U.F.R SCIENCES ET TECHNIQUES
Département d'Informatique
B.P. 1155
64013 PAU CEDEX
Téléphone secrétariat : 05.59.40.79.64
Télécopie : 05.59.40.76.54

III-TYPES DES STRUCTURES ENSEMBLISTES

I- TYPE ABSTRAIT **SET**

II- TYPE ABSTRAIT **BAG**

Un **set** désigne une collection **finie** d'objets :

- **distincts**,
- et de **même type**.

Un parking où sont garés des véhicules peut être **modélisé** à l'aide d'un **set**.

Par contre, une **file de véhicules** en attente devant un feu tricolore **ne peut pas** être adéquatement **représenté** par un **set**.

Le **groupe de mots** résultant de la segmentation d'un texte ne peut être représenté par un set : il est modélisé par un **bag**.

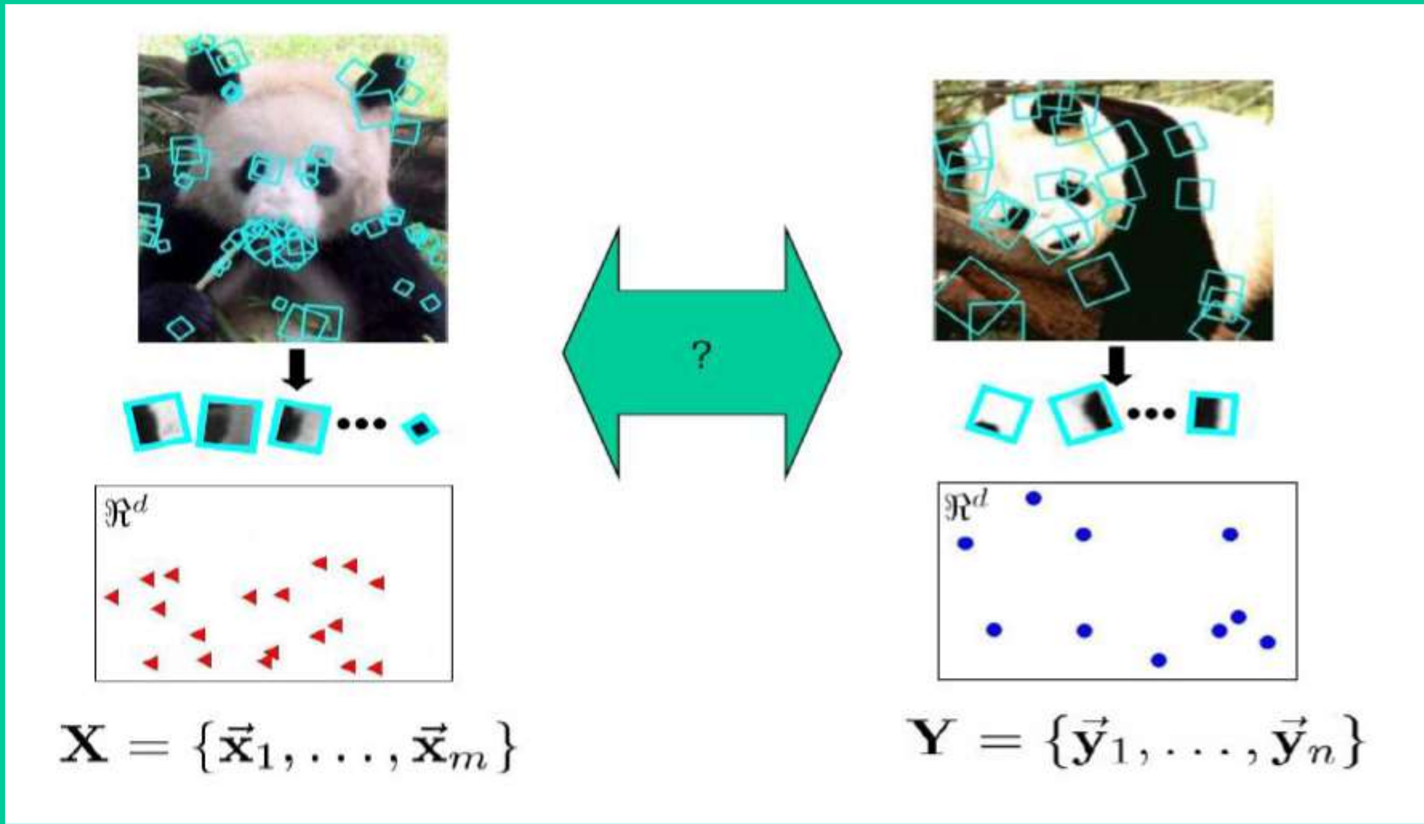
Dans une **structure ensembliste**, l'**ordre** dans lequel les objets sont considérés peut n'avoir aucune signification.

