Arbres

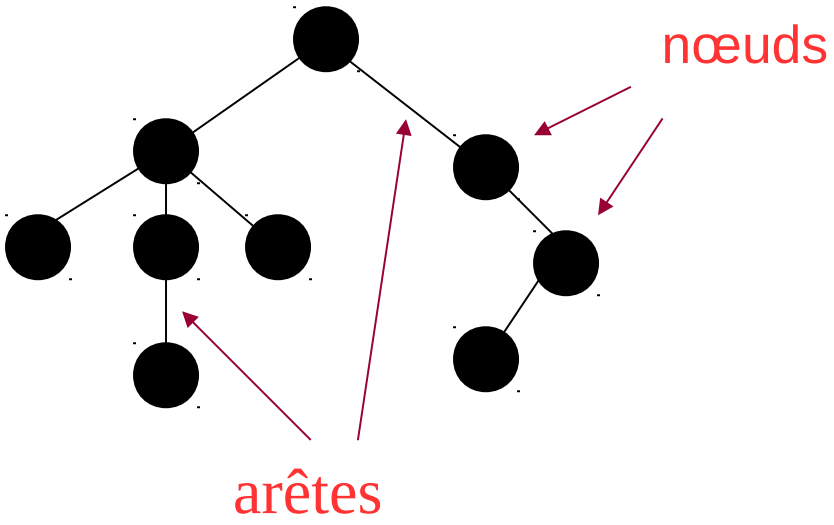
- Un **arbre** est une structure de données organisée de façon hiérarchique, à partir d'un nœud distingué appelé racine.
- Très importante en informatique!

- Arbre de jeux (i.e., Echecs), système de fichiers UNIX/Windows, Arbres Syntaxiques, Expressions Arithmétiques, etc.

- Nous étudierons principalement deux types d'arbres : **Arbres Binaires de Recherche (ABR)** et **Arbres Equilibrés (AVL, 234)**

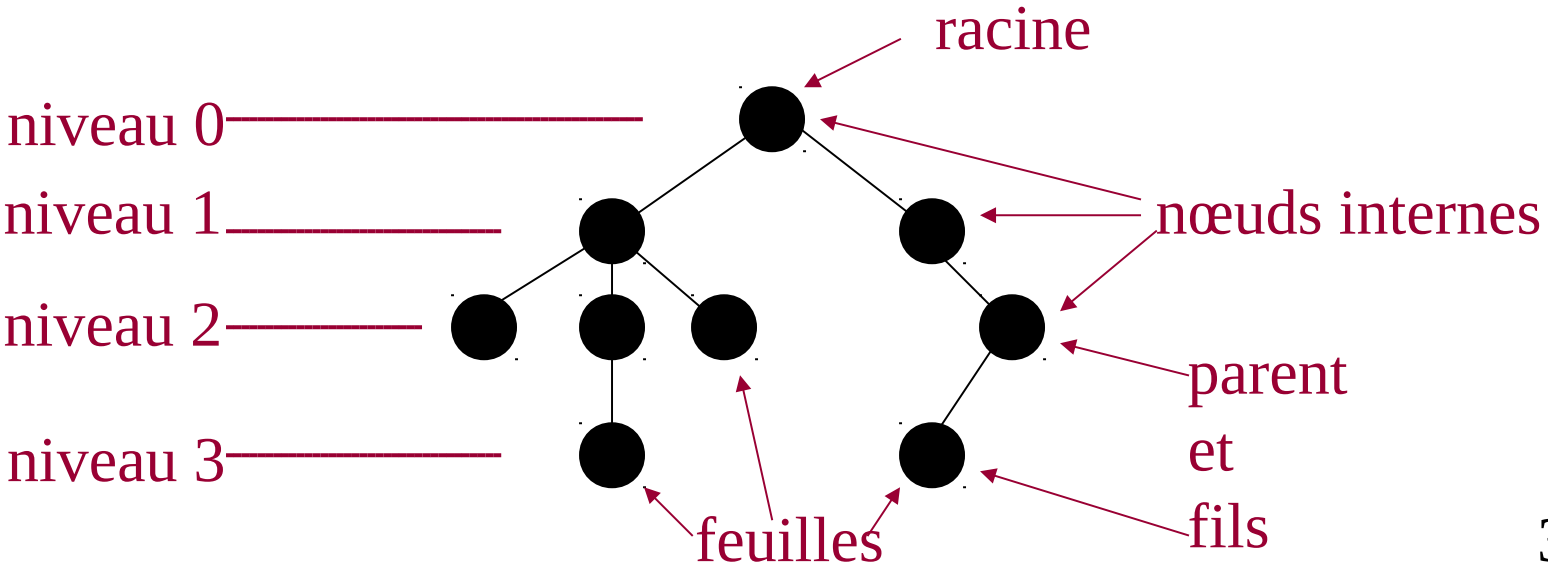
Arbres: définitions

- Un arbre est un ensemble de **Nœuds**, reliés par des **Arêtes**. Entre deux nœuds, il n'existe toujours qu'un seul chemin.



