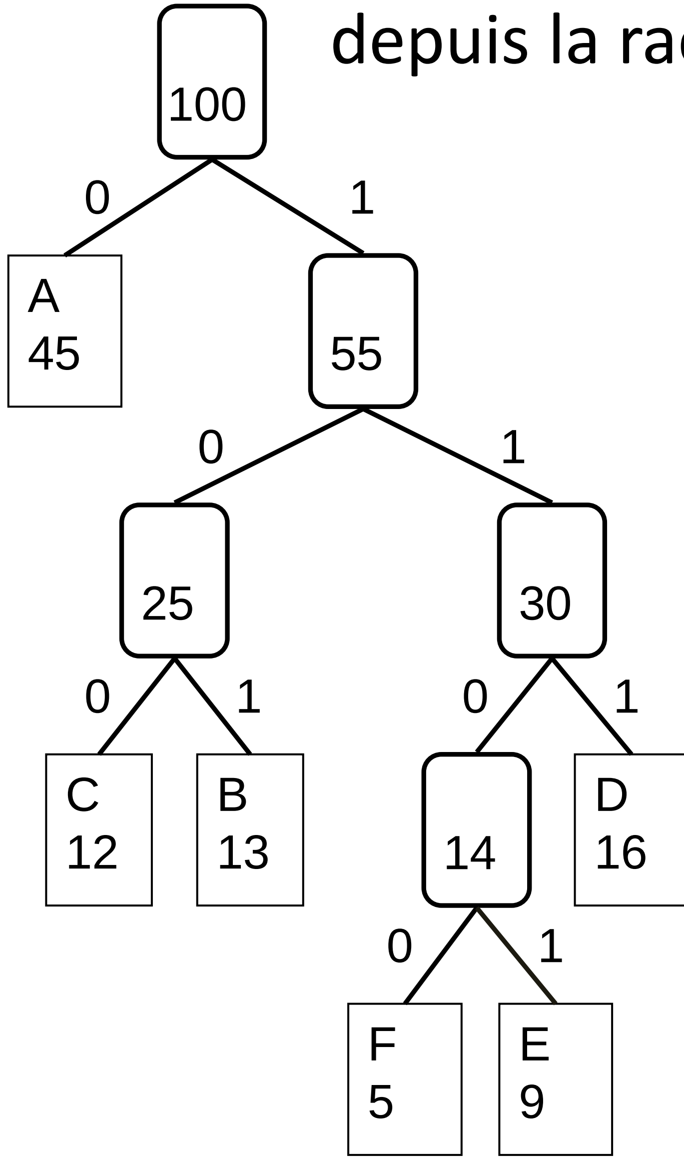
11111011000

DECODAGE :

Plusieurs parcours de l'arbre depuis la racine jusqu'une feuille

0 → à gauche

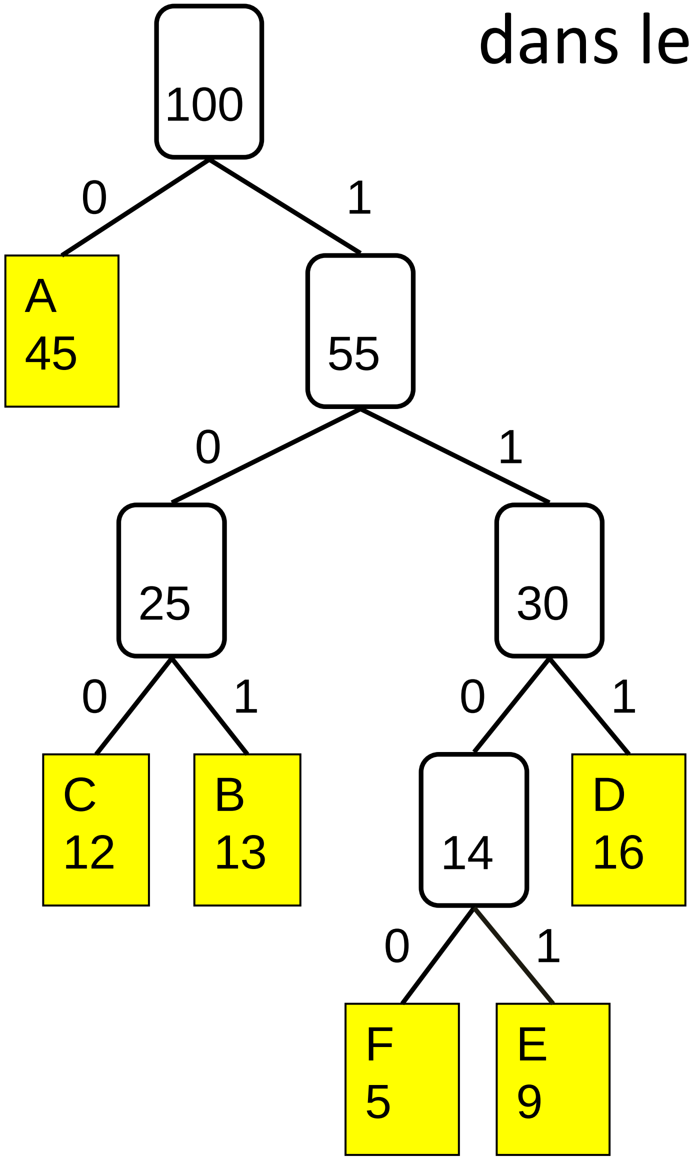
1 → à droite



11111011000

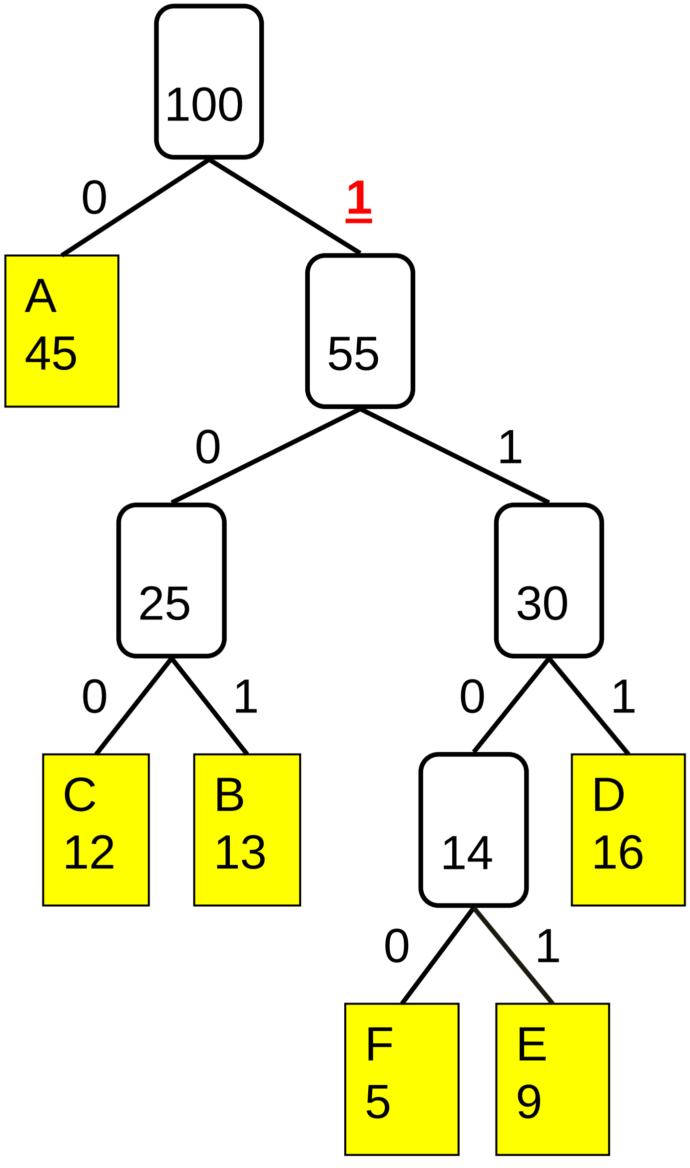
DECODAGE :

Les caractères se trouvent dans les feuilles!