La seule propriété importante est :

- la **présence**
- ou l'**absence,**

d'un certain **objet** dans cette structure

Recherche d’ «indices visuels» discriminants



Les indices visuels sont tous uniques

Décomposition/ Reconstitution d'images

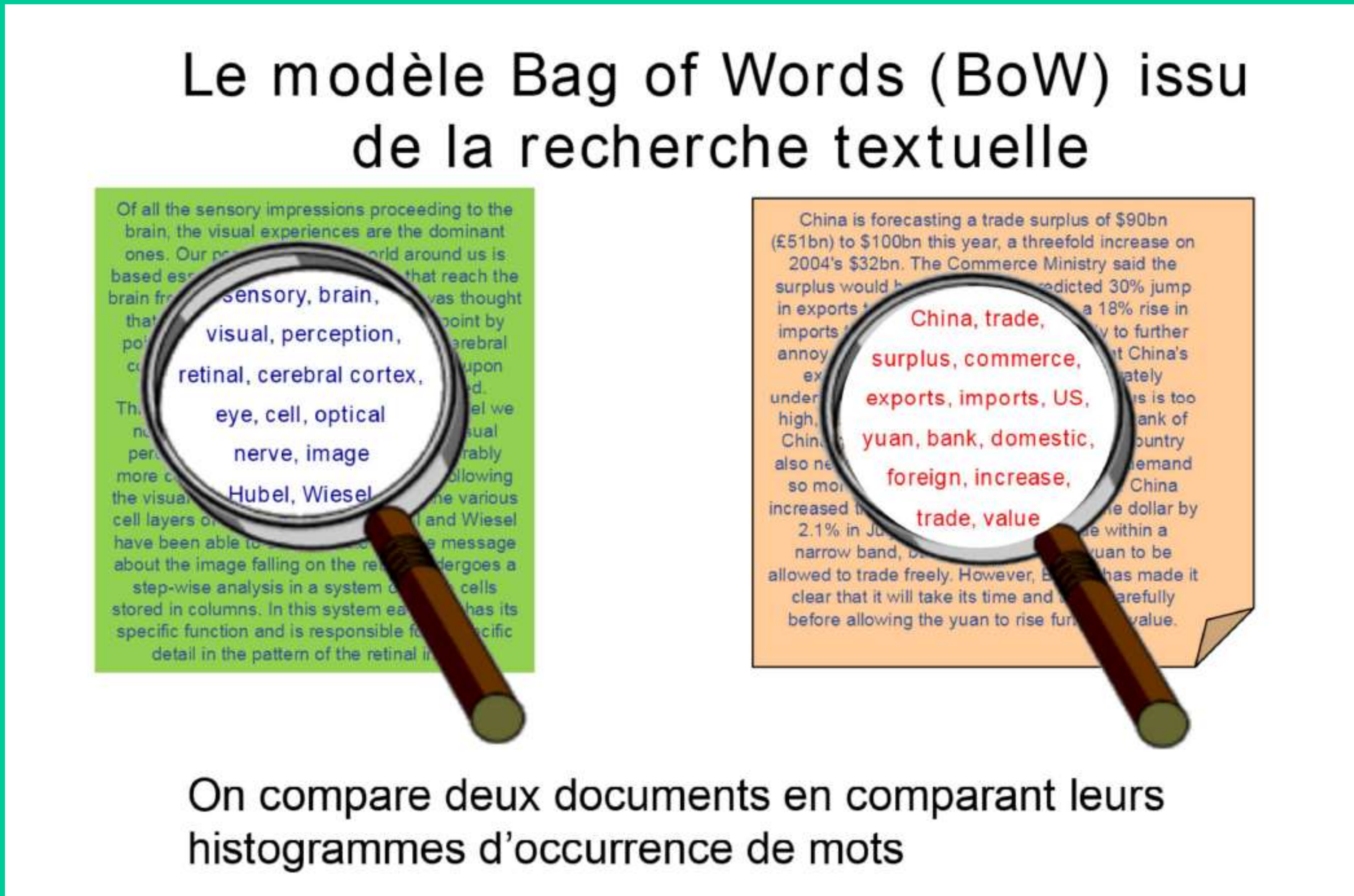
Les images sont toutes distinctes deux à deux

