

R3.02

Développement efficace

Cours 2 – arbres binaires & parcours

Hervé Blanchon

Université Grenoble Alpes

IUT 2 – Département Informatique

Plan du cours



Notion d'arbre



Terminologie



Arbres binaires de recherche (ABR)



un Type Abstrait de Données



une définition inductive



Parcours en profondeur



3 parcours : préfixé, infixé, postfixé

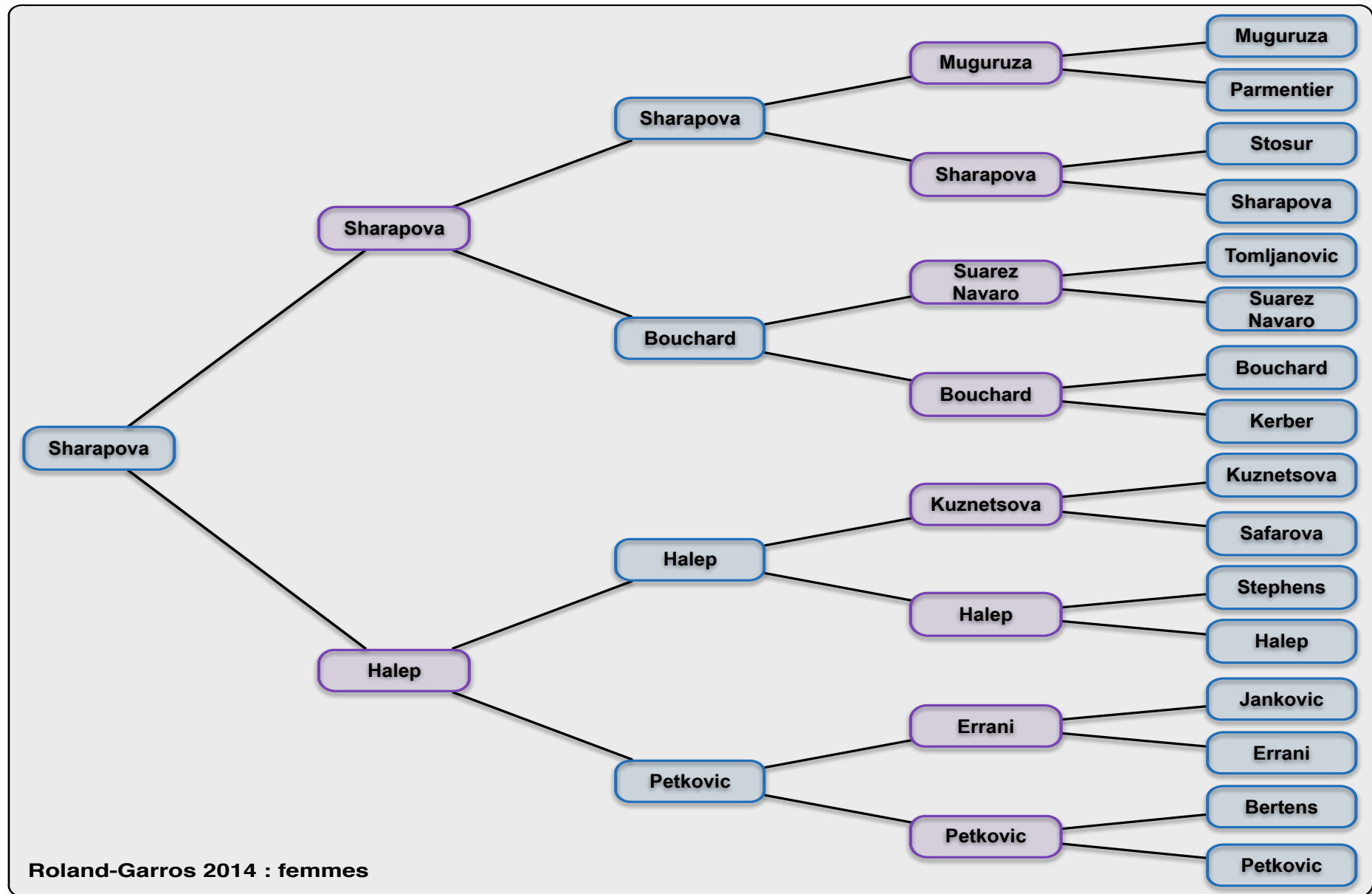


Recherche dans un ABR

Où je comprends l'intérêt des arbres à travers quelques exemples !