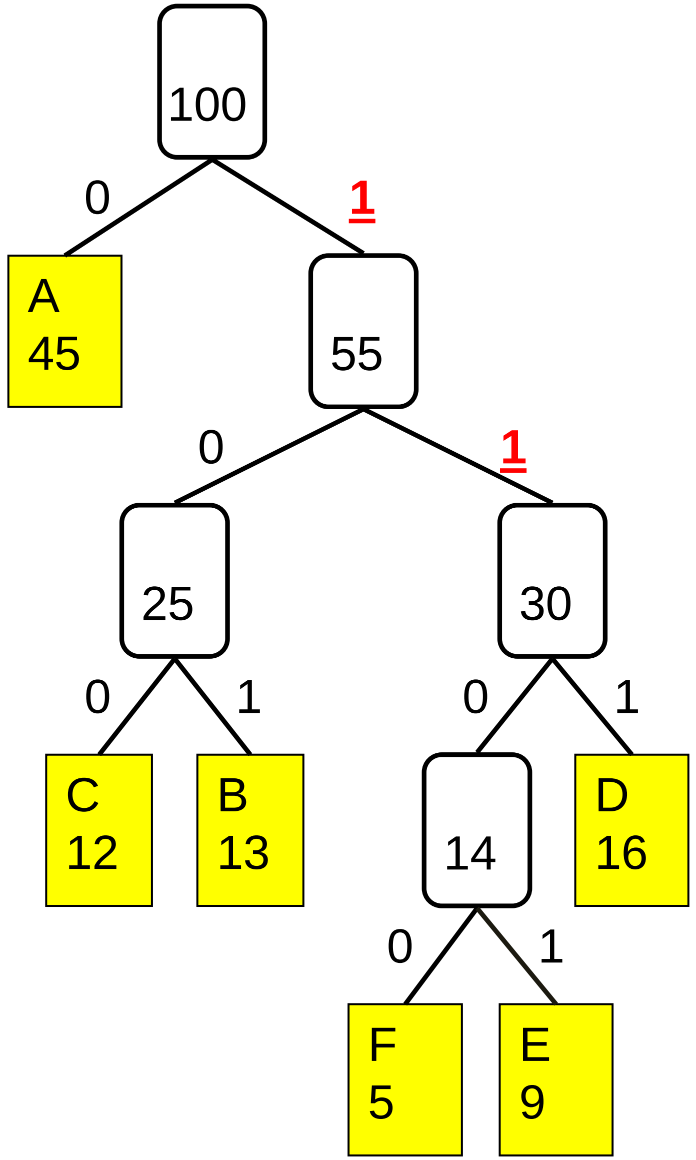
11111011000

DECODAGE :



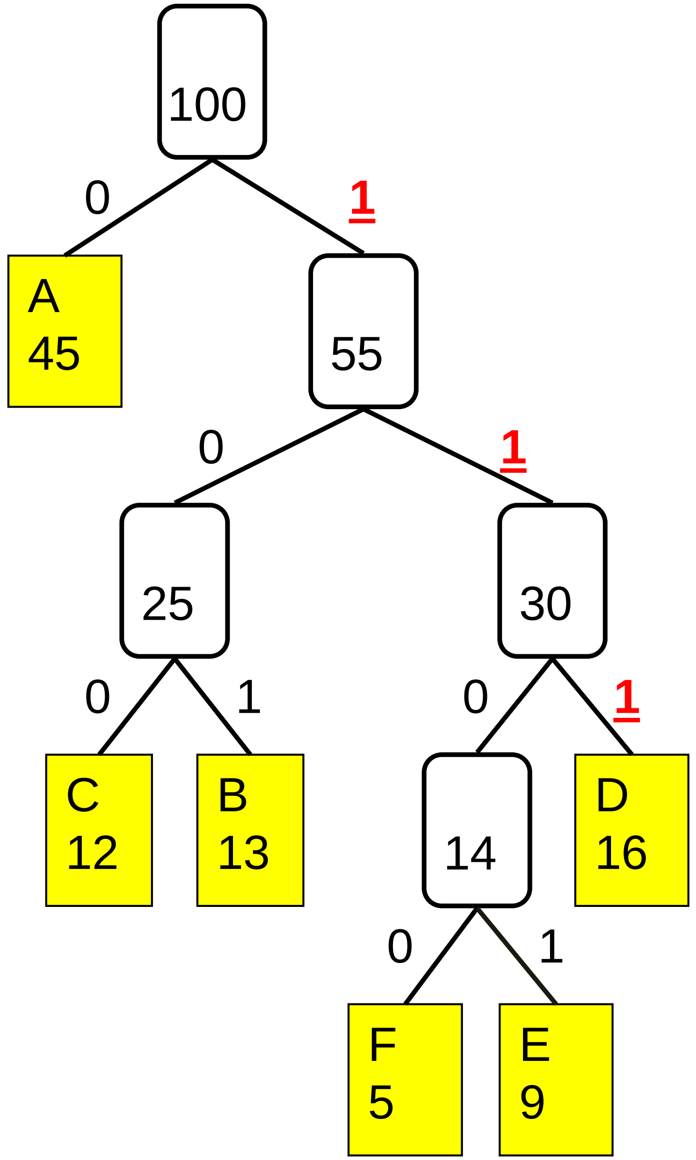
1111011000

DECODAGE :



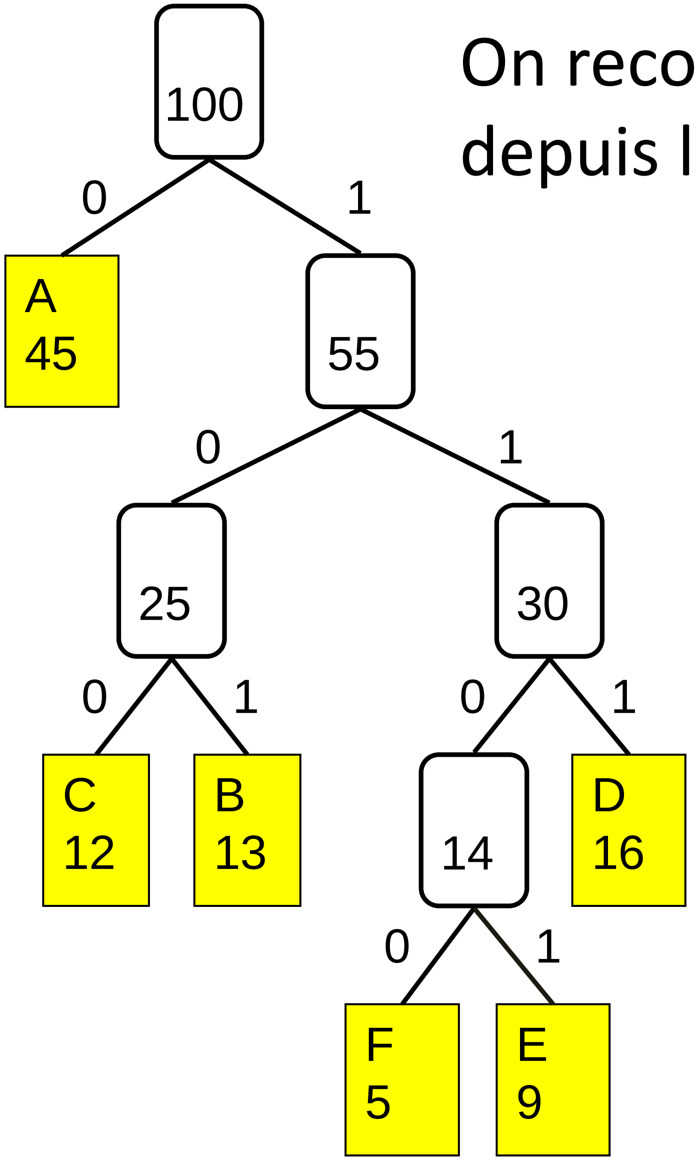
11111011000

DECODAGE :



<u>111</u>11011000

DECODAGE :

