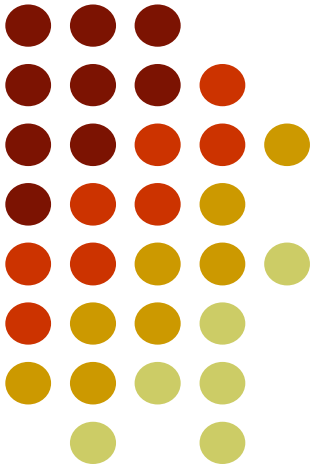
Le langage Prolog

Cours n°4

Représentation des graphes et des arbres

Compléments Prolog

Jean-Michel Adam



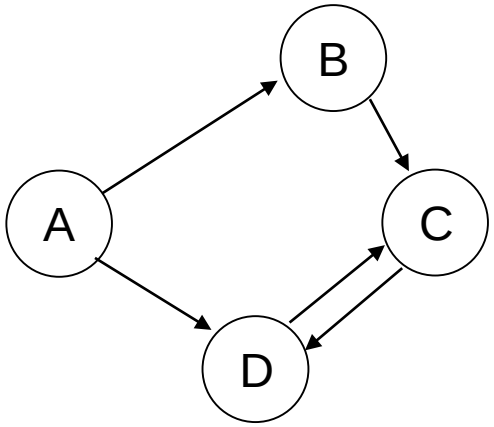
Plan du cours

- Les structures de données
  - Les graphes
  - Les arbres
- Généralisation d’un prédicat
- Dynamisme

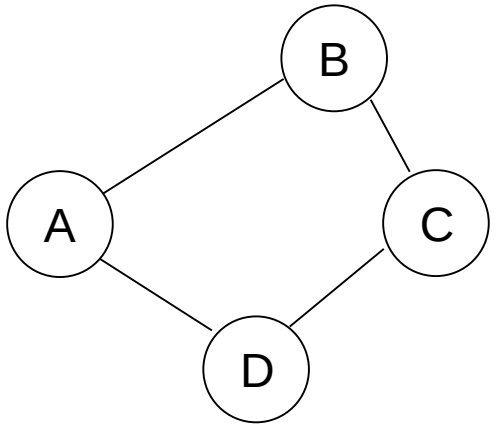
Partie 1
-
Les Graphes

Les graphes en Prolog

- Graphe
  - Ensemble de sommets (ou nœuds) reliés par des arcs (graphe orienté) ou des arêtes (graphe non orienté).
  - Permet de modéliser de nombreuses situations concrètes comme par exemple : des liaisons routières, des flux, des tâches à ordonner, etc.



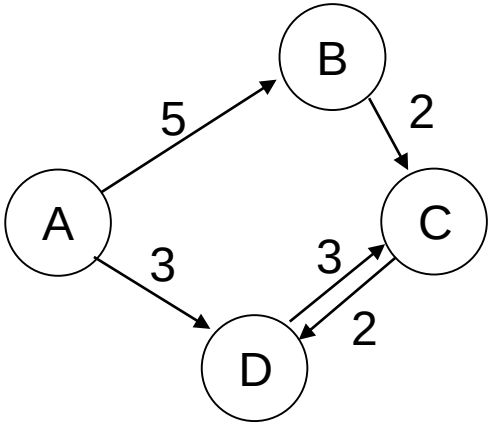
Graphe orienté



Graphe non orienté

Représentation des graphes

- Une manière simple de représenter un graphe en Prolog est de décrire
  - Les arcs qui relient les sommets et leurs caractéristiques
  - Les sommets et leur caractéristiques



Graphe orienté

```
arc(a,b,5).
arc(b,c,2).
arc(a,d,3).
arc(c,d,2).
arc(d,c,3).
sommet(a).
sommet(b).
sommet(c).
sommet(d).
```

Exemple : des figures à colorier

Problème posé :

- On veut colorier les zones des figures, de sorte que deux zones adjacentes n'aient pas la même couleur.
- On peut représenter ces figures par des graphes :
  - Les sommets représentent les zones de la figure
  - Les arêtes relient deux zones adjacentes

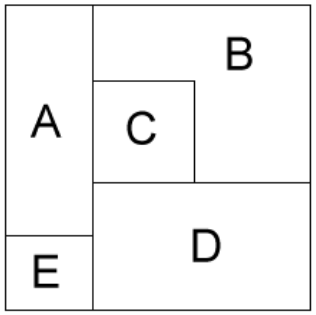


Figure 1

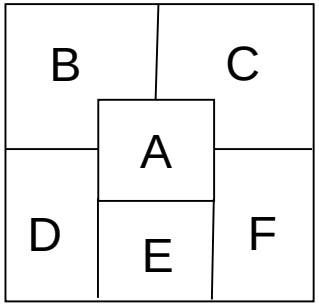


Figure 2

Exemple : des figures à colorier

- Les sommets représentent les zones de la figure
- Les arêtes relient deux zones adjacentes

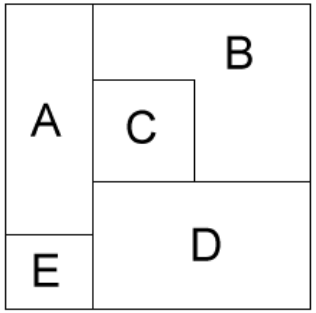
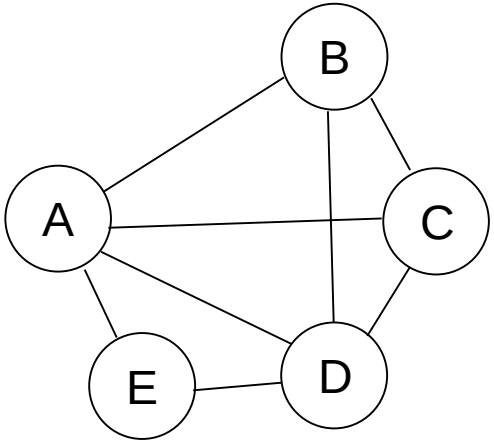