NOTION D'ARBRE

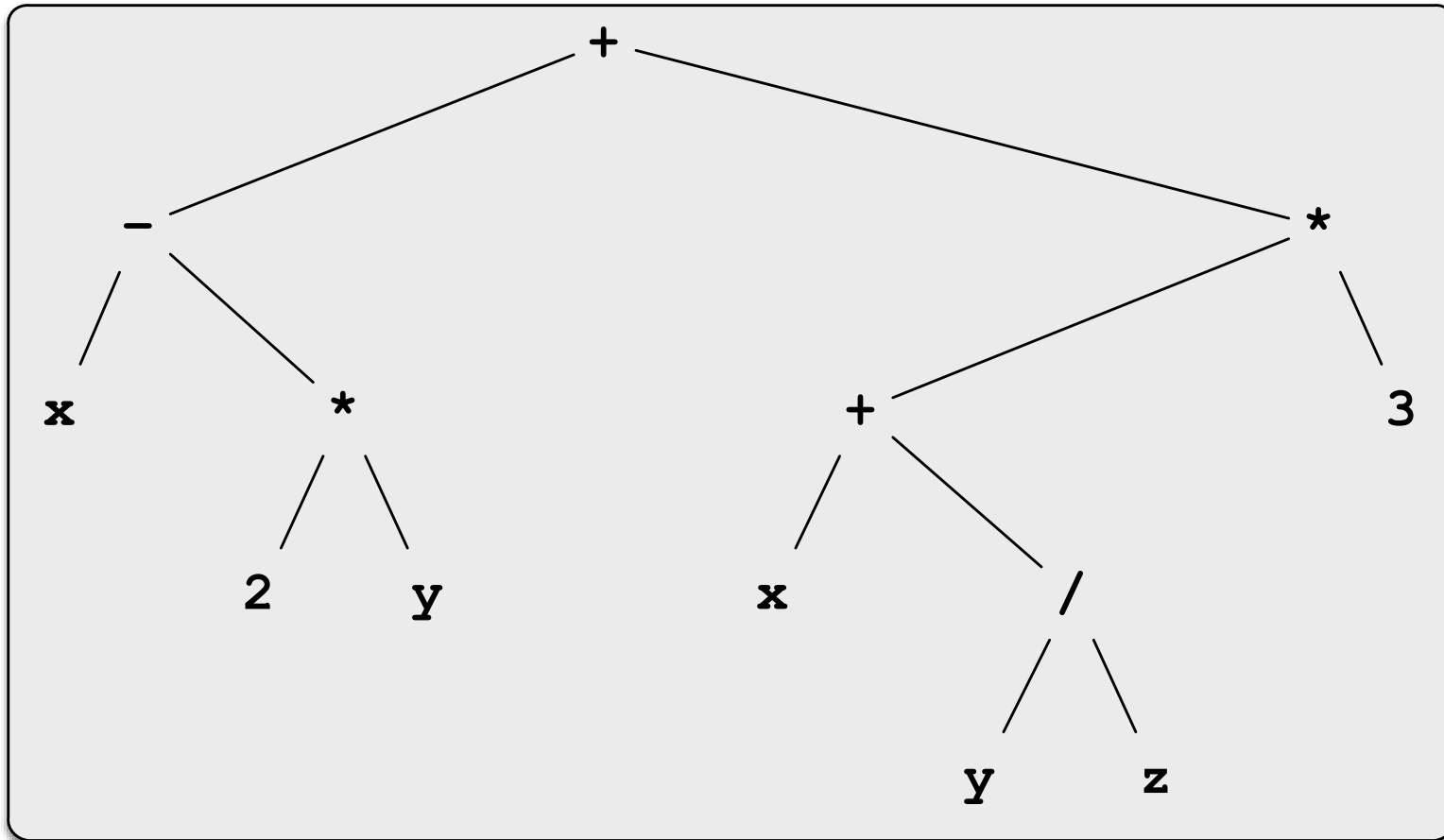
Un arbre pour ...