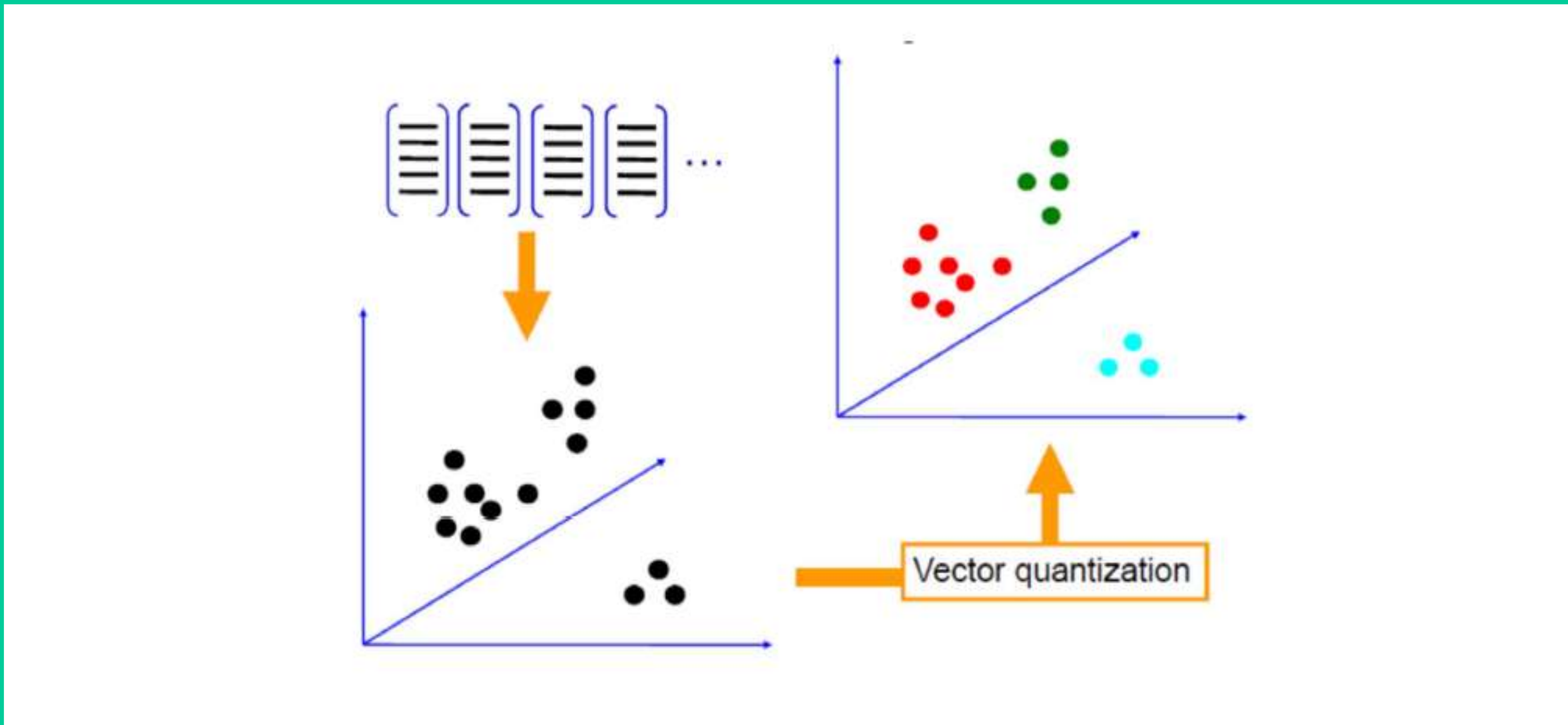
Dans un set un certain objet peut figurer **au plus une fois**.

Exemple du Set en Java

```
// dans une méthode main
 // création du set
set<String> set =  new HashSet<String>() ;

 // ajout d'élément
System.out.println("J'ajoute un : " + set.add("un")) ;
System.out.println("J'ajoute deux : " + set.add("deux")) ;

 // ajout d'un doublon : échec
System.out.println("J'ajoute encore un : " + set.add("un")) ;

 // affichage de la taille du set
System.out.println("Taille du set : " + set.size())
```

Un set où des objets peuvent avoir **plusieurs occurrences** «dégénère» en **bag**.