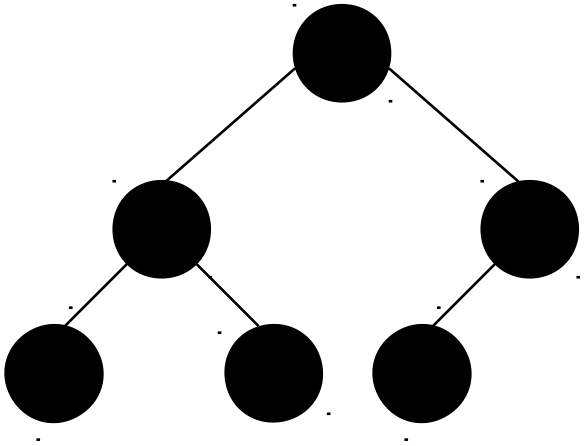
Arbres: définitions

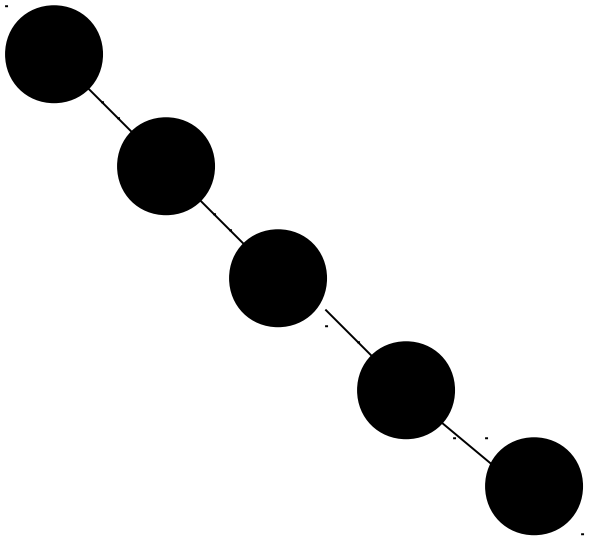
- Les arbres sont enracinés. Une fois la **racine** définie, tous les nœuds admettent un **niveau**.
- Les arbres ont des noeuds **internes** et des **feuilles** (nœuds externes). Chaque noeud (à l'exception de la racine) a un **parent** et admet zéro ou plusieurs **fils**.



Arbres binaires

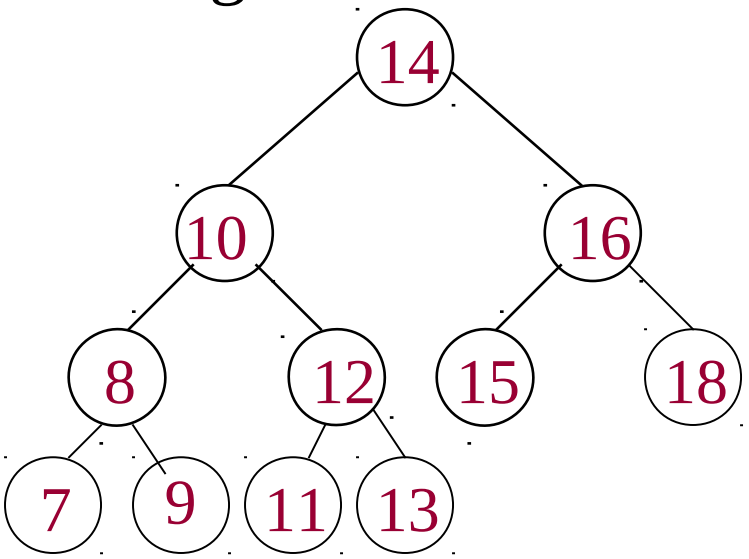
- Un **Arbre Binaire** est un arbre où chaque nœud admet au plus 2 fils.





Arbres Binaires: définitions

- Nœuds d'un arbre contiennent des **clés** (mots, nombres, etc)
- **Arbre Binaire parfait** : les feuilles sont toutes situées dans les deux derniers niveaux. Les feuilles du dernier niveau sont toutes à gauche.

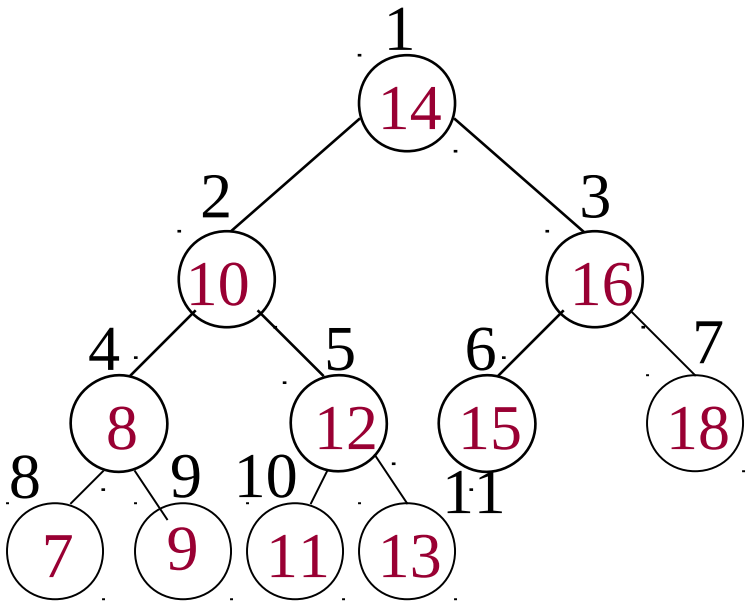


Arbres Binaires: représentation par tableaux

- Un arbre binaire complet peut être représenté par un tableau A avec un accès en O(1) à chaque noeud:

  - Mémoriser les noeuds séquentiellement de la racine aux feuilles et de gauche vers la droite.
  - Fils gauche de A[i] est en A[2i]
  - Fils droit de A[i] est en A[2i + 1]
  - Parent de A[i] est en A[i/2]

Arbres Binaires: représentation par tableau



	1	2	3	4	5	6	7	8	9	10	11
tab A:	14	10	16	8	12	15	18	7	9	11	13

Arbres Binaires: représentation par pointeurs

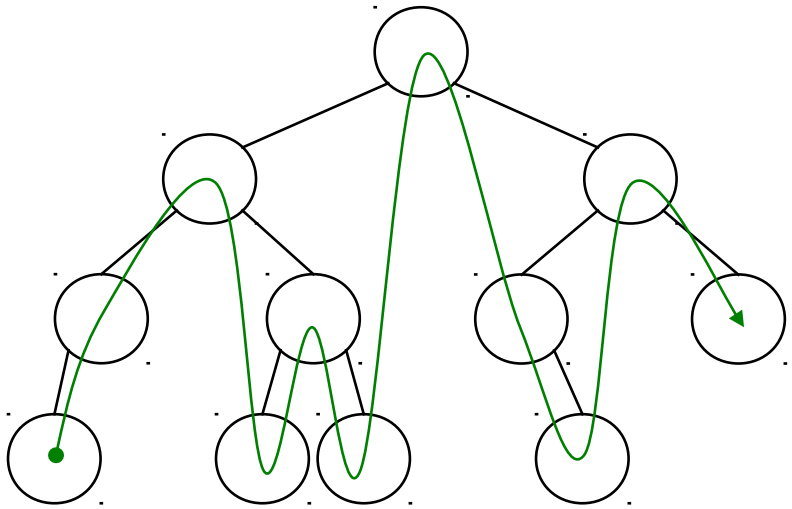
```
typedef struct n{
  int clé;
  struct n *fGauche, *fDroit;
}nœud_t, *pnoeud_t;

typedef pnoeud_t Arbre_t;
```