Un arbre pour ...

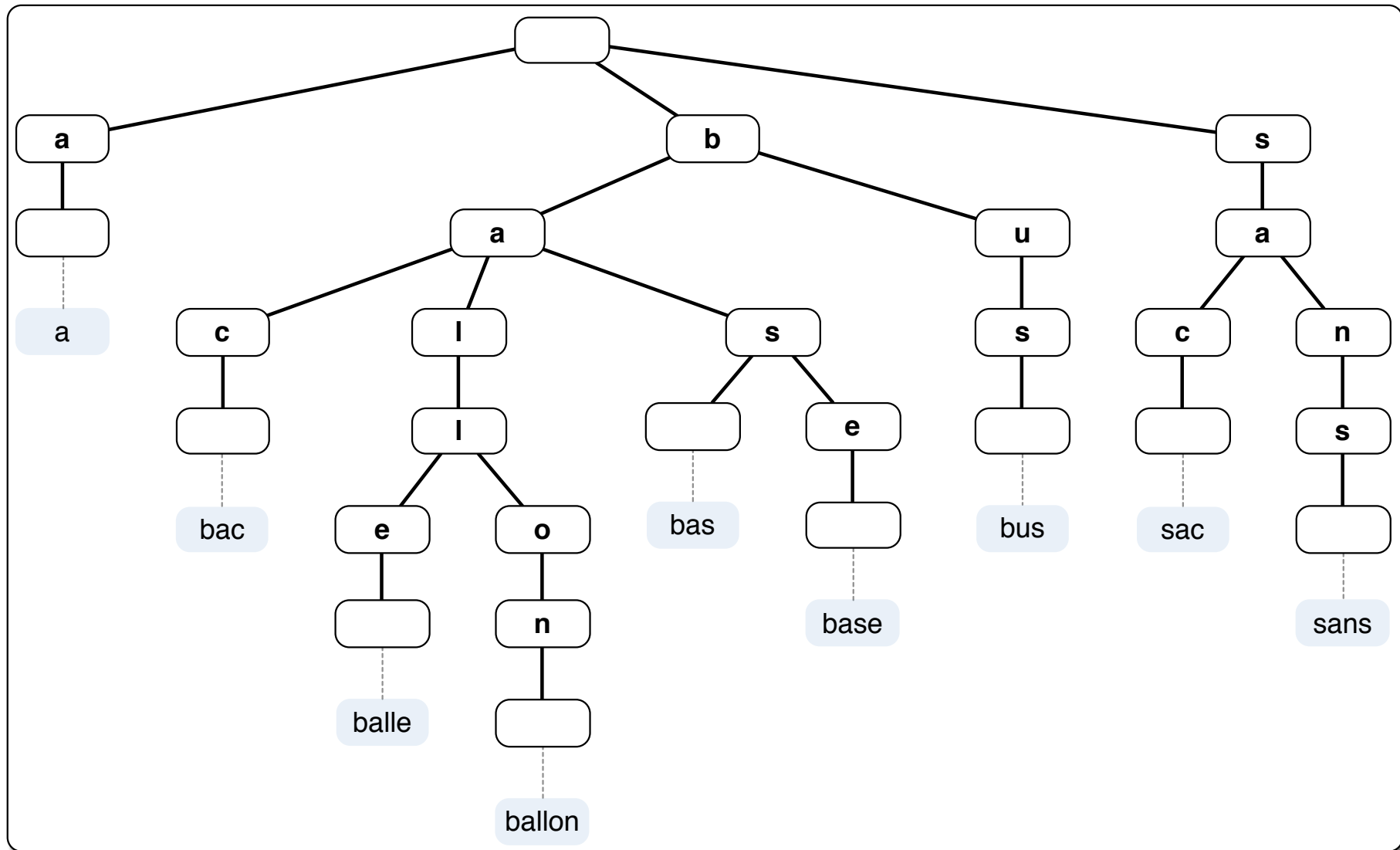
... représenter une expression arithmétique dans un programme lors de la compilation

$(x - (2 * y)) + ((x + (y / z)) * 3)$



Un arbre pour ...