Figure 1



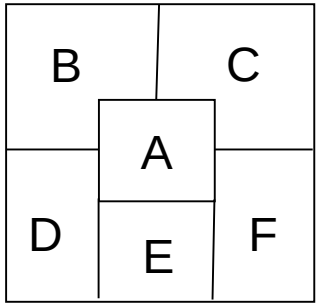
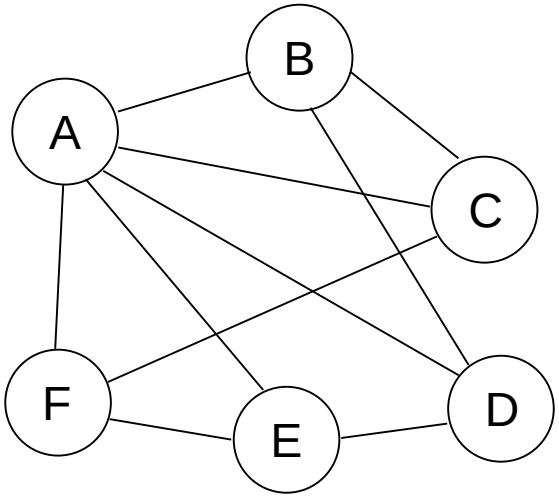


Figure 2



Exemple : des figures à colorier réalisation en Prolog

- Représentation des 2 figures en Prolog

```
zones(figure1,[a,b,c,d,e]).
zones(figure2,[a,b,c,d,e,f]).

arete(figure1,a,b).
arete(figure1,a,c).
arete(figure1,a,d).
arete(figure1,a,e).
arete(figure1,b,c).
arete(figure1,b,d).
arete(figure1,c,d).
arete(figure1,d,e).

arete(figure2,a,b).
arete(figure2,a,c).
arete(figure2,a,d).
arete(figure2,a,e).
arete(figure2,a,f).
arete(figure2,b,c).
arete(figure2,b,d).
arete(figure2,c,f).
arete(figure2,d,e).
arete(figure2,e,f).

aretes(Fig,L):- findall([X,Y], arete(Fig,X,Y), L).
```

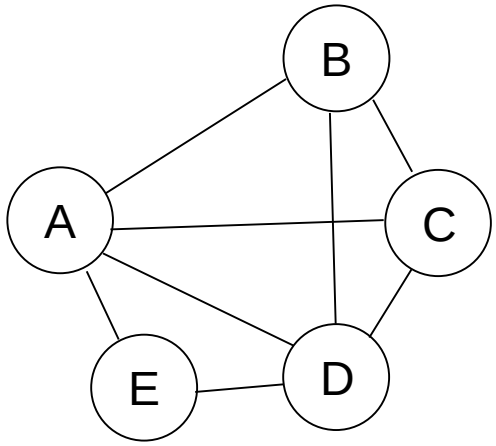


Figure 1

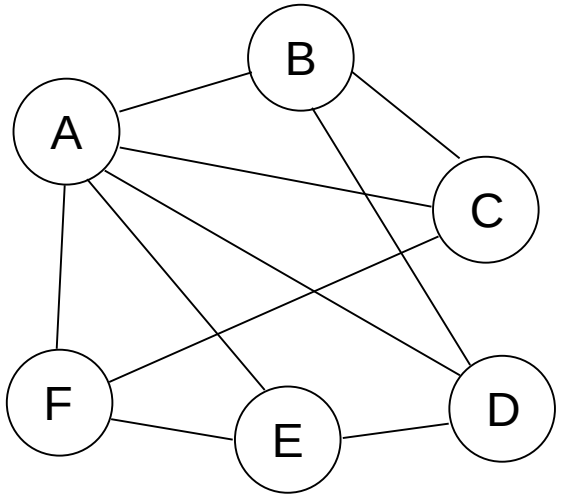


Figure 2

Exemple : des figures à colorier réalisation en Prolog

- On veut écrire le prédicat `coloriage/2` qui, à partir du nom de la figure, donne un coloriage possible à l'aide des couleurs disponibles
- Le résultat est une liste de couples [zone, couleur]

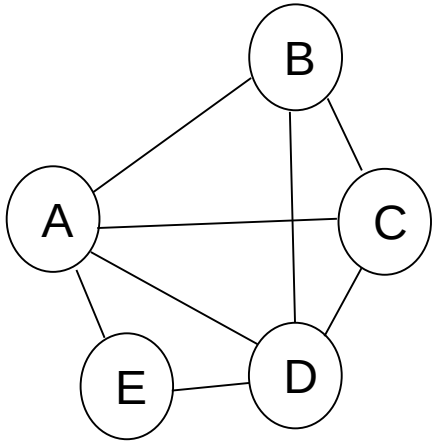
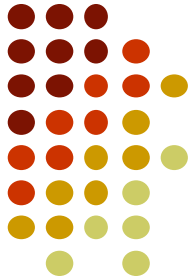


Figure 1

```
?- coloriage(figure1,L).
L = [[e, vert], [d, rouge], [c, bleu], [b, vert], [a, jaune]]
```

Exemple : des figures à colorier réalisation en Prolog



Programme de coloriage :

- **adjacent/3** : prédicat indiquant si 2 sommets (zones) sont adjacent(e)s dans la figure donnée.

  adjacent(Fig,X,Y):- arete(Fig,X,Y); arete(Fig,Y,X).

  dans la figure Fig, X adjacent à Y si une arête les relie.

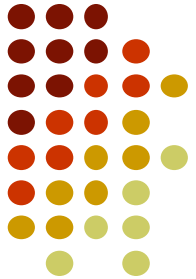
- **conflit/3** : prédicat indiquant s'il y a conflit de coloriage entre les zones Z1 et Z2 d'une figure.

  conflit(Fig,[Z1,C1],[Z2,C2]):- adjacent(Fig,Z1,Z2), C1==C2.

  ou plus simplement :

  conflit(Fig,[Z1,C],[Z2,C]):- adjacent(Fig,Z1,Z2).