Parcours infixé, inOrder

infixé est décrit réursivement :

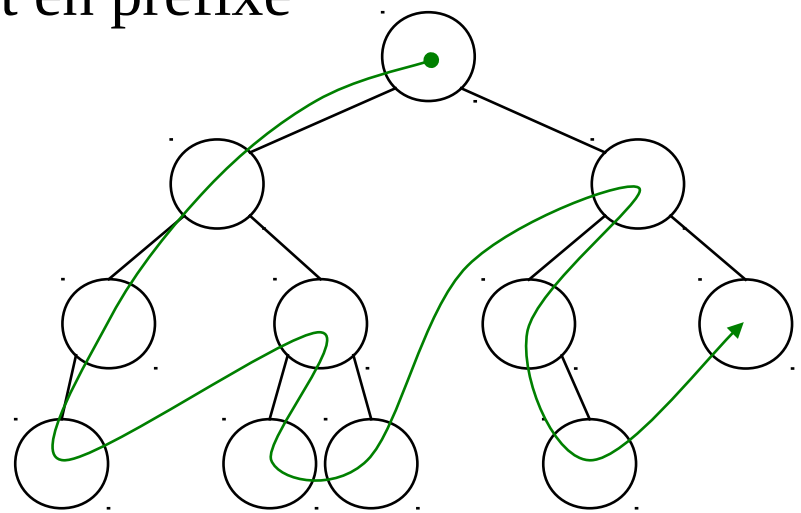
- Visiter le sous-arbre gauche en **infixé**
- **Traiter la racine (printf,...)**
- Visiter le sous-arbre droit en **infixé**



Parcours préfixé, PreOrder

préfixé est décrit réursivement :

- **Visiter la racine**
- Visiter le sous-arbre gauche en préfixé
- Visiter le sous-arbre droit en préfixé



Parcours: non-récursif

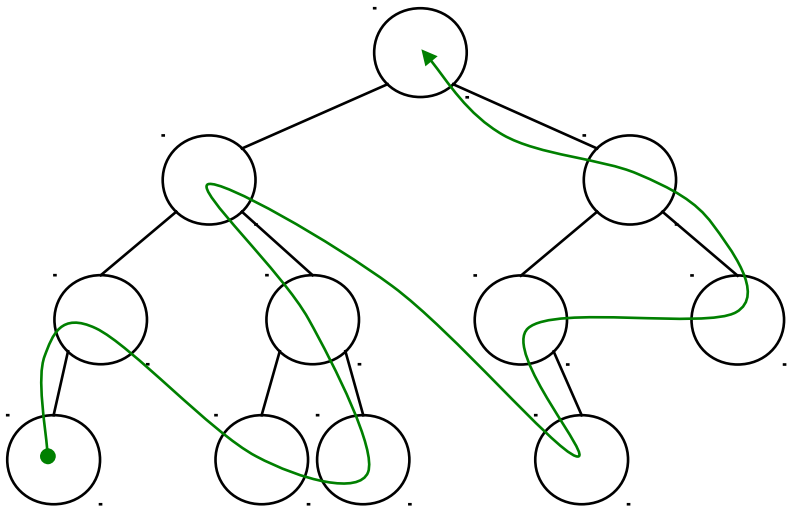
PréOrdre itératif en utilisant une Pile.

```
Pile S
empiler racine dans S
répéter jusqu'à S=ϕ
  v = dépiler S
  si v <> nil
     visiter v
     empiler le fils droit de v dans S
     empiler le fils gauche de v dans S
```

Parcours postfixé, postOrder

postfixé est décrit réursivement :

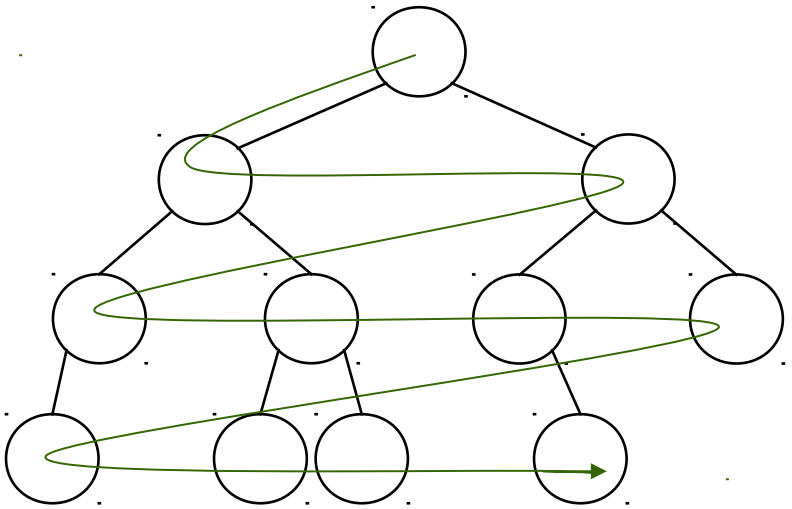
- Visiter le sous-arbre gauche en **postfixé**
- Visiter le sous-arbre droit en **postfixé**
- **Visiter la racine**



Parcours par niveau: levelOrdre

LevelOrdre visite les noeuds niveau par niveau depuis la racine:

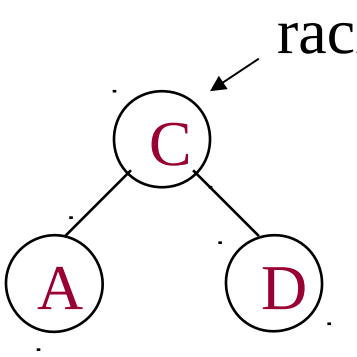
- Peut être décrit facilement en utilisant une File **(Comment??)**
- Parcours appelé "Breadth First Search" (parcours en largeur) dans les graphes