... représenter un dictionnaire



Où j'apprends à nommer les choses !

TERMINOLOGIE

Terminologie usuelle

(1/4)

 **Nœud** : élément d'arbre

 A, B, C, ..., I, J, K, L, M

 **Fils**

 de J : D, L

 de D : B, E, I

 de A, C : aucun

 **Père**

 de E : D

 de L : J

 de J : aucun


 **Racine** : nœud sans père

 J

 **Feuille** : nœud sans fils


 A, C, F, H, K, M

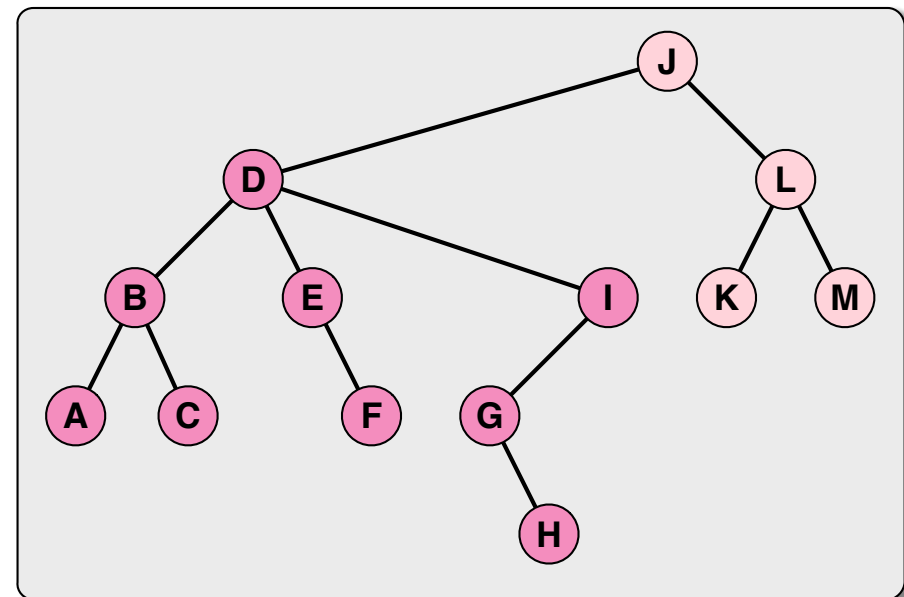
 **sous-arbre de racine r**

 ensemble des nœuds dans la descendance de r (ses fils)