Exemple : des figures à colorier réalisation en Prolog



- **noconflit/3** : prédicat indiquant si la couleur d'une zone n'est pas en conflit avec les couleurs déjà sélectionnées pour les autres zones (liste de doublets)

```
noconflit(Fig,X,[]).
noconflit(Fig,X,[P|F]):- \+conflit(Fig,X,P)), noconflit(Fig,X,F).
```

- **coloriage** : coloriage de la figure Fig

```
coloriage(Fig,L):- zones(Fig,Zones), colorier(Fig,Zones,[],L).
```

L est une liste de doublets [zone, couleur].

- **colorier** : trouve la liste des couleurs de chaque zone de la figure

```
colorier(_,[],L,L).
colorier(Fig,[Z|F],L,R):- couleur(C), noconflit(Fig,[Z,C],L),
                          colorier(Fig,F,[[Z,C]|L],R).
```

Démonstration

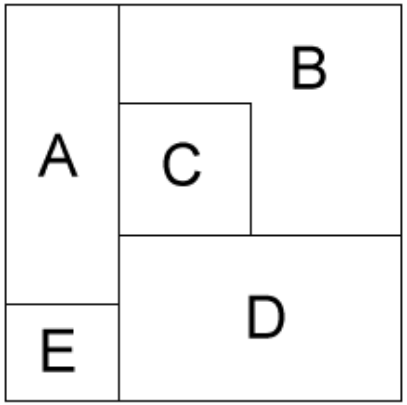


Figure 1

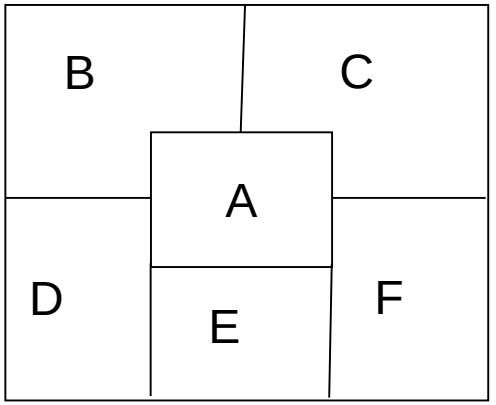


Figure 2

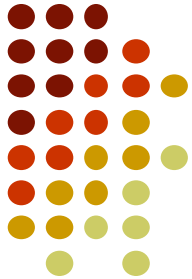
Partie 2
-
Les Arbres

Les Arbres en Prolog

Eléments abordés :

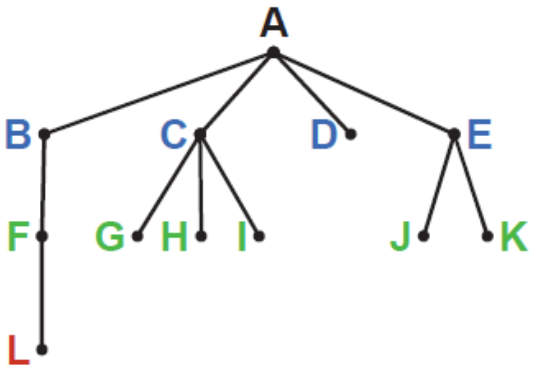
- Arbres n-aires et arbres binaires
- Représentation des arbres en Prolog
- Parcours d’arbres

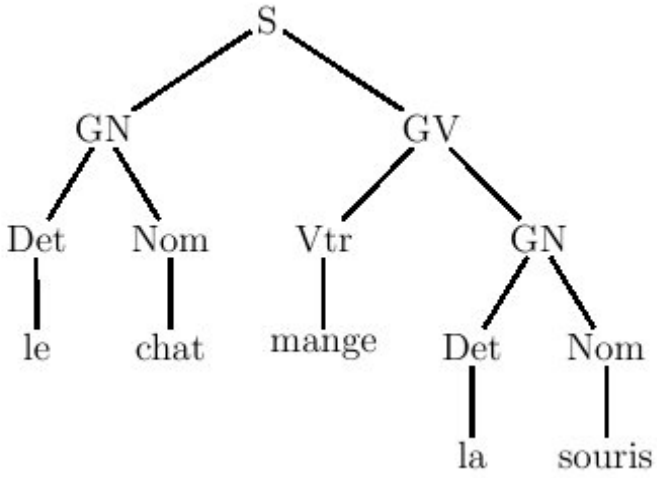
Les Arbres

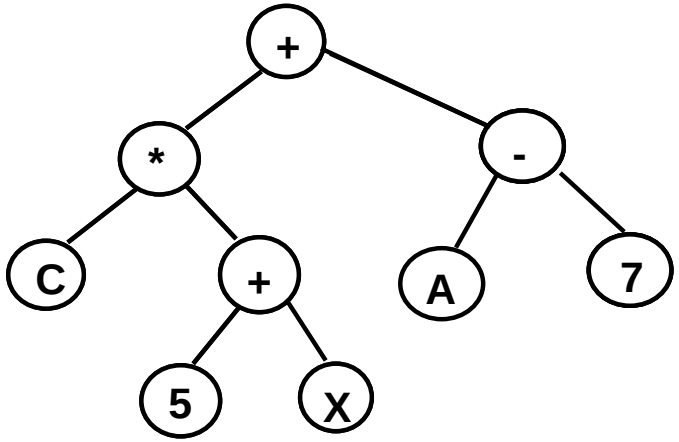


- Les arbres sont des graphes particuliers.
- Les arbres, comme les listes, permettent de représenter un nombre variable de données
- Le principal avantage des arbres est qu'ils permettent d'organiser les données selon un ordre partiel.
- Exemples d'arbres :
  - Arbre généalogique
  - Sommaire d'un livre, hiérarchie de répertoires
  - Arbre de dérivation d'une phrase d'un langage défini par une grammaire