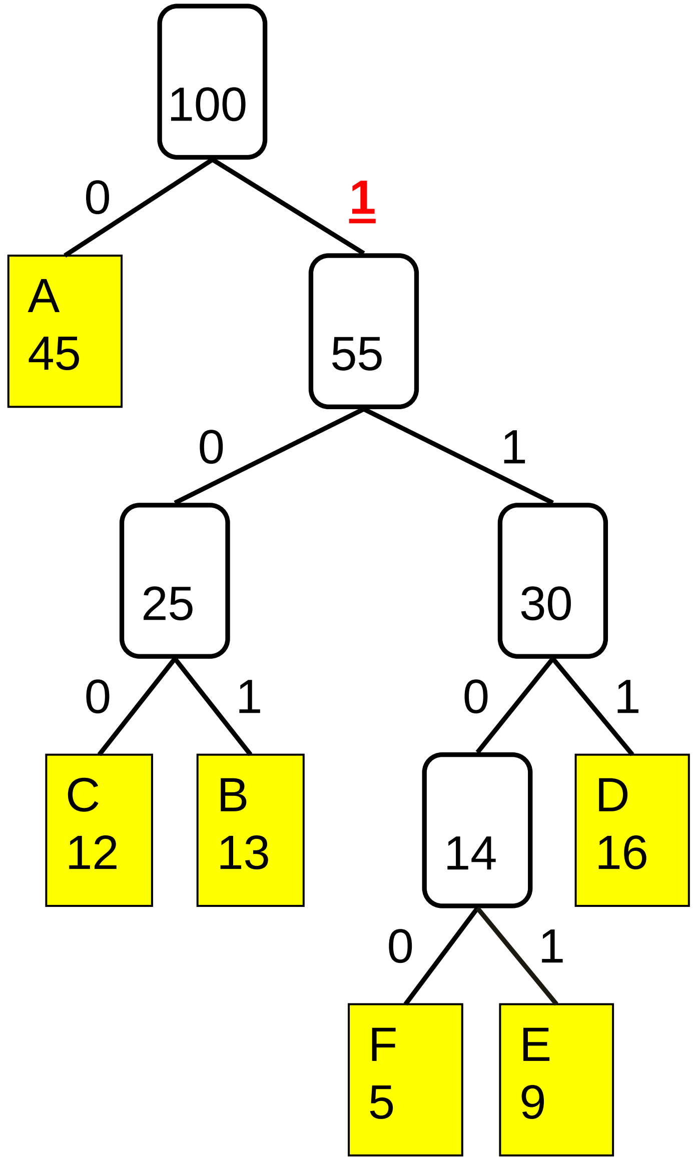


On recommence un parcours depuis la racine

<u>111</u>11011000

D

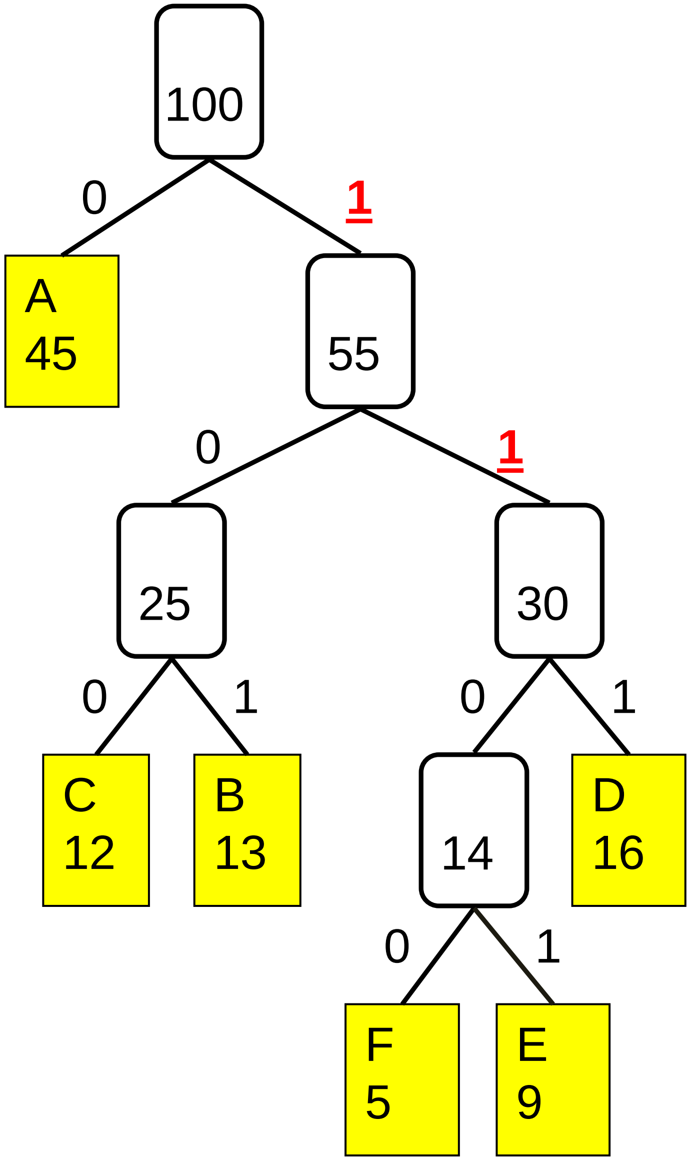
DECODAGE :



111<u>1</u>1011000

D

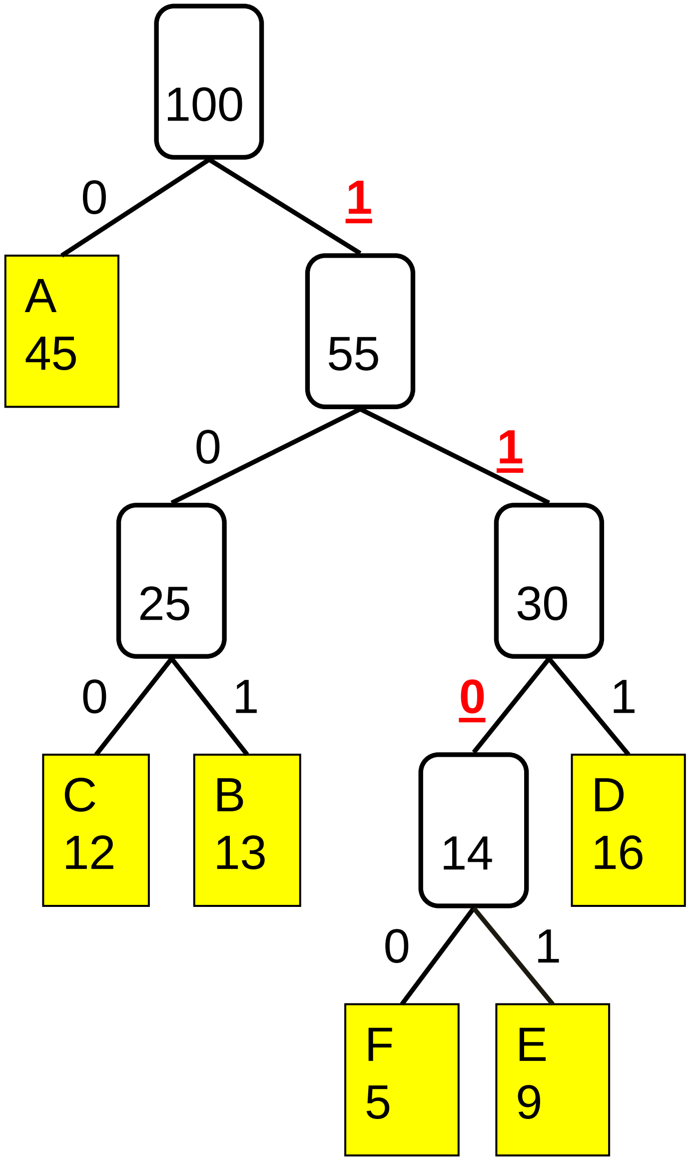
DECODAGE :