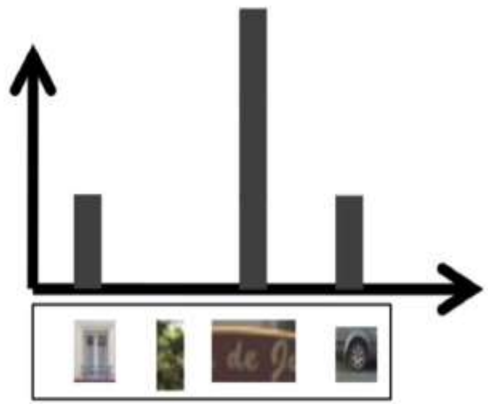



Vector quantization

Modèle Bag of Visual Words (BoW)

Sac de descripteurs (features)

BoW : histogramme sur un dictionnaire



I- TYPE ABSTRAIT SET

Sur les **set**, on doit réaliser, au moins, les opérations suivantes:

1- créer un set **vide** :

setVide$\longrightarrow$ Set [Elem]

2- ajouter un objet:

ajouter: Set [Elem] x Elem$\longrightarrow$ Set

3- enlever un objet:

enlever: Set [Elem] x Elem $\rightarrow$ Set

4- tester si un objet **appartient** à un set:

appartient : Elem x Set [Elem] $\rightarrow$ Booleen

5- tester la vacuité d'un set :

estVide : Set [Elem] $\rightarrow$ Booleen

Options possibles :

Dans le cas d’un élément **déjà présent**, il y a deux choix pour le résultat de l’opération **ajouter**.

Il en va de même, pour l’opération **enlever** dans le cas d’un **élément absent**.

On peut :

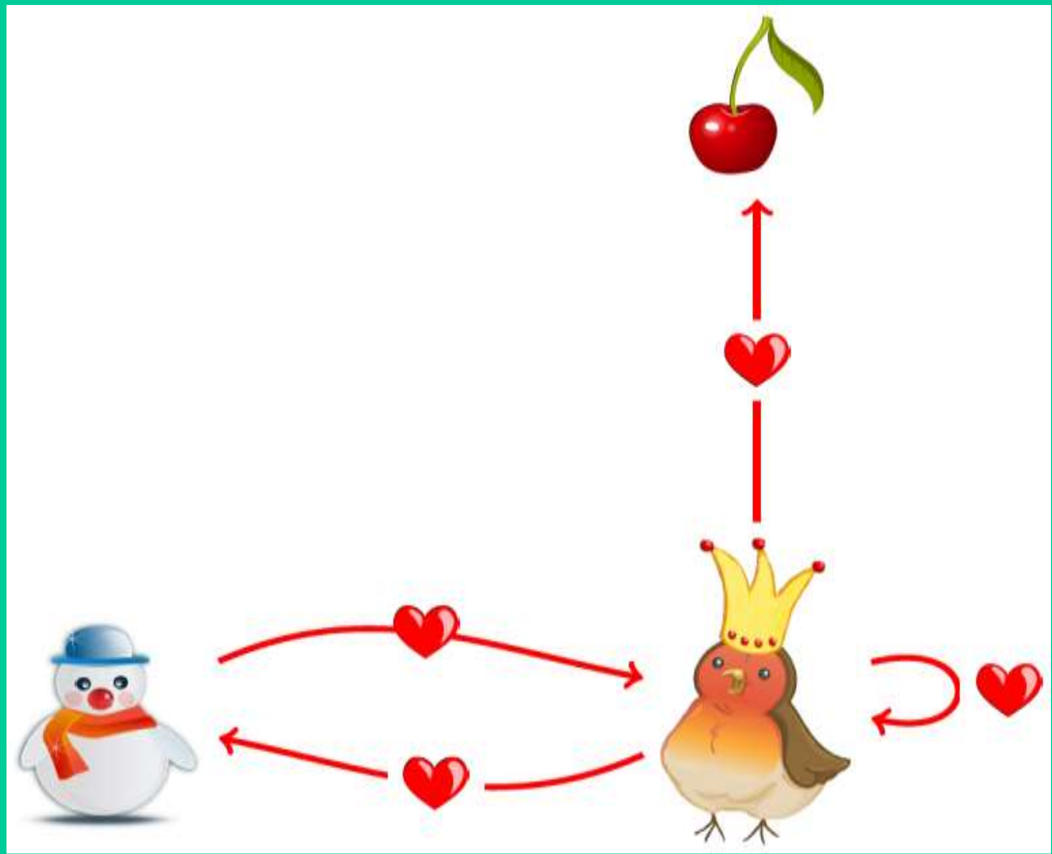
- soit considérer que ces opérations sont **sans effet**,