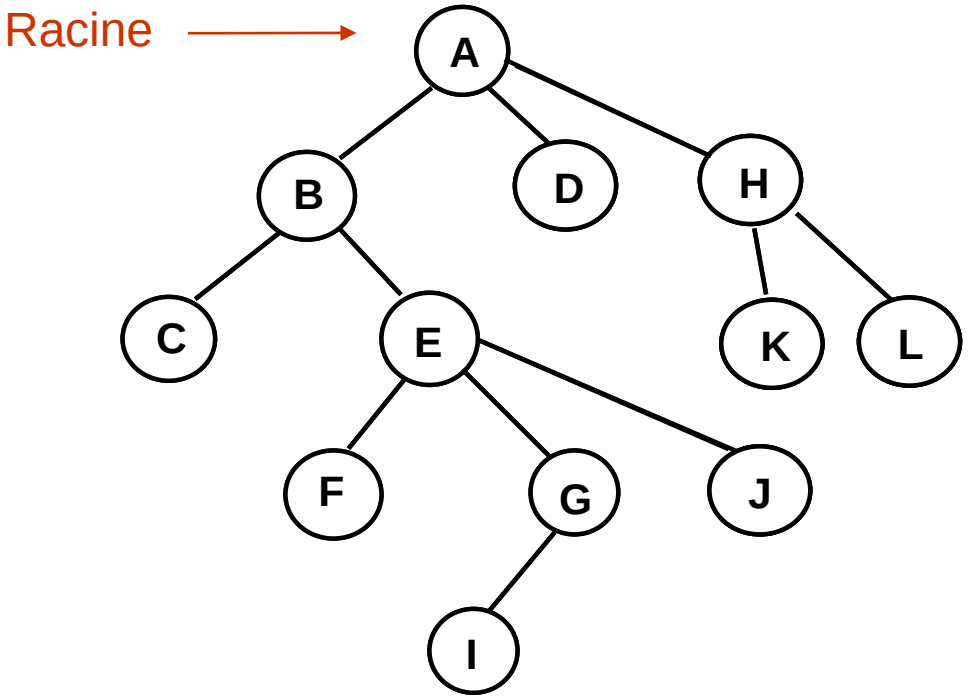
Exemples d'arbres



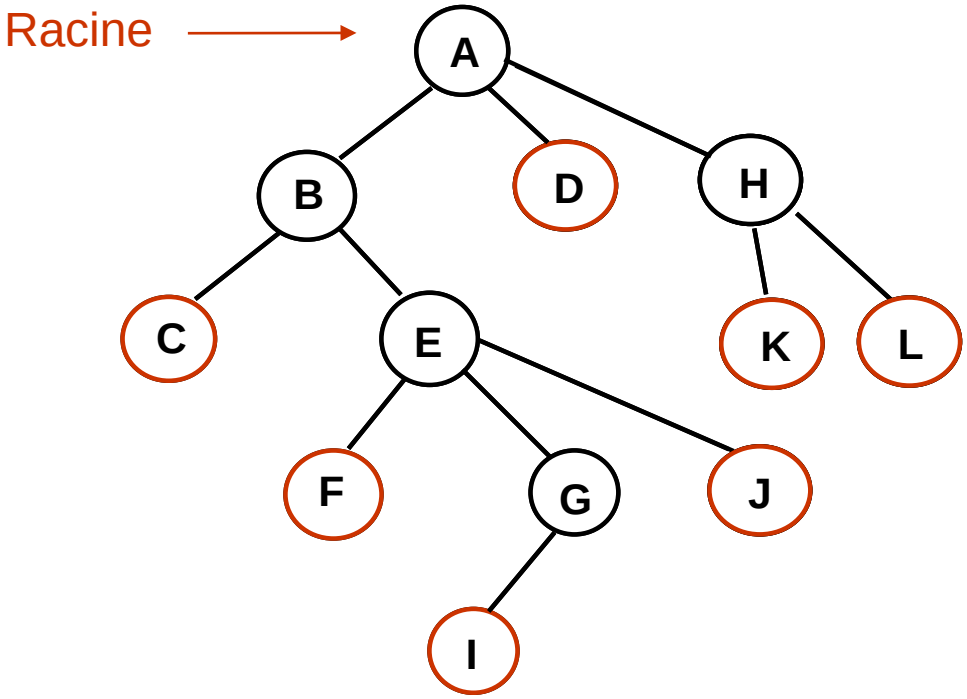




Représentations graphiques d'un arbre



Représentations graphiques d'un arbre



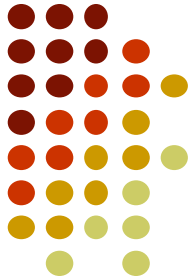
Définition d'un Arbre

- Un Arbre est un ensemble non vide structuré comme suit :
  - un des éléments est désigné comme étant la « racine » de l'arbre
  - il existe une partition sur les éléments restants, et chaque classe de cette partition est elle-même un arbre : on parle des sous-arbres de la racine.
- Si le nombre de sous-arbres est variable, l'arbre est dit n-aire.
- L'ensemble représenté par un arbre est la réunion d'un élément (la racine) et des sous-arbres qui lui sont directement associés.
- Chaque élément de l'ensemble structuré en arbre est appelé un nœud. A tout nœud est associée une information élémentaire.
- Pour décrire les relations entre les nœuds on utilise la terminologie de la généalogie, un nœud est donc le père de ses fils.
- Le degré d'un nœud est le nombre de ses fils. On distingue :
  - les nœuds non terminaux de degré non nul
  - les nœuds terminaux ou feuilles, de degré nul.

Arbre binaire

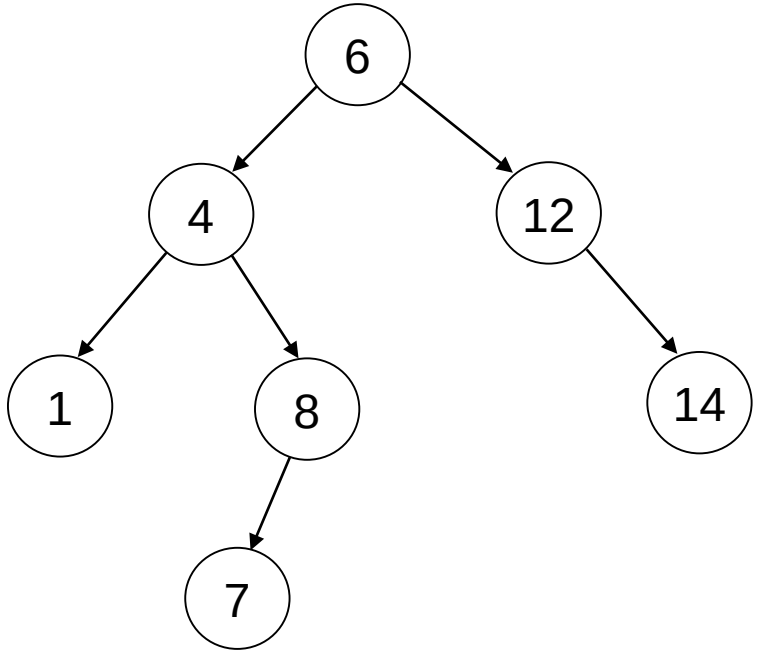
- Un arbre binaire est un arbre pour lequel tout nœud a au plus deux fils.
- Un arbre binaire est un ensemble fini qui est soit vide, soit composé d'une racine et de deux sous-arbres binaires appelés sous-arbre gauche et sous-arbre droit.
- On peut donc dire qu'un arbre binaire est :
  - soit l'arbre vide
  - soit un nœud qui a exactement deux sous-arbres éventuellement vides