111110111000

D

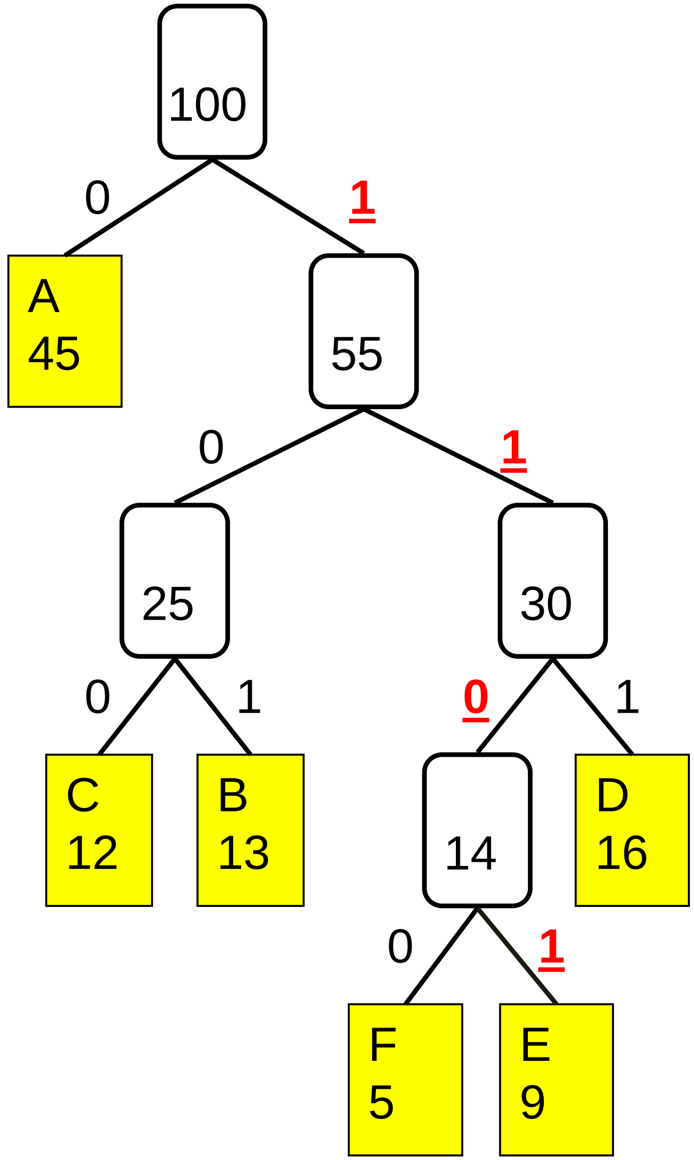
DECODAGE :



111<u>110</u>11000

D

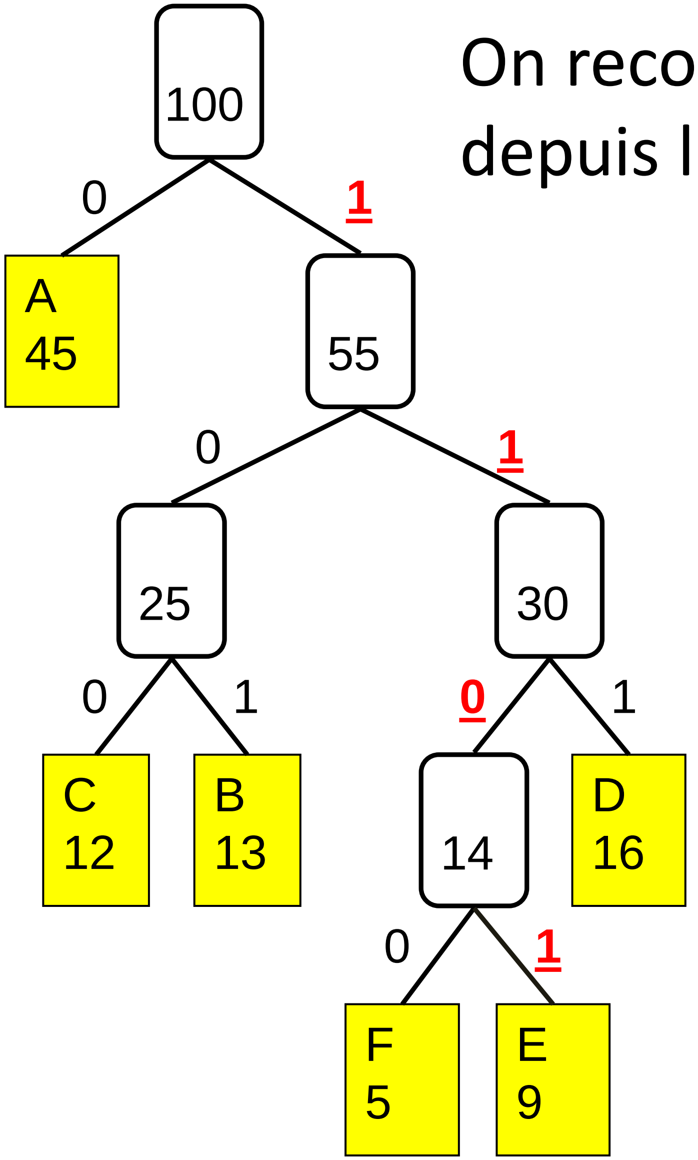
DECODAGE :



111<u>1101</u>1000

D

DECODAGE :