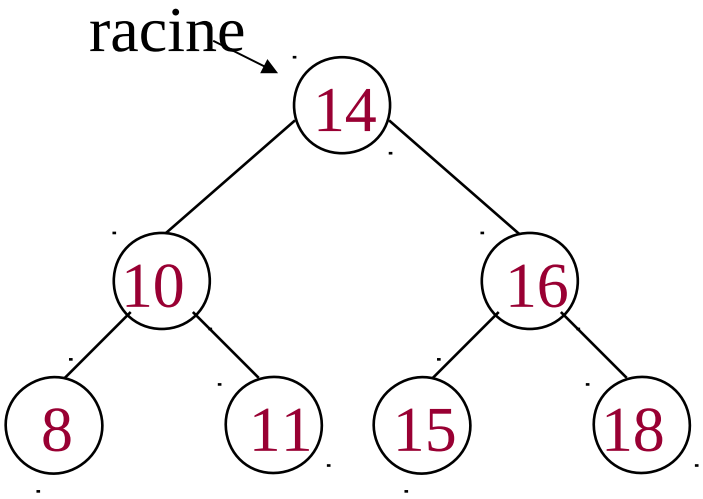


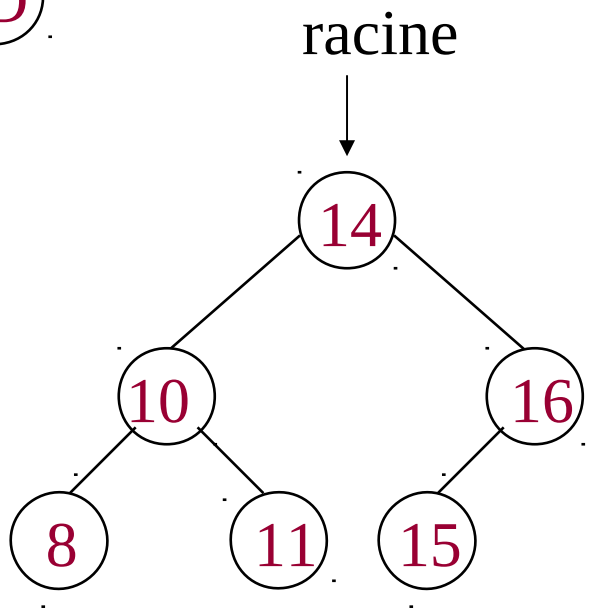
Arbre Binaire de Recherche

- Un **Arbre Binaire de Recherche (ABR)** est un arbre binaire avec les propriétés suivantes :
  - La clé associée à un noeud est *supérieur* aux clés des nœuds de son sous-arbre gauche
  - La clé associée à un noeud est *inférieur* aux clés des nœuds de son sous-arbre droit

Arbre Binaire de Recherche: Exemples







Arbre Binaire de Recherche

- ABR est un arbre avec la **propriété** suivante :

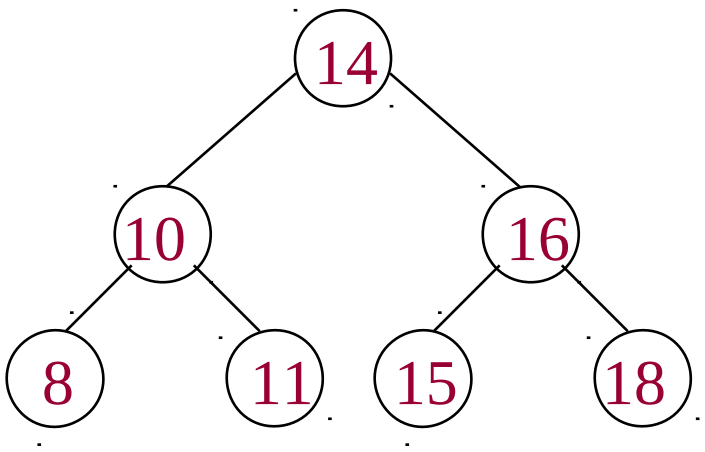
Clé.fGauche < Clé.parent < Clé.fDroit

NOTER! Le parcours InOrdre visite les clés dans l'ordre croissant.

```
void inOrdre(Arbre racine) {
    inOrdre(racine->fGauche)
    print(racine->key)
    inOrdre(racine->fDroit)
}
```

ABR: InOrdre

Exemple:

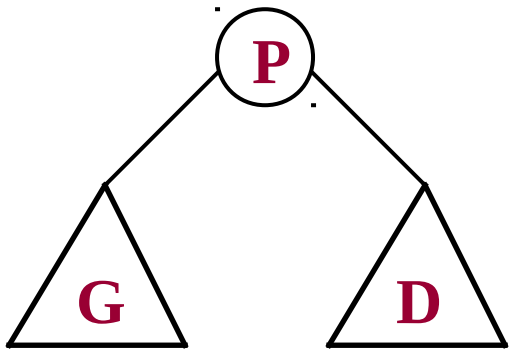


InOrdre visites :

(8)
(10)
(11)
(14)
(15)
(16)
(18)

ABR : Rechercher un élément

Soit un ABR :



Problème: rechercher un noeud avec une clé x ?

ABR : Rechercher un élément

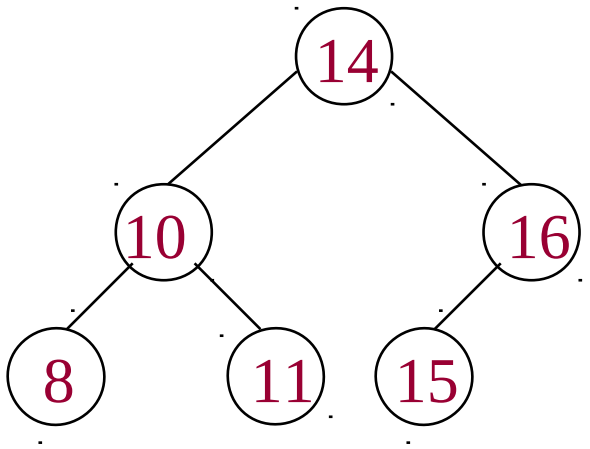
rechercher(racine, x)

comparer x à la clé de racine:

- si x = clé return
- si x < clé => chercher dans G
- si x > clé => chercher dans D

chercher de la même manière dans G ou D

Exemple:



x=8 (oui) x=17 (non)

ABR : Rechercher un élément

```
bool searchABR(Arbre racine; typeCle clé){
   if (racine==NULL)  return false
   if (racine->clé==clé)
       return true;
   else if (key < racine->clé)
       return searchABR(racine->fGauche, clé);
   else
       return searchABR(racine->fDroit, clé)
}
```