Représentation des arbres binaires en Prolog



- Nous choisissons d'utiliser les listes pour représenter les arbres binaires
  - Un arbre vide sera représenté par la liste vide []
  - Un nœud sera représenté par une liste de 3 éléments :
    - le premier est la valeur portée par le nœud,
    - le deuxième son sous-arbre gauche,
    - le troisième son sous-arbre droit.

Exemple de représentation d'un arbre binaire en Prolog

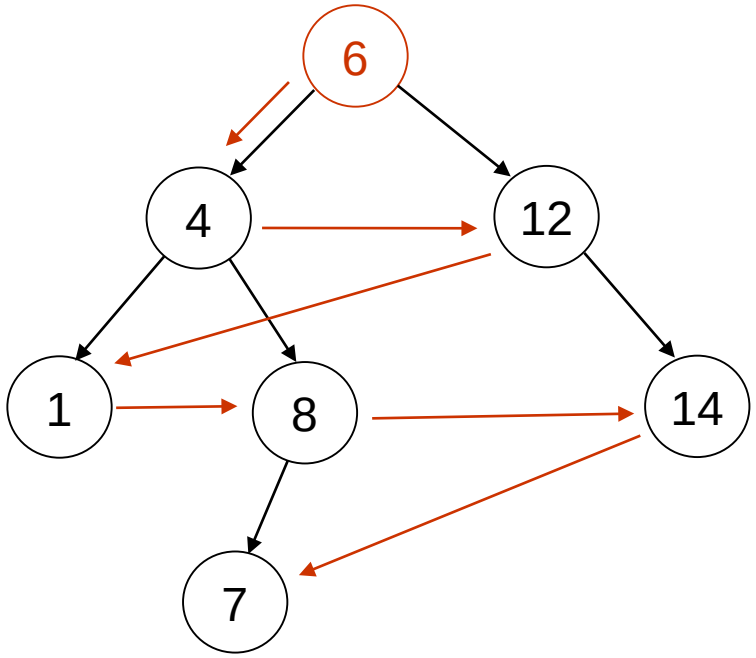


Cet arbre binaire ordonné est représenté par la liste suivante :
[6, [4, [1, [], []], [8, [7, [], []], []]], [12, [], [14,[],[]]]]

Parcours d'arbres

- Un arbre contient un ensemble de données.
- Pour utiliser ces données, on peut parcourir l'arbre
  - en largeur

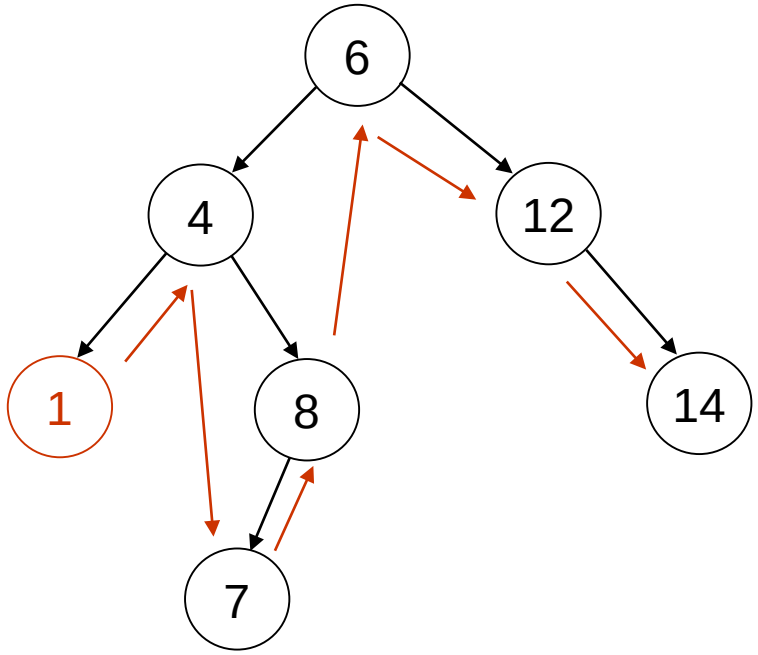
On parcourt l'arbre par niveaux : d'abord la racine, puis les nœuds de niveau inférieur, et ainsi de suite



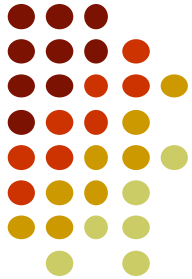
Parcours d'arbres

- Un arbre contient un ensemble de données.
- Pour utiliser ces données, il faut parcourir l'arbre
  - en profondeur

On visite d'abord le sous-arbre gauche, ensuite la racine, puis le sous-arbre droit



Exemples de parcours en profondeur



- Calcul de la somme des nœuds d'un arbre binaire de nombres

```
somme([],0).
somme([N, G, D], S) :- somme(G, N1), somme(D,N2),
                       S is N+N1+N2.
```

Exemples de parcours d'un arbre binaire

- Affichage des nœuds d'un arbre binaire
  - Dans l'ordre du parcours en profondeur (infixé)

    afficher([]).

    afficher([N, G, D]) :- afficher(G), write(N), afficher(D).
  - Dans l'ordre préfixé : racine puis les sous-arbres

    afficher([]).