- soit imposer la satisfaction des **pré-conditions** suivantes :

$\forall$s :Set ; e :Elem

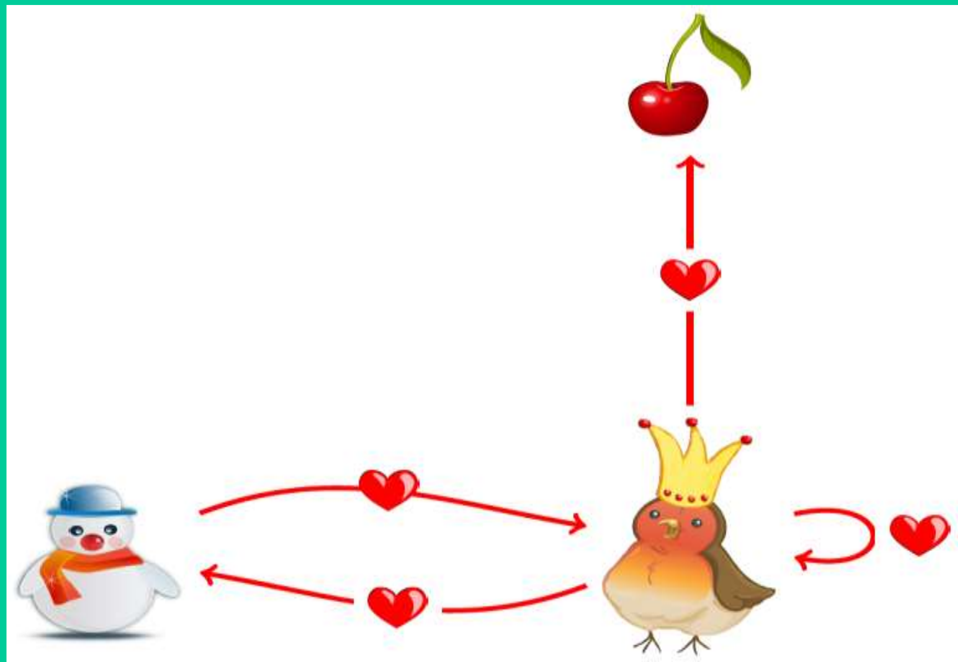
- **def ajouter**(s,e) $\Leftrightarrow$ **not appartient**(e,s)
- **def enlever**(s,e) $\Leftrightarrow$ **appartient**(e,s)