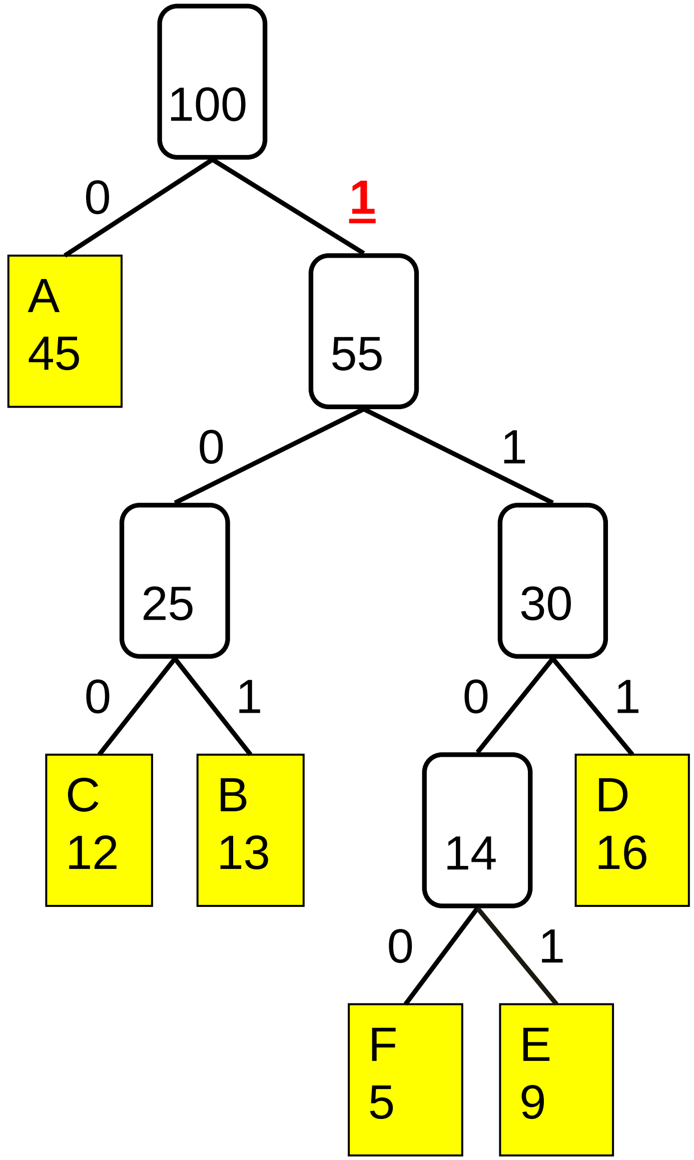


On recommence un parcours depuis la racine

11111011000

D E

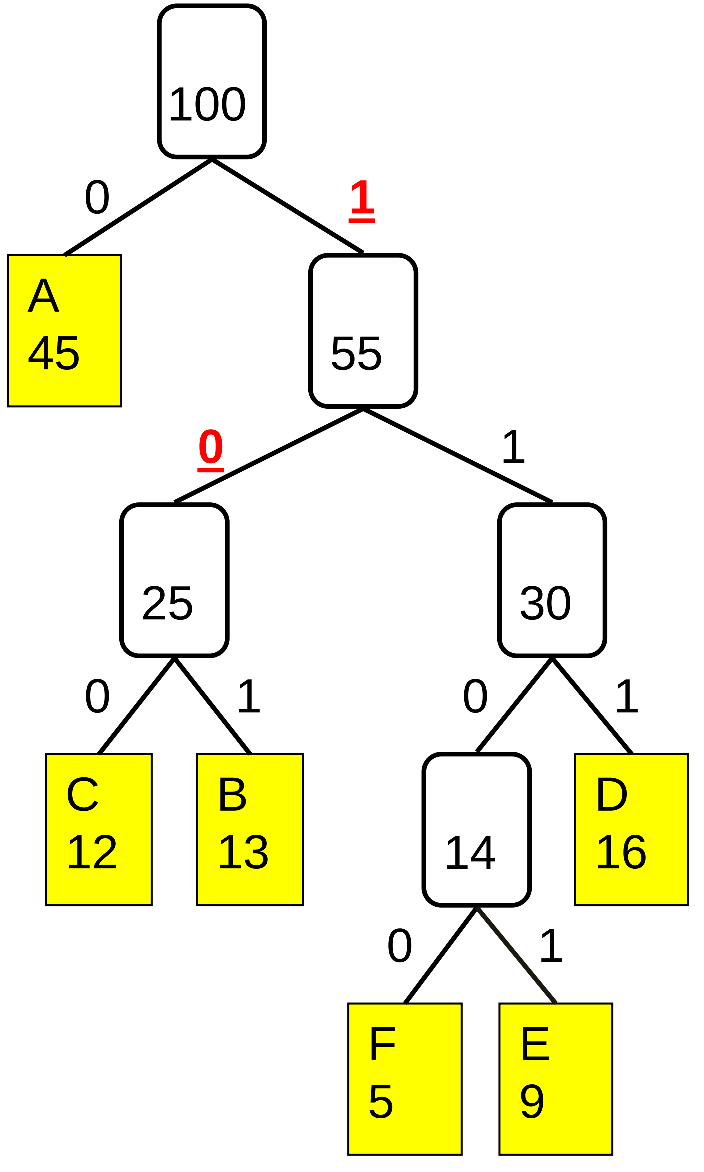
DECODAGE :



11111011000

D E

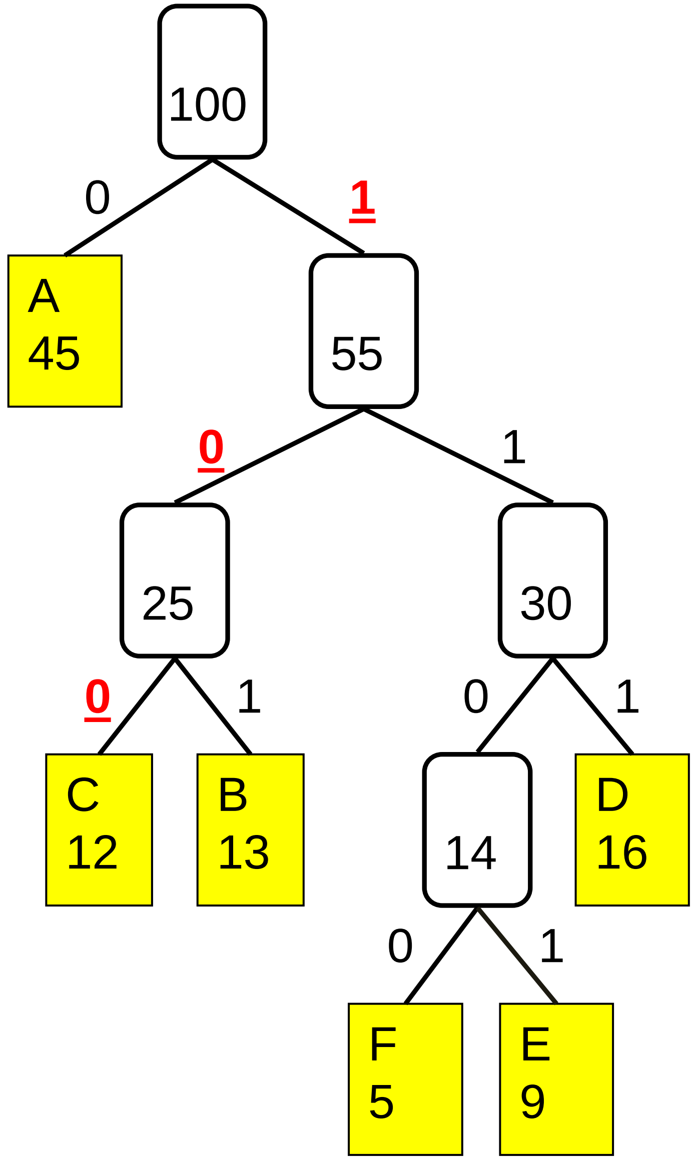
DECODAGE :



1111101<u>10</u>00

D E

DECODAGE :

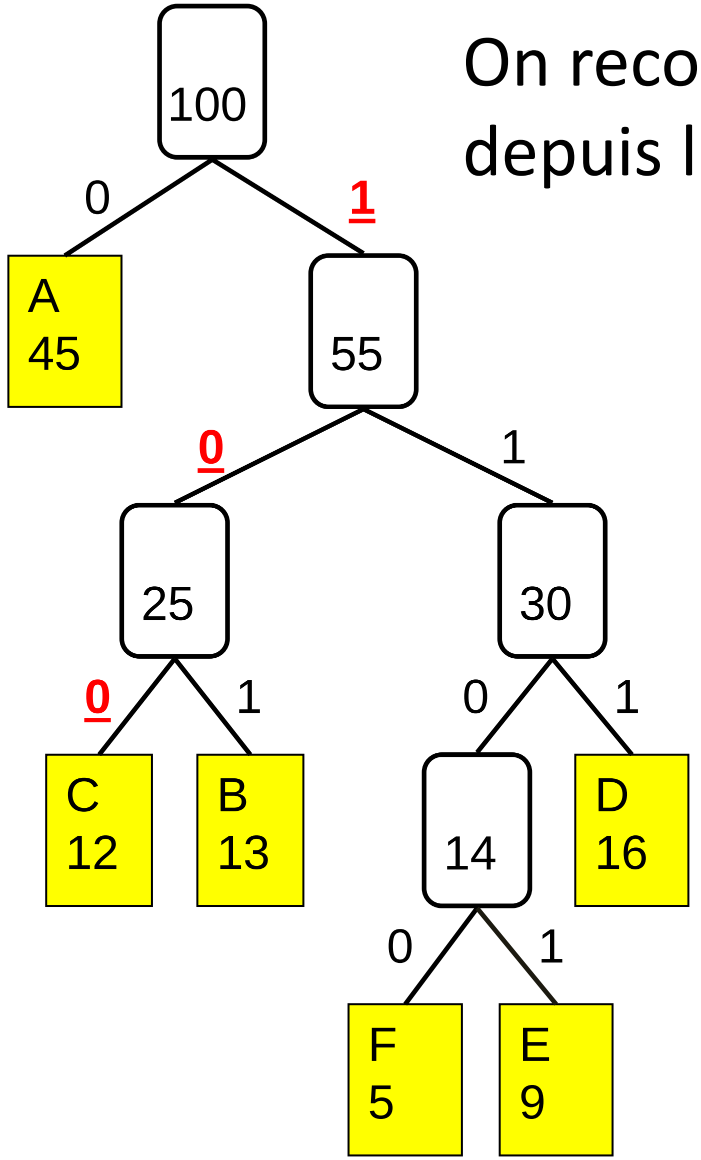


11111011000

D E

DECODAGE :