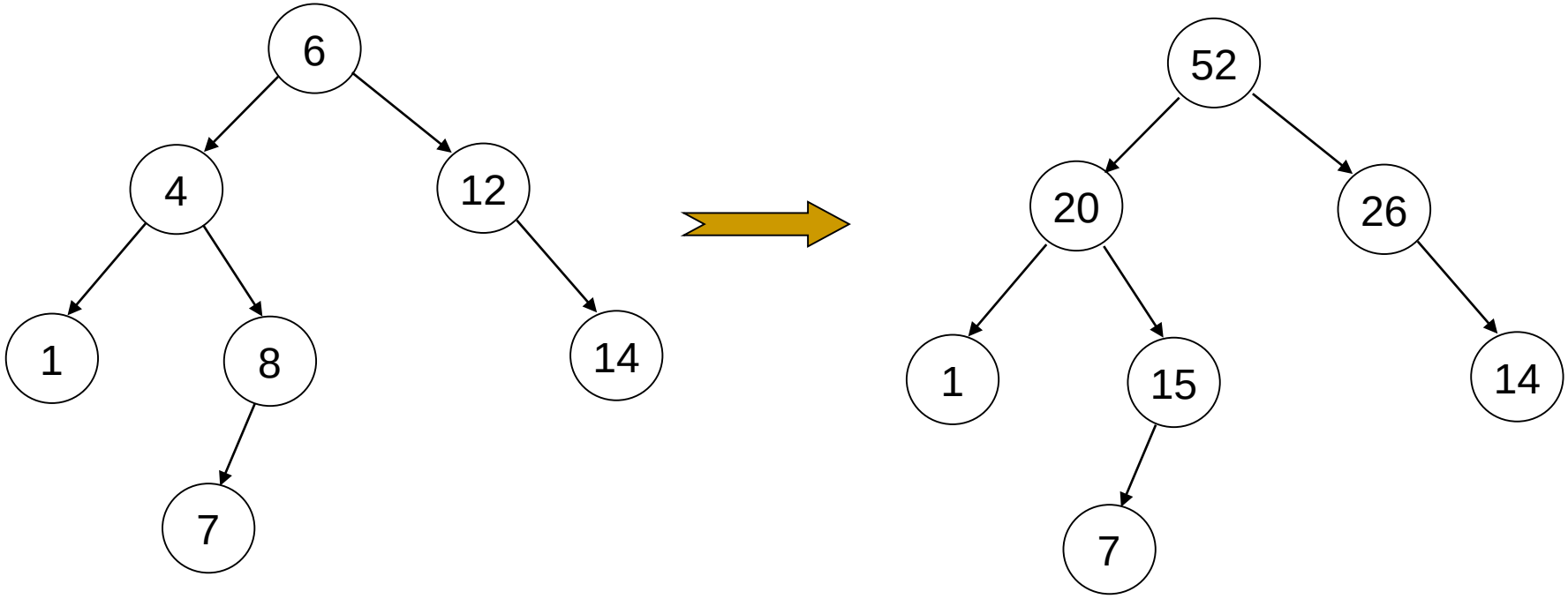
    afficher([N, G, D]) :- write(N), afficher(G), afficher(D).
  - Dans l'ordre postfixé : les sous-arbres puis la racine

    afficher([]).

    afficher([N, G, D]) :- afficher(G), afficher(D), write(N).

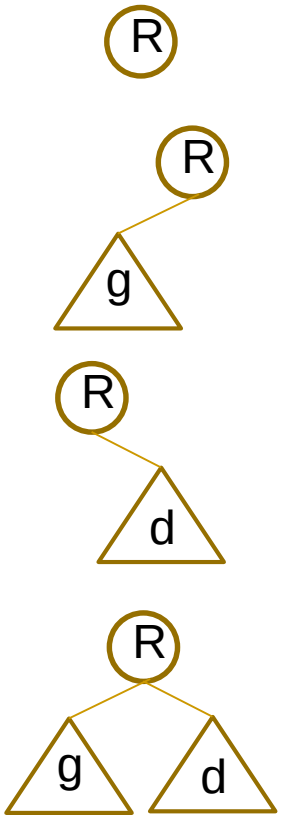
Exemples de construction d'un arbre binaire

- A partir d'un arbre, créer un arbre qui représente la somme des valeurs des sous-arbres