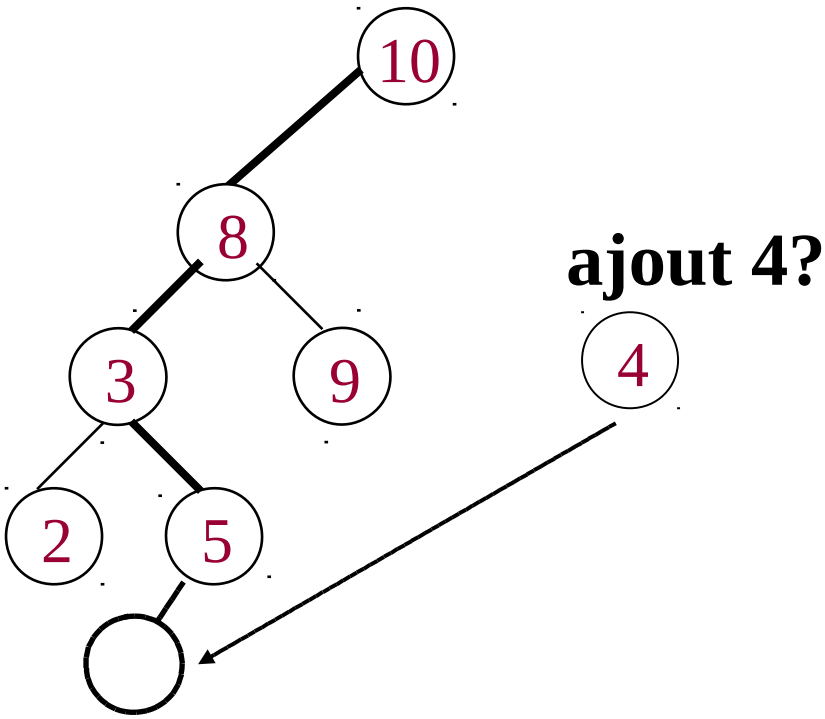
Donner une version itérative ?

ABR : Ajout d'un élément

Comment ajouter une clé?

La même procédure que searchABR s'applique:
Déterminer la position d'insertion par searchABR.
Ajouter la nouvelle clé si la recherche échoue.

Exemple:

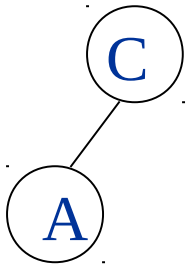


Construction d'un ABR

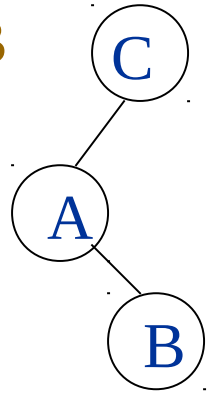
Exemple: ajouter **C A B L M** **(dans l'ordre!)**

1) Ajouter C

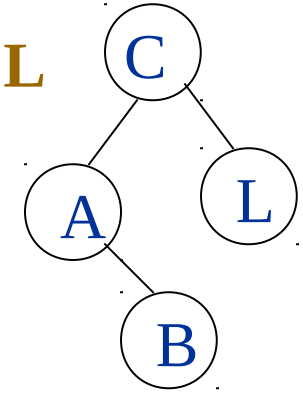
2) ajouter A



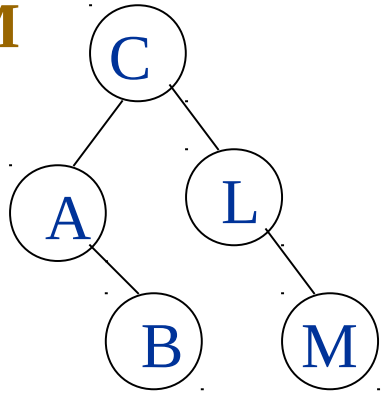
3) ajouter B



4) Ajouter L



5) Ajouter M

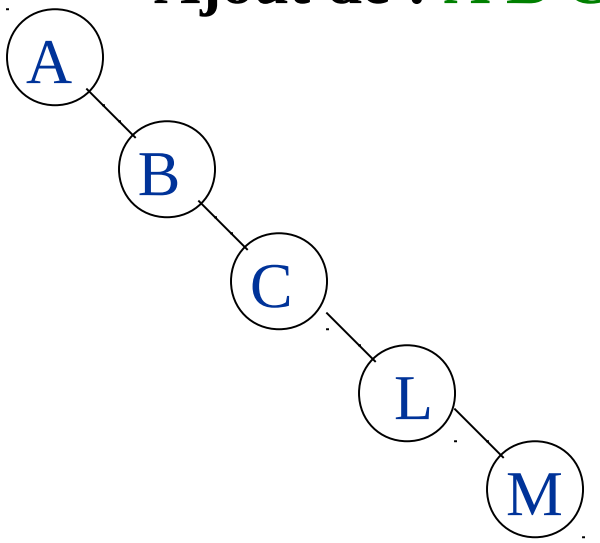


Construction d'un ABR

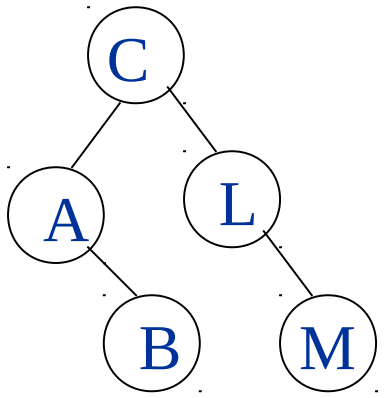
L'ABR est-il unique pour une séquence de lettres A B C L M ?

NON! différentes séquences donnent différents ABR

Ajout de : A B C L M



Ajout de : C A B L M



Trier avec un ABR

Soit un ABR, peut-on afficher les clés dans l'ordre?

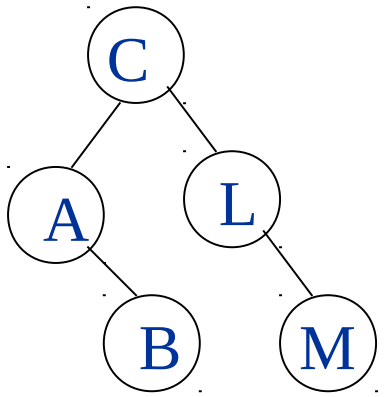
Visiter l'ABR avec un parcours InOrdre:

- visiter le sous-arbre gauche
- afficher racine
- visiter le sous-arbre droit

Comment trouver le minimum?