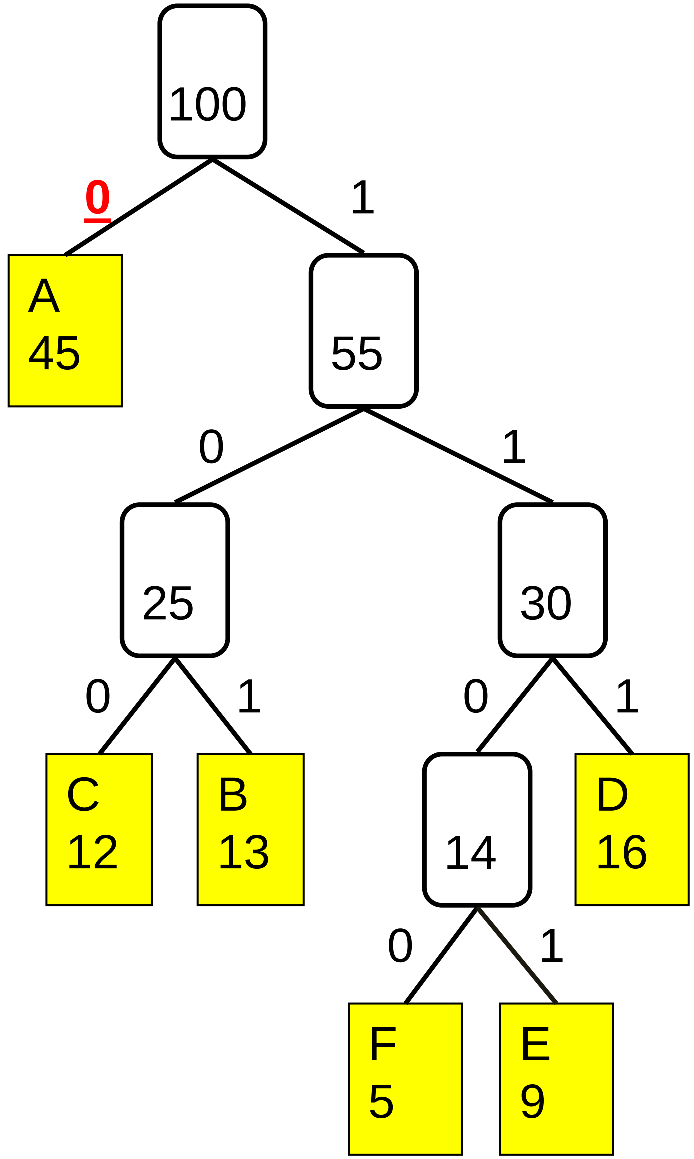
On recommence un parcours depuis la racine



111110110<u>100</u>0

D E C

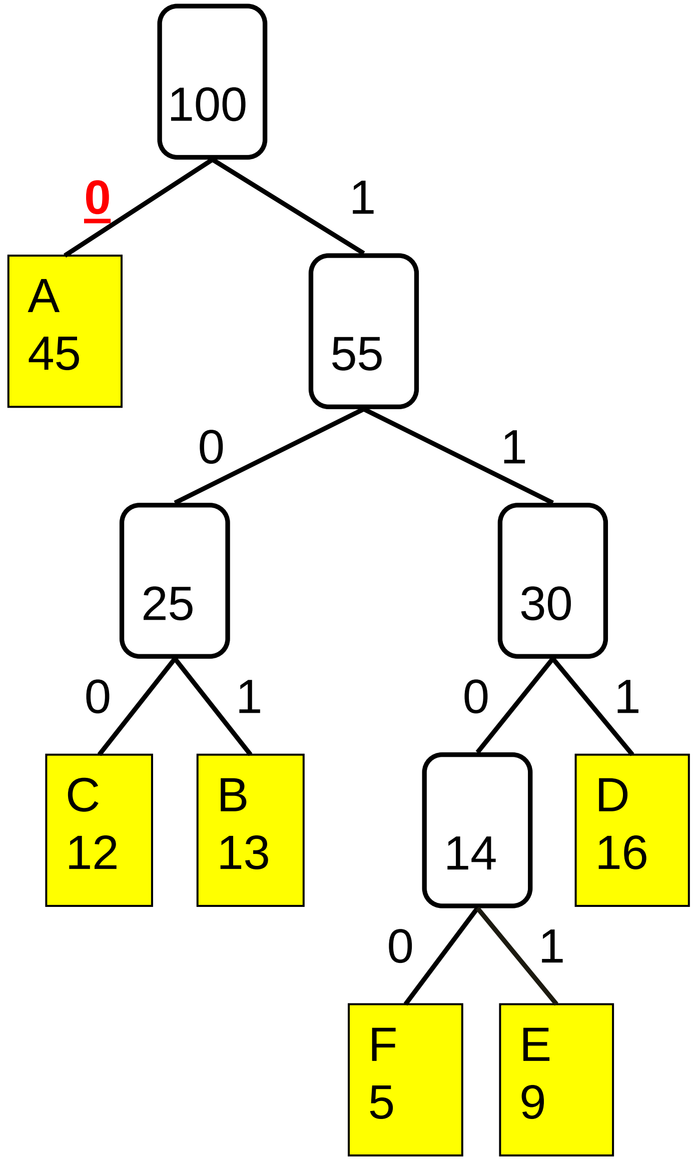
DECODAGE :



11111011000

D E C

DECODAGE :



1111101100<u>0</u>

D E C A

ALGORITHME :

L'algorithme de Huffman utilise une file de priorité.

Cette file de priorité contient des arbres.

La priorité correspond à la fréquence.

ALGORITHME :

L'algorithme de Huffman utilise une file de priorité.

Cette file de priorité contient des arbres.

La priorité correspond à la fréquence.

Dans les figures suivantes, la file est triée pour faciliter la compréhension de l'algorithme!

F 5	E 9	C 12	B 13	D 16	A 45

Etape initiale

n insère()

F	E	C	B	D	A
5	9	12	13	16	45

A chaque étape :

2 x supprimeMax()

Σ fréquences

insère()

E	C	B	D	A
9	12	13	16	45

Etape 1

supprimeMax()