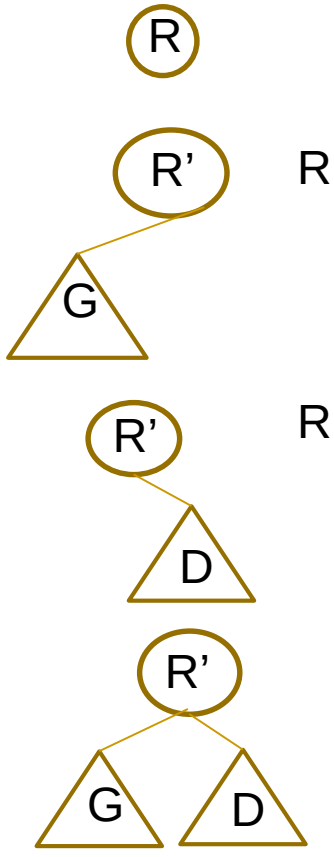


Arbre représentant la somme des valeurs des sous-arbres : **analyse**

Cas possibles



Arbre construit



R' = R + somme des élts de g

R' = R + somme des élts de d

R' = R + somme des élts de g
+ somme des élts de d

racine(G) = somme des élts de g
racine(D) = somme des élts de d

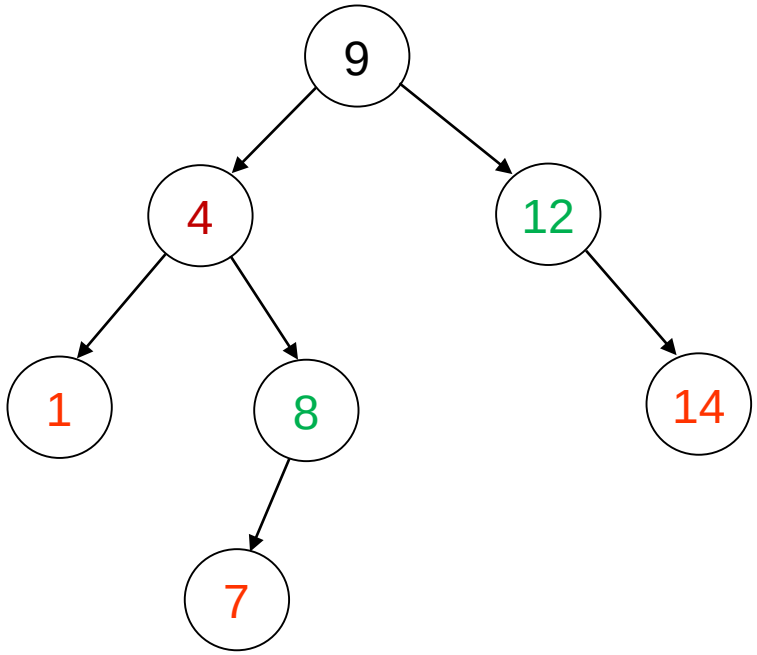
Exemples de construction d'un arbre binaire

- A partir d'un arbre, créer un arbre qui représente la somme des valeurs des sous-arbres
- Soit `creer_som/2` ce prédicat :

```
creer_som([],[]).
creer_som([N,[],[]], [N,[],[]]).
creer_som([N,G,[]], [NR, [NG,FG,FD], []]):- creer_som(G,[NG,FG,FD]), NR is N+NG.
creer_som([N,[],D], [NR, [], [ND,FG,FD]]):- creer_som(D,[ND,FG,FD]), NR is N+ND.
creer_som([N,G,D], [NR, [NG,FG1,FD1], [ND,FG2,FD2]]):-
creer_som(G,[NG,FG1,FD1]), creer_som(D,[ND,FG2,FD2]), NR is N+ND+NG.
```

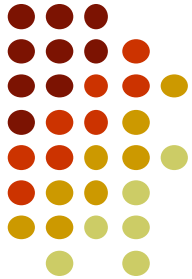
Une autre manière de représenter les arbres binaires en Prolog

- Un arbre binaire peut être représenté par le terme de suivant: t(SousArbreG, Racine, SousArbreD).



- t(t(t(nil, 1, nil), 4, t(t(nil, 7, nil), 8, nil)), 9, t(nil, 12, t(nil, 14, nil)))
- Arbre vide : nil

Exercice : construction d'un arbre binaire de recherche



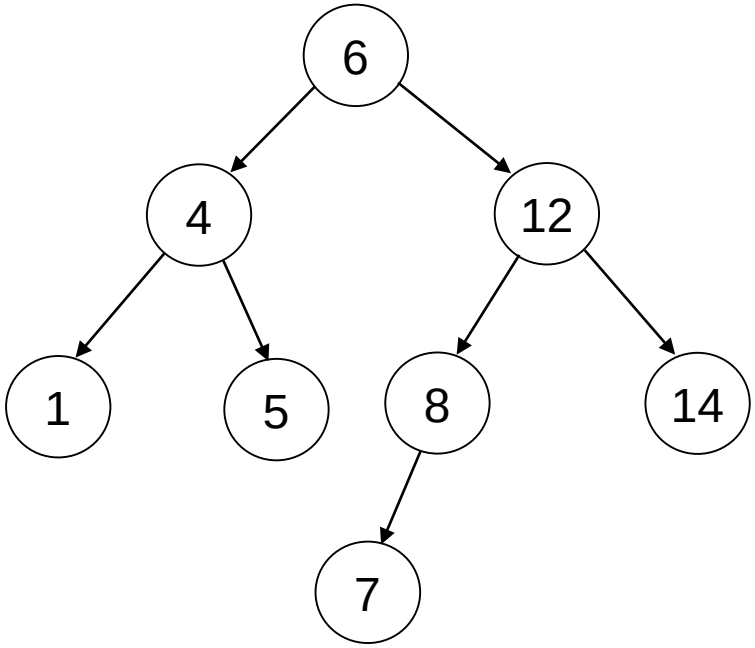
- Ecrire un prédicat qui, à partir d'une liste d'entiers, construit un arbre dont les nœuds sont les éléments de la liste et qui vérifie les propriétés suivantes :
  - Tout nœud situé dans un sous-arbre gauche est plus petit que la racine
  - Tout nœud situé dans un sous-arbre droit est plus grand que la racine

Exemple : à partir de

la liste [6,12,4,8,14,5,7,1]

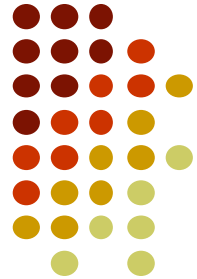
on obtient l'arbre binaire

de recherche suivant :



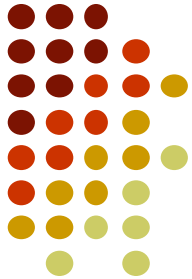
Partie 3
-
Généralisation de prédicat

Généralisation d’un prédicat pour le rendre symétrique



- Exemples de prédicats symétriques :
  - append(L1,L2,LR).
  - atom_chars(Atome,Liste_caractères).
  - name(mot,Liste_codes_Unicode).

Généralisation d'un prédicat pour le rendre symétrique



- Exemple de prédicat non symétrique :

```
fact(0, 1).
fact(N, F) :- N >= 1, N1 is N-1, fact(N1, F1), F is N*F1.

?- fact(5,F).
F = 120 ;
false

?- fact(X,120).
ERROR: is/2: Arguments are not sufficiently instantiated
```

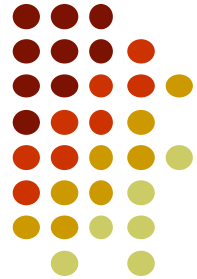
Généralisation d'un prédicat pour le rendre symétrique

- Pour rendre symétrique un prédicat, il faut pouvoir distinguer le cas où une variable est instanciée de celui où elle ne l'est pas.
- Le prédicat var/1 permet cette distinction :

  var(X) réussit si X est une variable non instanciée.
- Le prédicat nonvar/1 permet également cette distinction :

  nonvar(X) réussit si X est une variable instanciée.