Comment trouver le maximum?

Example:



InOrdre affichage:

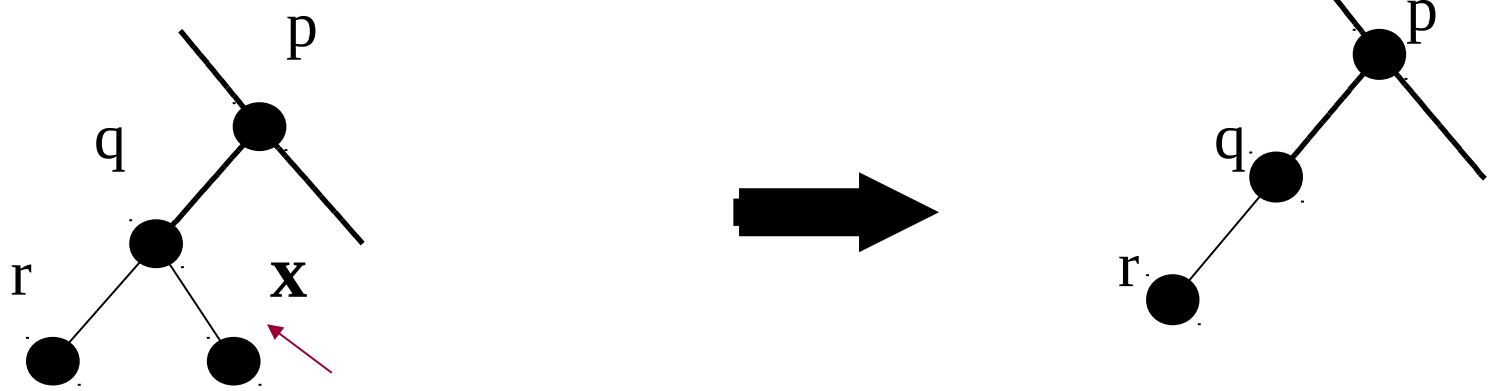
A B C L M

ABR : supprimer un élément

Pour supprimer un nœud contenant x,

rechercher x, une fois trouvé appliquer l'un des trois cas suivants:

CAS A: x est une feuille

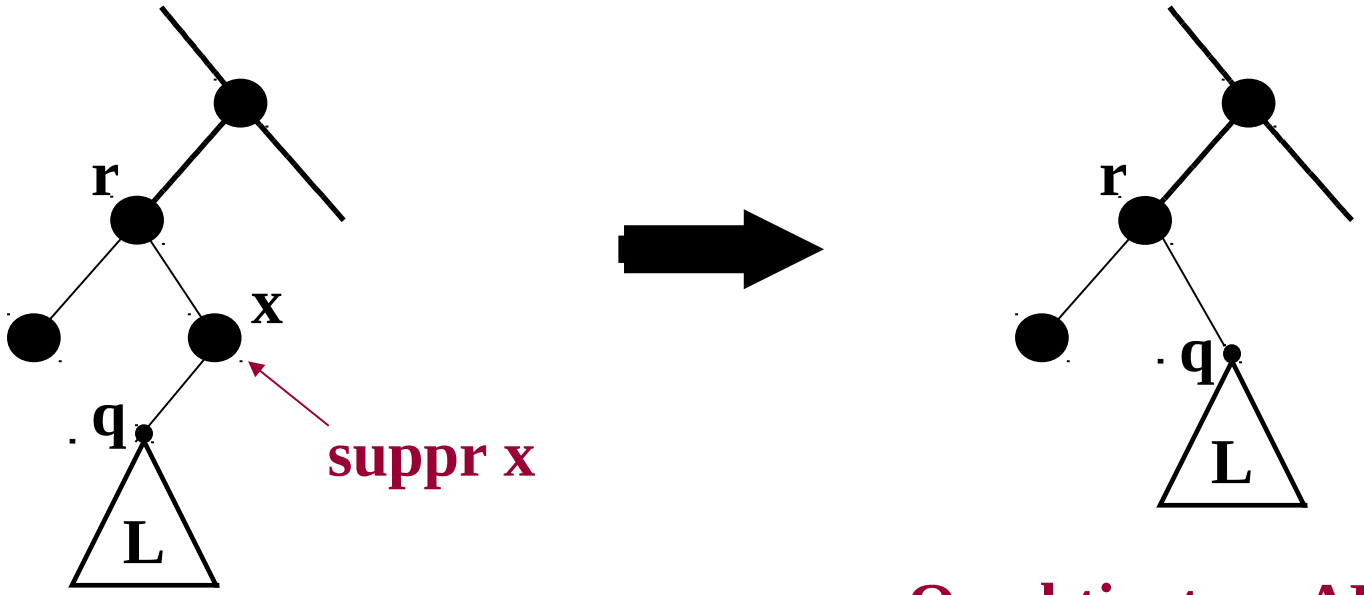


supprimer x

On obtient un ABR

ABR : supprimer un élément

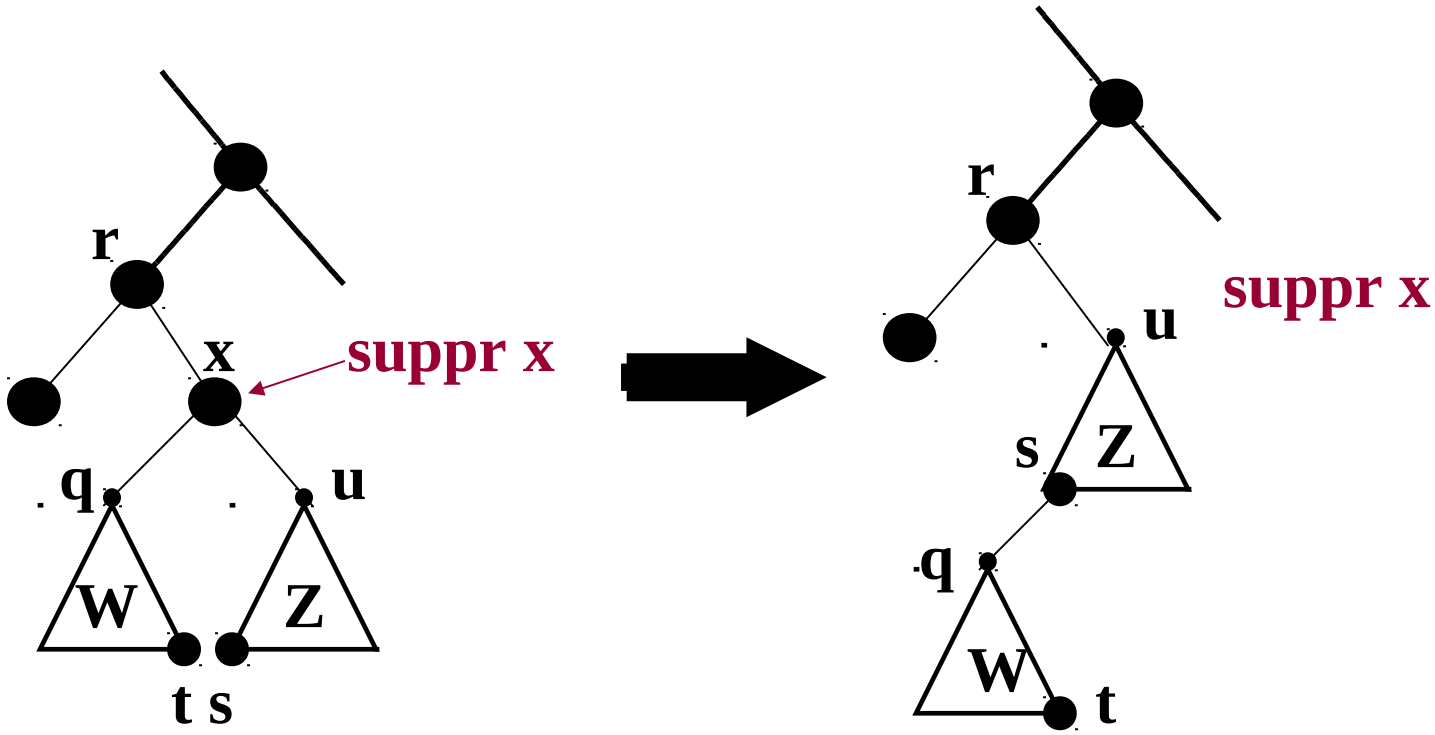
Cas B: x est un nœud interne avec un seul sous-arbre



On obtient un ABR

ABR : supprimer un élément

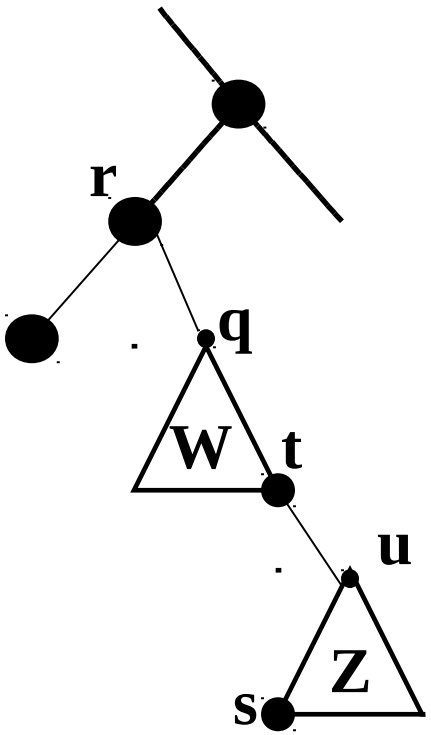
Cas C: x est un nœud interne avec 2 sous-arbres



propriété ABR est conservée

ABR : supprimer un élément

Cas C suite: … ou encore comme suit



q < x < u

⇒ **q est inférieur au plus petit élément de Z**

⇒ **r est supérieur au plus grand élément de W**

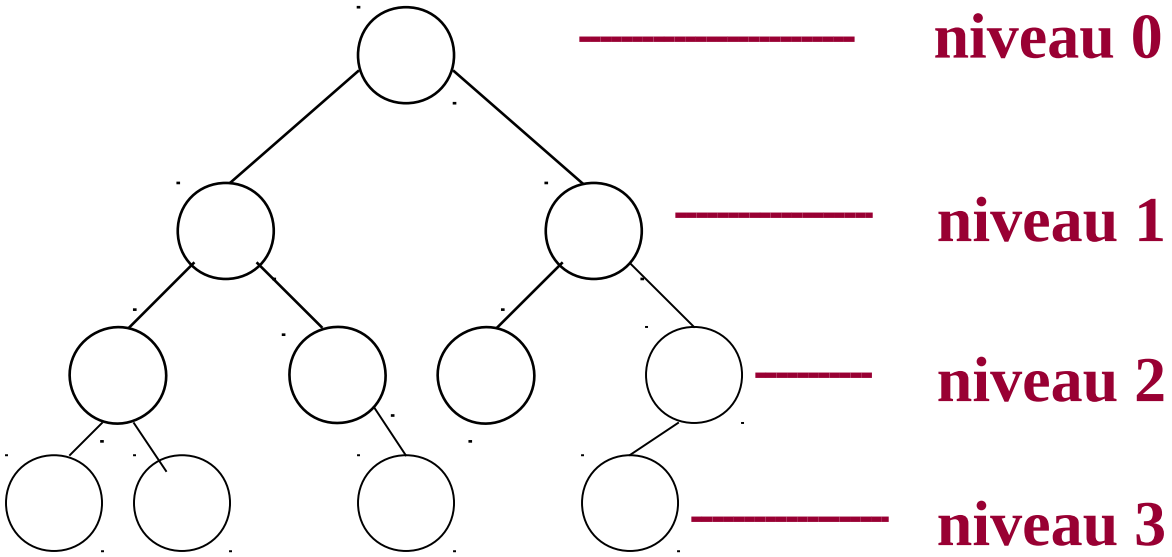
D'autres façons ?