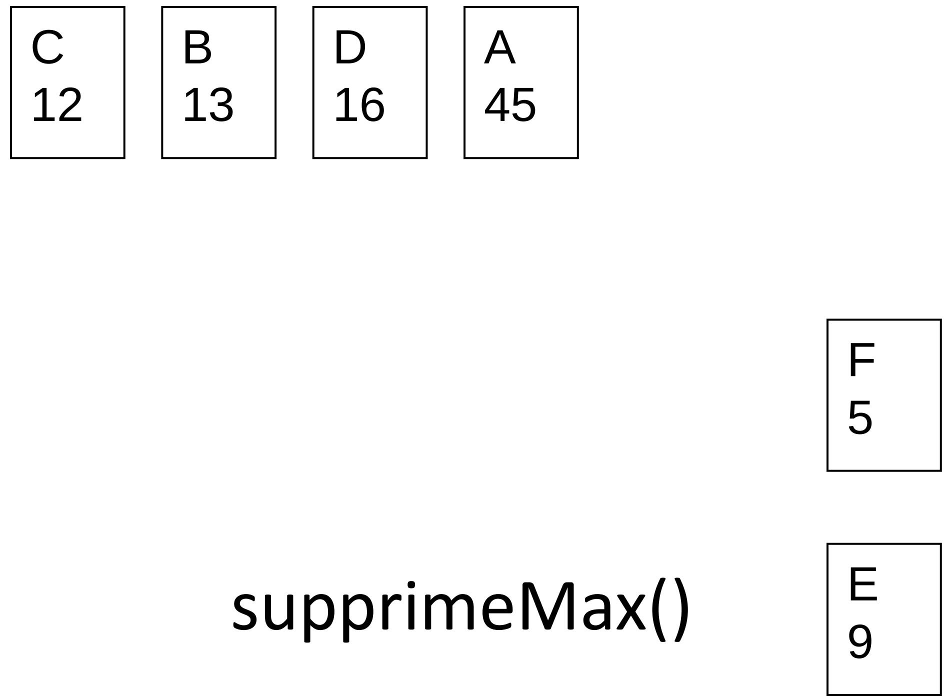
F
5

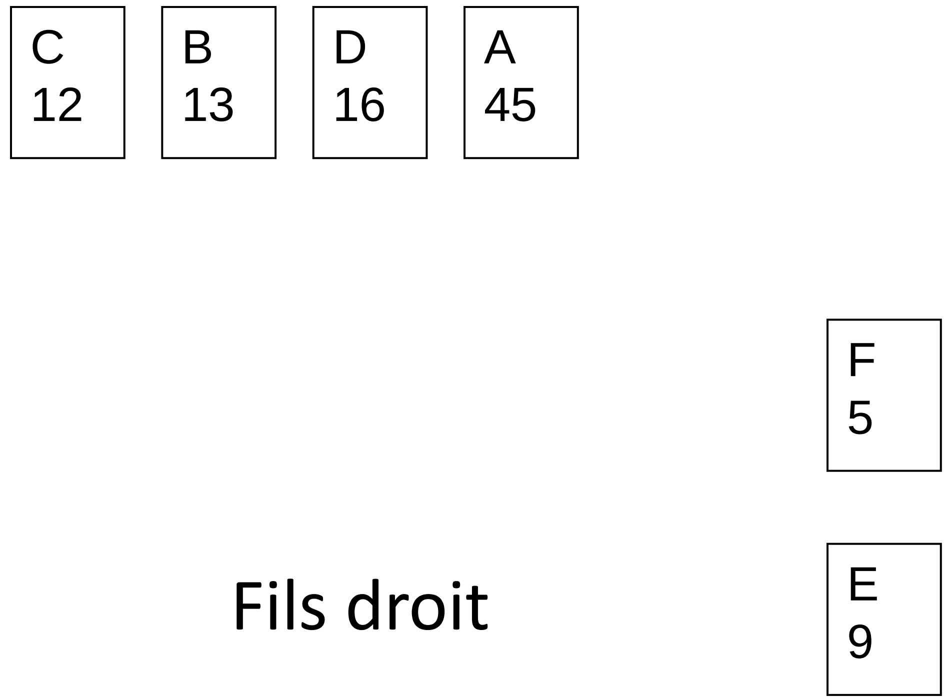
E 9 | C 12 | B 13 | D 16 | A 45

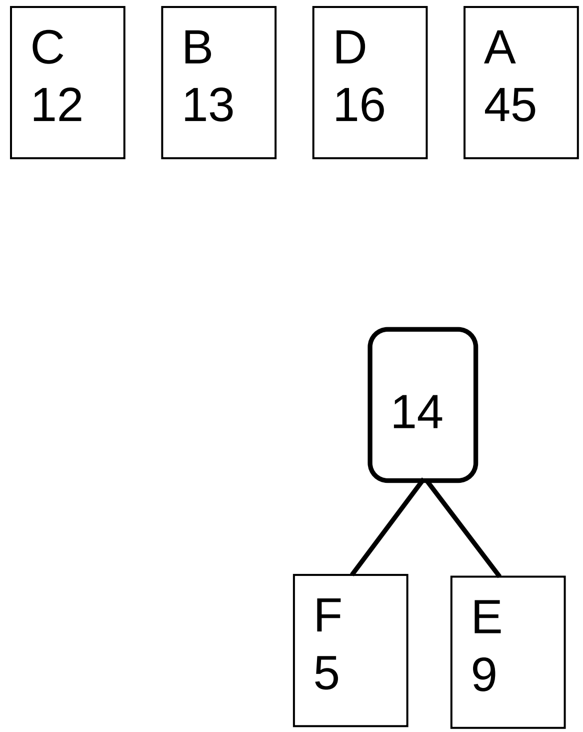
Etape 1

Fils gauche

F 5

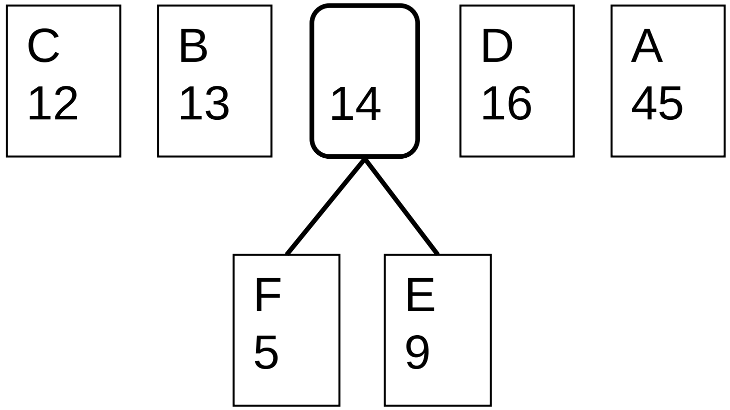

C 12
B 13
D 16
A 45
F 5
supprimeMax()
E 9


C
12
B
13
D
16
A
45
F
5
Fils droit
E
9



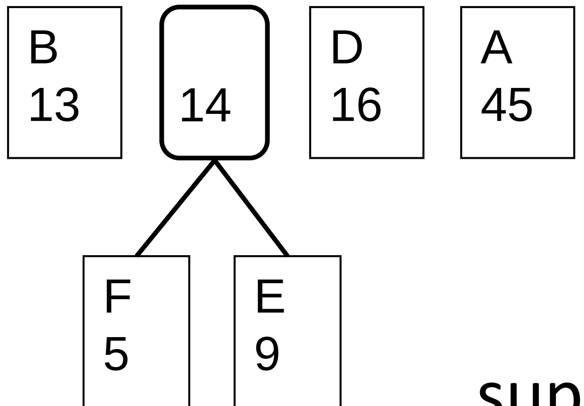
Σ des 2 fréquences

→ racine

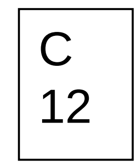


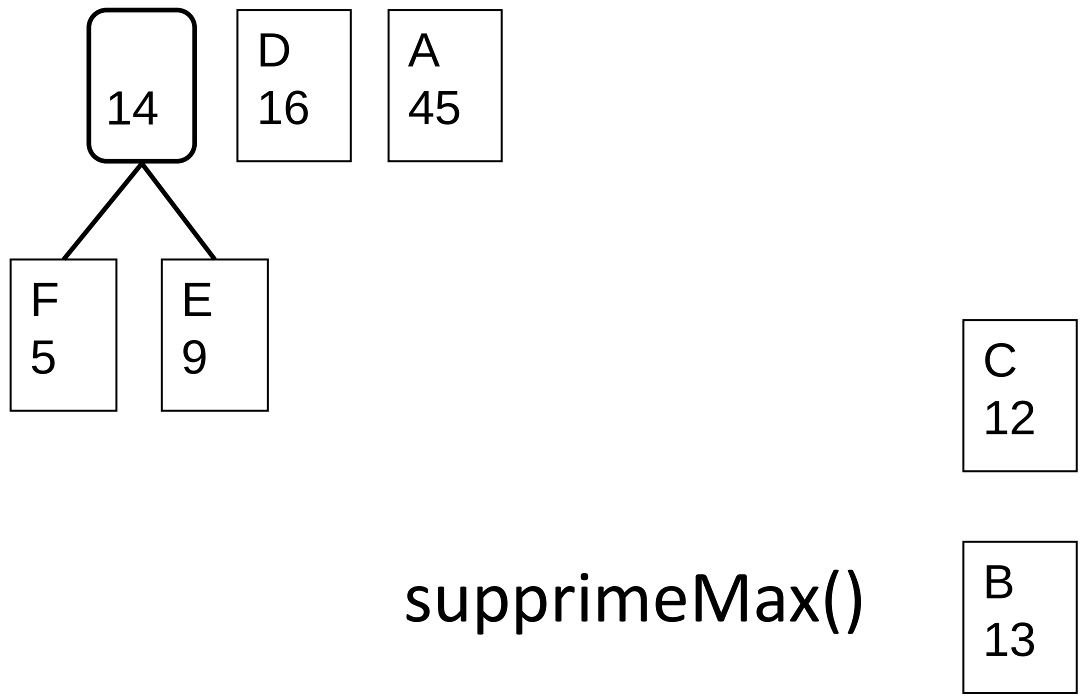
insère()

Etape 2

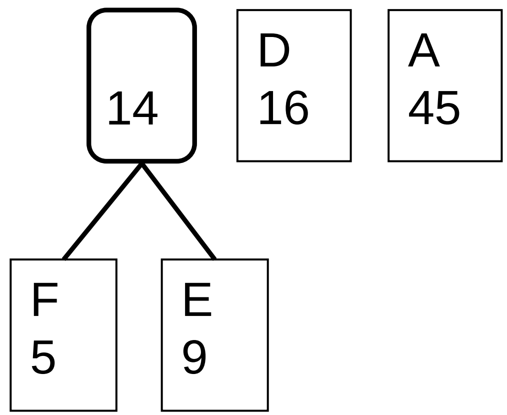


supprimeMax()