Généralisation d'un prédicat pour le rendre symétrique



- Réalisation du prédicat fact symétrique :

```
fact(N,F):- var(N), !, fact(N,2,F).
fact(0, 1).
fact(N, F) :- N >= 1, N1 is N-1, fact(N1, F1), F is N*F1.
% pour retrouver N, on divise F successivement par 2,3,4, etc.
% jusqu'à obtenir 1, le résultat correspond au nombre de divisions effectuées
fact(N,K,1):- !, N is K-1.
fact(N,K,F):- F mod K =:=0, K1 is K+1, F1 is F // K, fact(N,K1,F1).
```

Partie 4
-
Programmation dynamique

Listing

- `listing/0` : liste tous les faits et règles de la base de connaissance
- `listing(pred)` : liste tous les prédicats ayant pour nom pred dans la base de connaissance
- `listing(pred/n)` : liste tous les prédicats d'arité n ayant pour nom pred dans la base de connaissance
- Utile pour savoir si vous avez bien chargé votre base
- SWI-Prolog charge automatiquement un certain nombre de prédicats au démarrage.

assert : création dynamique de faits et de règles

- Un prédicat particulier nommé « **assert** » permet de créer dynamiquement en mémoire de nouveaux faits ou de nouvelles règles.
- Exemples:
  - ?- assert(homme(louis)).
  - ?- assert(fait(not(idiot(jules)))).
  - ?- assert(intelligent(X):-fait(not(idiot(X)))).
  - ?- intelligent(A).

    A = jules

Création dynamique de faits et de règles

- On peut insérer des faits (ou règles) en tête de la liste des faits (ou règles) existant(e)s par le prédicat **asserta**
- On peut insérer des faits (ou règles) en fin de la liste des faits (ou règles) par le prédicat **assertz**

asserta(homme(jean)).

Insère le fait homme(jean) en tête de la liste des faits nommés « homme »

assertz(homme(bernard)).

Insère le fait homme(jean) en fin de la liste des faits nommés « homme »

Exemple de création dynamique de règle

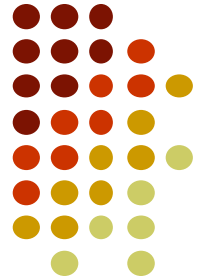
```
?- assert(maxim(A,B,A):-A>B).
true.

?- asserta(maxim(A,B,B):-A=<B).
true.

?- listing(maxim).
:- dynamic maxim/3.

maxim(B, A, A) :-
        B=<A.
maxim(A, B, A) :-
        A>B.
true.
```

Création dynamique de faits et de règles



- Cette possibilité de pouvoir construire des programmes dynamiquement est l'une des caractéristiques de l'Intelligence Artificielle
- Cette possibilité existe aussi en Scheme
- Une autre possibilité de créer des programmes dynamiquement consiste à les produire dans un fichier de texte, puis à les charger dynamiquement par le prédicat **consult**.

Partie 5
-
Modularité

Intérêt

- Il est malcommode d'écrire un programme complexe dans un fichier unique
- On souhaite réutiliser certaines fonctionnalités dans des programmes différents sans avoir à les recopier à chaque fois.
- On souhaite distribuer certaines fonctionnalités à d'autres programmeurs
- Solution : Mettre une fonctionnalité dans un fichier unique

Charger un programme source

- Dans SWI-Prolog, on utilise le sous-menu « Consult »
- Il est aussi possible de taper dans l'interpréteur la commande ['nomFichier1']. ou consult('nomFichier1‘).
- Possibilité de charger plusieurs fichiers

?- ['nomFichier1', …, 'nomFichierN'].

- Possibilité d'effectuer ce même chargement dans un programme source (main.pl).

:- [nomFichier1, …, nomFichierN].

Vérification avant chargement

- En utilisant la forme suivante, Prolog ne vérifie pas s'il a ou pas déjà chargé le fichier.

  :- ['nomFichier1', …, 'nomFichierN'].

- Problème dans des programmes complexes avec des fichiers très longs qui prennent du temps à charger.
- Solution :

  :- ensure_loaded('nomFichier').

Intérêt des modules

- Imaginons un prédicat défini deux fois de manière différente dans deux fichiers chargés
- L'interpréteur va demander si l'on souhaite conserver l'ancienne ou la nouvelle version
- Problème : on souhaite conserver les deux définitions car elles correspondent toutes deux à un contexte particulier (utilisation interne)
- Solution 1 : renommer un des prédicats
- Solution 2 : Utilisation des modules pour cacher certains prédicats

Utilisation des modules

- Déclaration des modules en tête du fichier:

:- module(nomModule,ListePredicatsExportes).

- Exemple :

```
:- module(affichage, [afficherCarre/2]).
```

- Les prédicats exportés sont **publics**
- Les prédicats non exportés sont **privés**
- Chargement de modules grâce à au prédicat use_module :

```
:- use_module(affichage).
```

Librairies

- Chaque implémentation de Prolog fournit en général un certain nombre de prédicats prédéfinis.
- SWI-Prolog en charge automatiquement certains, mais pas tous.
- Possibilité de charger ces librairies prédéfinies avec la commande use_module, mais en précisant que ce sont des librairies. Exemple :

```
:- use_module(library(csv)).
```

- Ces librairies diffèrent selon les implémentations de Prolog

Exemple

- Vous disposez de deux fichiers sources : 'main.pl' qui est votre programme principal, et 'fichier.pl' dans lequel vous avez défini les prédicats ecrire/1, lire/1, ecrire/2, lire/2 qui vous permettent de lire et écrire dans des fichiers.
- Vous ne souhaitez utiliser que les prédicats ecrire/1 et lire/1 dans le programme principal.
- Utilisez les modules pour implémenter ce comportement

Solution

- Dans 'fichier.pl' :

```
:- module(fichier, [ecrire/1, lire/1]).
```

- Dans 'main.pl' :

```
:- use_module(fichier).
```