Notation : **évaluation du prédicat aime**

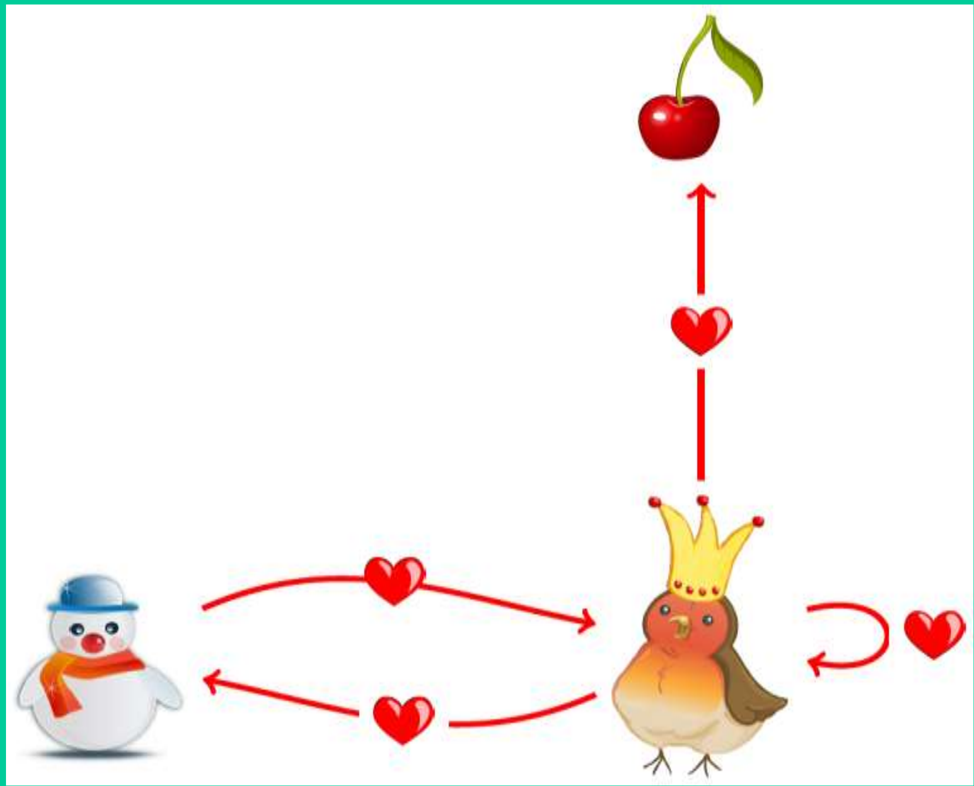


Domaine à 3 éléments

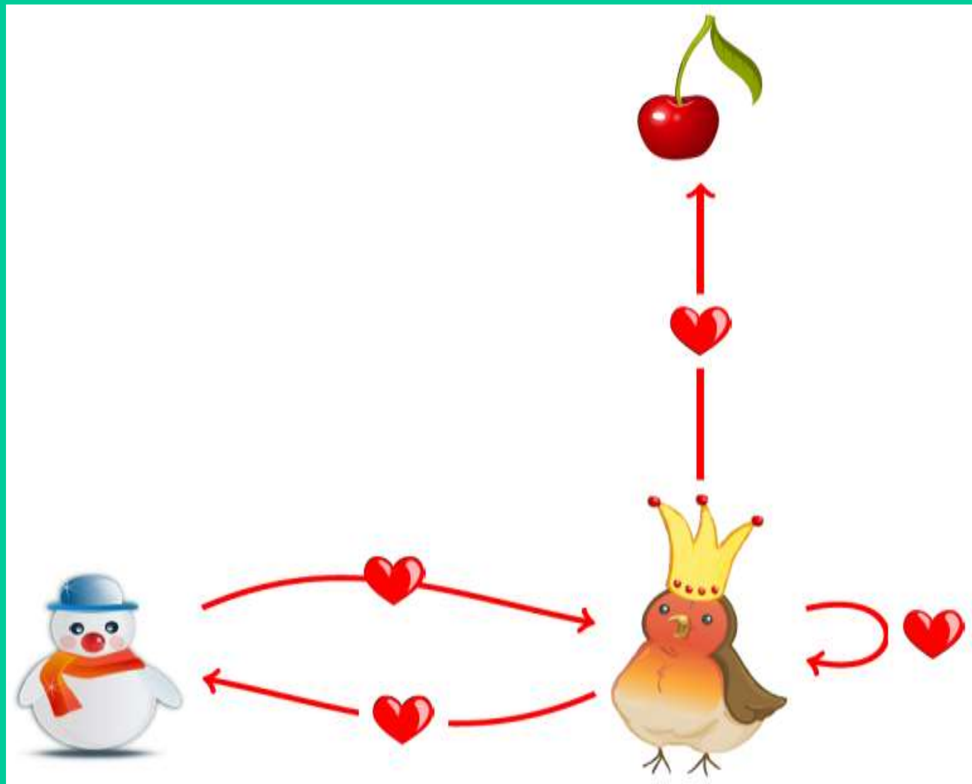
Logique à deux états



aime(oiseau, bonhomme)
aime(oiseau, cerise)
aime(oiseau, oiseau)
aime(bonhomme, oiseau)



not **aime**(cerise, oiseau)
not **aime**(cerise, bonhomme)
not **aime**(cerise, cerise)



aime(bonhomme, oiseau)

not aime (bonhomme, cerise)

not aime(bonhomme, bonhomme)

Spécification du type abstrait SET

La spécification du type peut être établie en plusieurs niveaux :

-niveau 0 : **setVide**, **ajouter**, **appartient**

-niveau 1 : **enlever** , **estVide**,

-niveau 2 : **inter**, **union**, **complement** ,
singleton, **inclut**, **card**

Spécification étendue SET2

inter, union, complement, singleton

inclut, card

extension 2

Spécification étendue SET1

enlever, estVide

extension 1

Spécification minimale SET0

setVide, ajouter, appartient

Une spécification minimale **SET0** du type Set peut être exprimée en **Casl** comme suit:

```
spec SET0 [sort Elem] =
generated type Set[Elem] ::=  setVide |
                                ajouter(Set[Elem]; Elem)

pred
    appartient: Elem × Set[Elem]
```

```
forall x, y: Elem; M, N: Set[Elem]

. not appartient(x, setVide)
. appartient(x, ajouter(M,y)) <=> x = y \/ appartient(x, M)
. M = N ⇔ appartient(x ,M) <=> appartient(x,N)

end
```