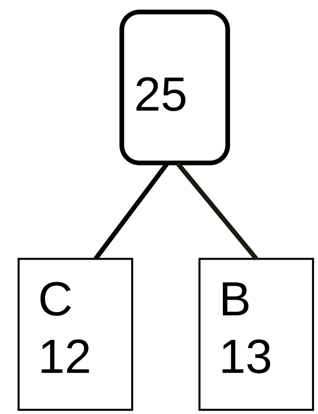


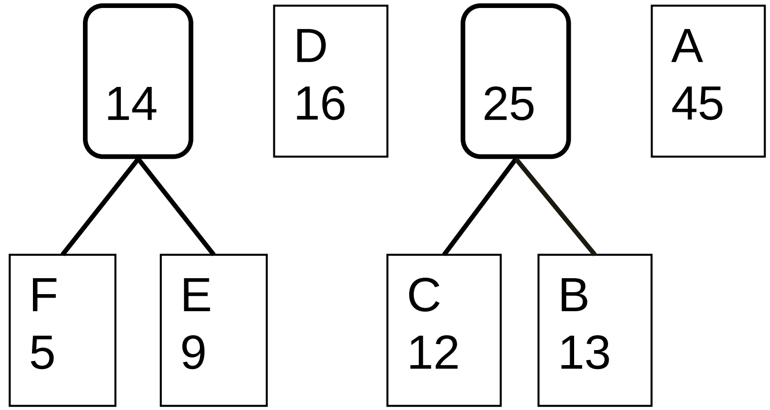


14
D
16
A
45
F
5
E
9
C
12
supprimeMax()
B
13

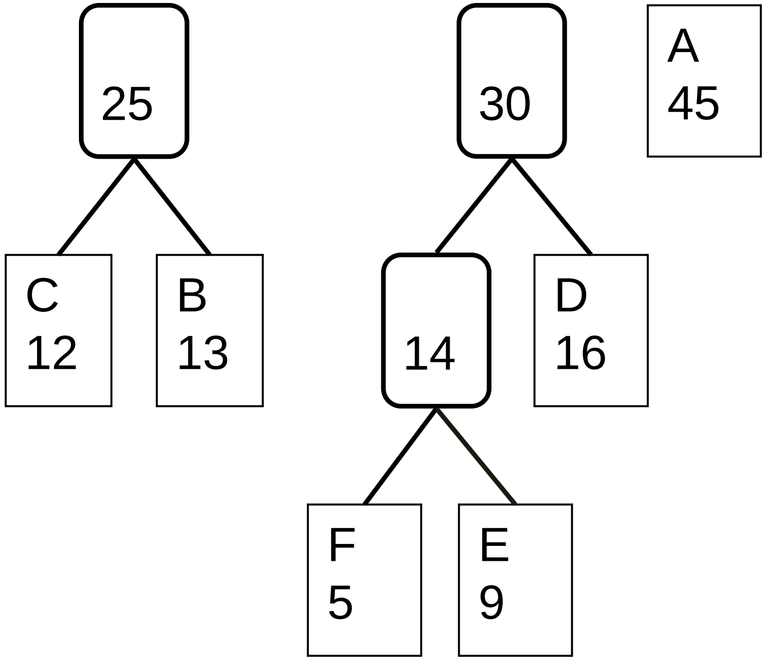


Σ des 2 fréquences

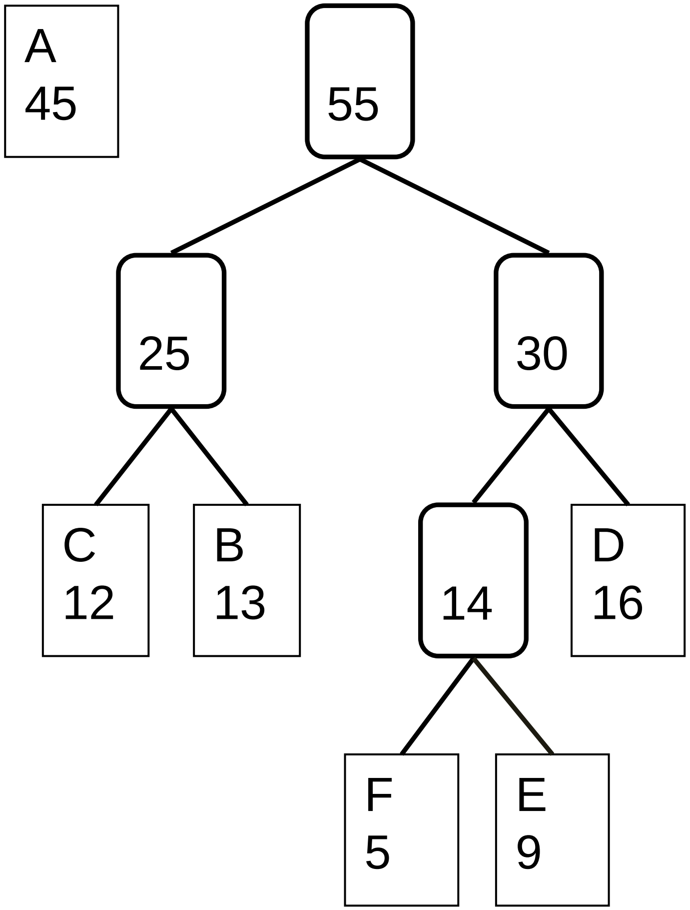




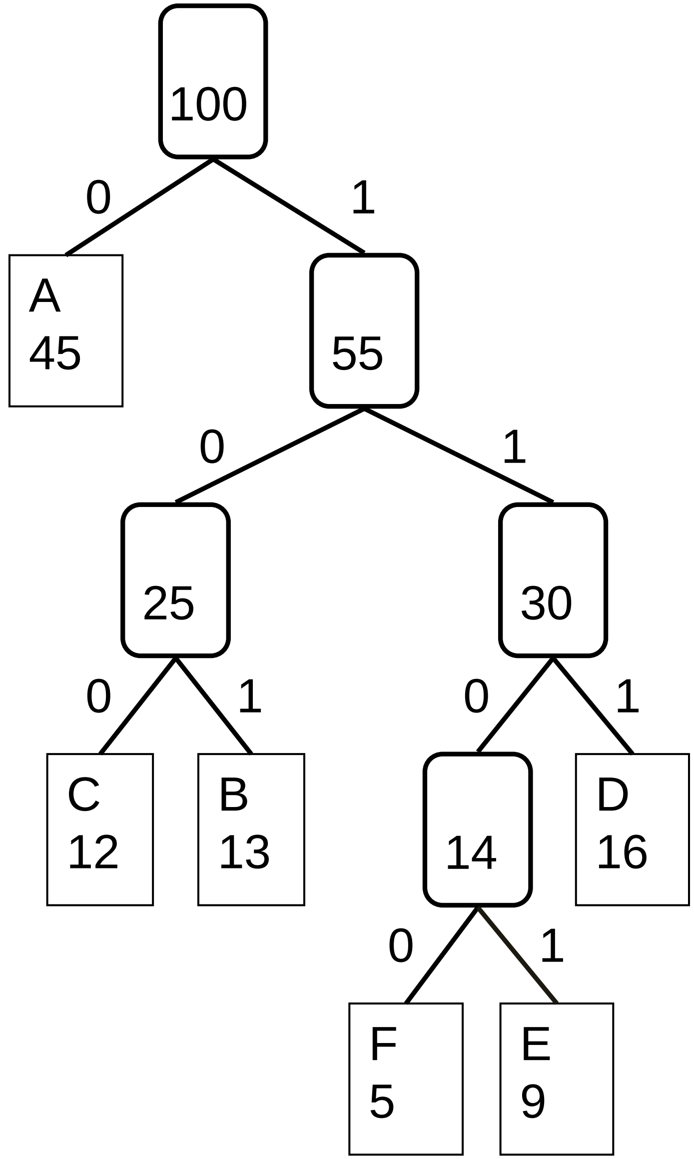
insère()



A la fin de
L’étape3



A la fin de
L’étape4



A la fin de L'étape5

Fin

PERFORMANCE :

L'utilisation d'une file de priorité donne un algorithme en $O(n*\log(n))$.

En effet les insertions et suppressions y sont en $O(\log(n))$.