ABR : Compléxité de rechercher

- Quelle est la compléxité de **`searchABR`** ?
- Dépend de :
  - la clé x
  - des autres données
  - De la forme de l'arbre

Analyse de la compléxité : On est intéréssé par la compléxité dans le meilleur cas, pire cas et en moyenne

ABR : Complexité de rechercher



(h =3)

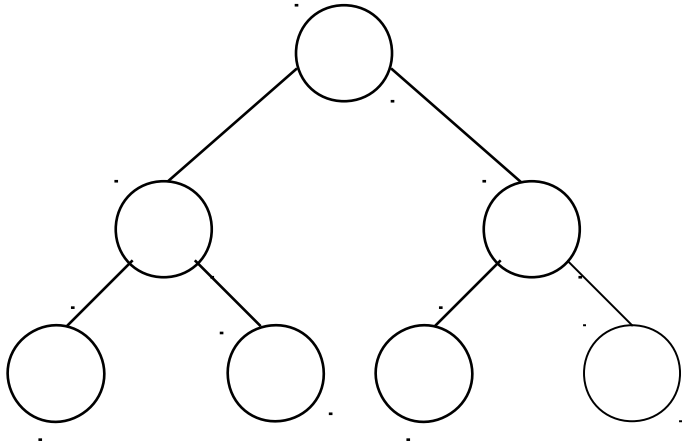
- **hauteur d'un ABR = niveau max**
- **hauteur d'un noeud**

  h(x) = 0 si x est la racine

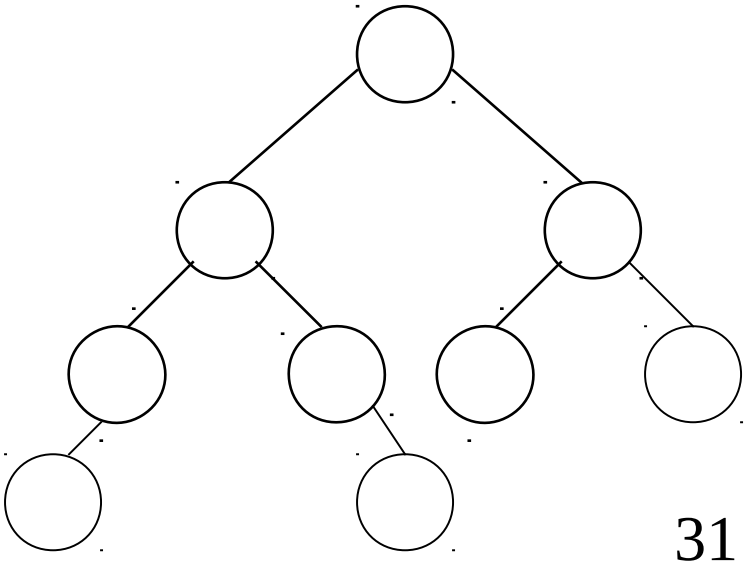
  h(x) = 1+ h(y), y = pere(x)
- **hauteur d'un ABR B : h(B) = max{h(x), x nœud de B}**

ABR : Complexité de rechercher

Si tout les nœuds de l'arbre existent : **ABR plein**



Si tout les nœuds existent sauf ceux du dernier niveau : **niveau-min ABR**



ABR : Complexité de rechercher

Théorème:

Un ABR plein (complet) de hauteur h a $2^{h+1} - 1$ noeuds

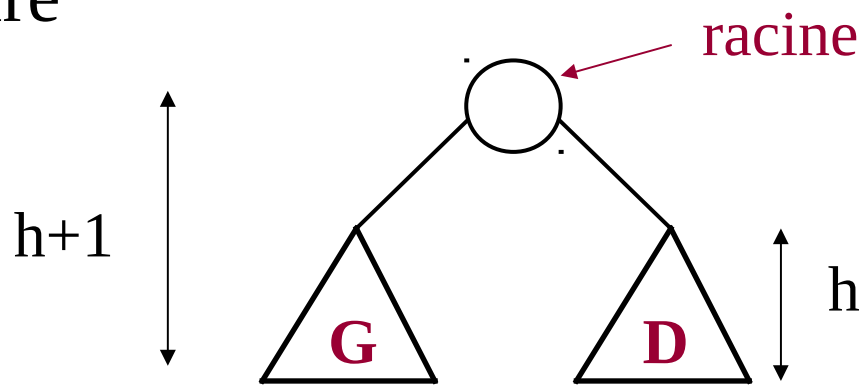
Preuve: Par induction

Cas de base: un arbre de hauteur 0 a 1 nœud (racine)

Hypothèse inductive: Supposant qu'un ABR de hauteur h a $2^{h+1} - 1$ noeuds

ABR : Complexité de rechercher

Etape d'induction: Connecter 2 ABR de hauteur h pour construire un ABR de hauteur h+1. On a besoin d'ajouter un noeud supplémentaire



Par hypothèse inductive le nouveau nombre de noeuds est

$(2^{h+1} - 1) + (2^{h+1} - 1) + 1 = 2^{h+2} - 1$ ……CQFD!

Ou encore : $n = 1+2+\ldots+2^{h} = 2^{h+1} - 1$

ABR : Complexité de rechercher

Lemme 1: pour un ABR ayant n nœud et de hauteur h :

$$\lfloor \log_2 n \rfloor <= h <= n - 1$$

Remarque: Un ABR *parfait* avec n noeuds a pour hauteur

$$h = \lfloor \log_2 n \rfloor$$

car

$$2^h <= n <= 2^{h+1} - 1$$

ABR : Complexité de rechercher

Conséquence : pour un ABR plein avec N noeuds la compléxité de **`searchABR:`**

meilleur cas ……. O(1)

Pire cas ……. O(log N)

en moyenne ……. ???

ABR : Complexité de rechercher

⇒ complexité en moyenne pour une recherche dans un ABR plein est une fonction logarithmique du nombre de nœuds de l'arbre

Complexité en moyenne pour des ABR quelconque est approximativement 39% plus chère que la recherche dans un ABR plein pour le même nombres de nœuds :

$$T_{avg}(N) \approx 1.386 \log_2 N - 3$$

ABR : complexité de rechercher

- Maintenant que nous connaissons la compléxité de **`searchABR`** que peut-on dire des autres opérations?

Insertion ……. O(log N)
Suppression ……. O(log N)
Trouver le Min ……. O(log N)
Trouver le Max ……. O(log N)

ABR : Complexité de rechercher

- En résumé, il est nécessaire d'avoir un ABR plein ou niveau-min ABR

  ↳ garder un arbre le plus équilibré possible

  à tout moment (Arbre AVL)