La spécification précédente peut être étendue en considérant :

-le constructeur **enlever** avec la signature :

enlever: Set[Elem] * Elem -> Set[Elem]

-et le prédicat **estVide** avec la signature :

estVide: Set[Elem]

On remarquera **enlever** aura le statut d'accesseur car le type **Set[Elem]** a été déjà déclaré dans **Set0**

Ce qui donne l'extension de la spécification **SET0** en **SET1** décrite comme suit:

spec SET1 [sort Elem] **given** NAT=

SET0 [sort Elem]

then

pred

estVide: Set[Elem]

op

enlever: Set[Elem] * Elem -> Set[Elem]

```
forall x, y: Elem; M: Set[Elem]
    . estVide(M)  <=> M = setVide

%% le constructeur enlever est défini par induction à partir de setVide et ajouter
    . enlever(setVide, y) = setVide
    . enlever(ajouter(M,x), y) = M when x = y
                                   else ajouter(enlever(M,y),x)
end
```

La spécification précédente peut être **étendue** en introduisant d'autres **opérations classiques** sur les ensembles .

- **card** ,
- **union, intersection ,**
- **complémentaire, singleton**, etc.

qui auront également le statut d'accesseurs car le type **Set[Elem]** a été déjà déclaré dans **Set0**

D'où l'**extension** de la spécification **SET1** en **SET2**

```
spec SET2 [sort Elem] given NAT =
     SET1 [sort Elem]
     then
    pred
          inclut: Set[Elem] * Set[Elem]
     ops
         card:    Set[Elem] -> Nat;
         inter:   Set[Elem] * Set[Elem] -> Set[Elem];
```

```
        union: Set[Elem] * Set[Elem] -> Set[Elem];
        complement:   Set[Elem] * Set[Elem] ->? Set[Elem];
        singleton:        Elem -> Set[Elem]
forall x: Elem; M, N: Set[Elem]
    . def complement(M,N) <=> inclut(M,N)
    . inclut(M,N) <=> forall x:Elem • appartient(x,N)=>appartient(x,M)
    . card(setVide) = 0
    . card(ajouter(M, x)) = card(M) when appartient(x,M)
                                      else card(M)+1
    . inter(M, setVide)= setVide
    . inter(M , (ajouter(N, x)) =
            ajouter(inter(M,N) ,x)  when appartient(x, M) else inter(M, N)
```

. union(M, setVide)= M

. union(M,ajouter(N,x))= union(M, N) when appartient(x, M)

else ajouter(union(M,N),x)

. complement(M, setVide) = M

. appartient(x,M)$\Rightarrow$complement(M,ajouter(N,x))

= enlever(complement(M,N),x)

. singleton(x) = ajouter(setVide, x)

end

2-Enumération d'un Set

Dans le cas d'une liste, par exemple, le parcours s'effectue dans l'**ordre séquentiel** des objets.

Il n'en est rien dans le cas d'un set car **aucun ordre** n'est déterminé.

Pour traiter tous les objets d’un set **s** non vide:

1- on choisit un objet **arbitraire** **x** de ce set:

$$\mathbf{x} \leftarrow \mathbf{choisir}(\mathbf{s})$$

2- on le **traite**,

3- et on **recommence** sur le set **s** obtenu en enlevant l’objet **x**:

$$\mathbf{s} \leftarrow \mathbf{enlever}(\mathbf{s},\mathbf{x})$$

4- ceci est **répété** tant que **s** n’est pas vide :

not **estVide**(**s**)

On enrichit donc la signature par l’accesseur **choisir** dont la signature est la suivante :

choisir: Set $\rightarrow$ Elem

Cette opération est gardée par la précondition:

def choisir(s) $\Leftrightarrow$: **not** **estVide**(s)

Un seul axiome est suffisant pour exprimer sa propriété :

forall s :Set[Elem] . **appartient**(**choisir**(s),s)

La procédure de traitement de tous les objets d'un set **s** est la suivante :

```
s'← s ;
tant_que  not estVide(s')  faire
        début
         x   ← choisir(s');
         traiter(x) ;
         s'   ← enlever(s',x)
        fin
```

II- TYPE ABSTRAIT BAG

Dans les **bag**, on trouve les mêmes opérations que pour les **set**.

1-créer un bag **vide** :

bagVide $\rightarrow$ Bag

2-ajouter un objet:

ajouter: Bag x Elem$\rightarrow$ Bag

3-enlever un objet:

enlever: Bag x Elem$\rightarrow$ Bag

4-tester si un objet **appartient** à un bag:

appartient : Elem x Bag $\rightarrow$ Booleen

5-évaluer le nombre d'occurrences d'un objet :

fréquence : Elem x Bag $\rightarrow$ Entier

6- tester la vacuité d'un bag:

estVide : bag$\rightarrow$ Booleen

Différences entre un bag et un set :

1- le nombre d'objets augmente quand on ajoute un objet **déjà présent**,

bag1= {p_i, p_k, p_v, p_k , p_n}

ajouter(bag1, p_v)

bag1= {p_i, p_k, p_v, p_k , p_n, p_v}

2- enlever un objet n’implique pas que cet objet n’appartient plus au bag: il peut être **présent plusieurs fois**.

bag1= $\{p_i, p_k, p_v, p_k, p_n, p_v\}$

enlever(bag1, p_v)

bag1= $\{p_i, p_k, p_v, p_k, p_n\}$

Spécification du type Bag

La spécification du type peut envisager plusieurs niveaux :

-niveau 0 : **bagVide**, **ajouter**, **fréquence**

-niveau 1 : **enlever** , **estVide**,

-niveau 2 : **inter**, **union**, **complement** ,
singleton, **inclut**, **card**, **appartient**

Spécification étendue BAG2

inter, union, complement, singleton

inclut, card, appartient

extension 2

Spécification étendue BAG1

enlever, estVide

extension 1

Spécification minimale BAG0

bagVide, ajouter, frequence

Comme pour le type des **set**, on commence par établir une spécification **minimale** qui introduit :

-**les générateurs** :

bagVide, **ajouter**

-et un accesseur:

frequence

Cela donne la spécification **BAG0** établie comme suit:

```
spec BAG0 [sort Elem] given NAT =

generated type Bag[Elem] ::= bagVide |
                             ajouter(Bag[Elem]; Elem)
op
    frequence : Bag[Elem] * Elem -> Nat
```

```
forall x,y: Elem; M, N:Bag[Elem]
. frequence(bagVide, y) = 0
. frequence(ajouter(M,x), y) = frequence(M,y)+1 when x = y
                                        else frequence(M, y)

.M = N <=> forall  x: Elem .  frequence(M,x) = frequence(N,x )

end
```

La spécification précédente peut être étendue en **BAG1** par **extension** sur **BAG0** comme suit:

```
spec BAG1 [sort Elem] given NAT =
        BAG0 [sort Elem]

then
pred
        estVide: Bag[Elem]
op
        enlever: Bag[Elem] * Elem -> Bag[Elem]
```

forall x, y: Elem; M, N: Bag[Elem]

%% axiomes concernant le prédicat

. **estVide**(M) <=> M = **bagVide**

. N = **enlever**(M,x) <=>

  frequence(N,y) = **frequence**(M,x)−1 **when** x = y **else** **frequence**(M,y)

end

La spécification **BAG** peut être étendue en **BAG2** comme suit:

spec BAG2 [sort Elem] given NAT =

BAG1 [sort Elem]

then

preds

appartient: Elem * Bag[Elem];

inclut: Bag[Elem] * Bag[Elem]

ops

singleton: Elem -> Bag[Elem];

union: Bag[Elem] * Bag[Elem] -> Bag[Elem];

inter: Bag[Elem] * Bag[Elem] -> Bag[Elem];

complement: Bag[Elem] * Bag[Elem] -> Bag[Elem];

card: Bag[Elem] -> Nat

forall x, y: Elem; M, N, O: Bag[Elem]

. **def** **complement**(M,N) <=> **inclut**(M,N)

%% axiomes concernant les prédicats

. **appartient**(x,M) <=> **frequence**(M, x) > 0

. **inclut**(M,N) <=> **frequence**(M,x) $\geq$ **frequence**(N,x)

%% axiomes concernant les opérations

. **singleton**(x) = **ajouter**(bagVide,x)

. **union**(M,N) = 0 <=>

frequence(0,x) = **frequence** (M,x) + **frequence** (N,x)

. **inter**(M,N) = 0 <=>**frequence**(0,x)=min(**frequence**(M,x),**frequence**(N,x))

. **complement**(M,N) = 0 <=> **frequence**(0,x) = **frequence**(M,x) – **frequence**(N,x)

```
. card(bagVide)  = 0
. card(ajouter(M,x)) = card(M)+1
. card(enlever(M,x)) = card(M)-1  when appartient(M,x) else card(M)
end
```