 sous arbre de racine D

 **Propriété**

 tous les nœuds, sauf la racine, n'ont qu'un seul père



Frères

 nœuds ayant le même père

 frères de B : E, I

Degré d'un nœud r

 nombre de fils de r

 degré de D = 3

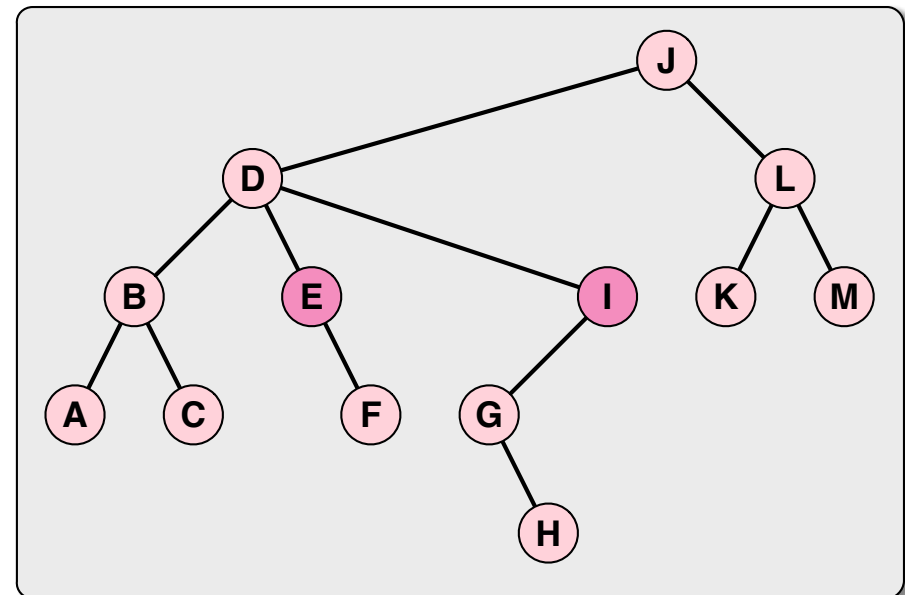
 degré de E = 1

 degré de F = 0

Degré d'un arbre a

 $\max(\text{degrés de ses nœuds})$

 $\text{degré}(D) = 3$



Terminologie usuelle

(3/4)



Arbre binaire



degré = 2



Arbre binaire complet



chaque niveau est rempli

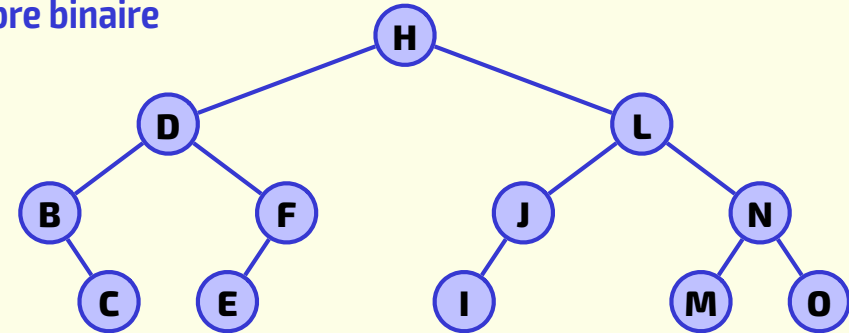


Arbre binaire parfait (semi complet)

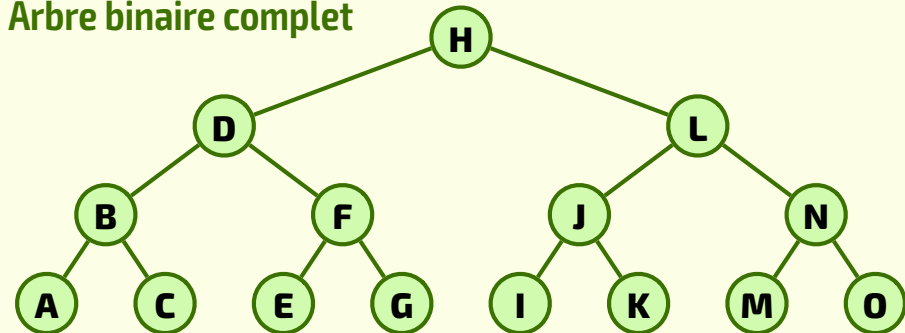


chaque niveau est rempli
sauf éventuellement le
dernier, dans ce cas les
nœuds (feuilles) sont
groupés le plus à gauche
possible

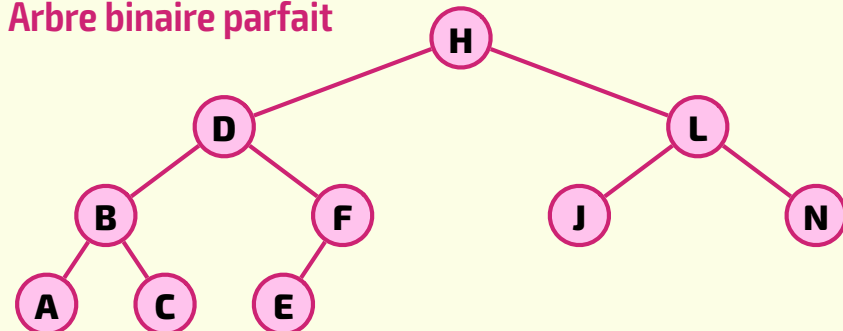
Arbre binaire






Arbre binaire complet

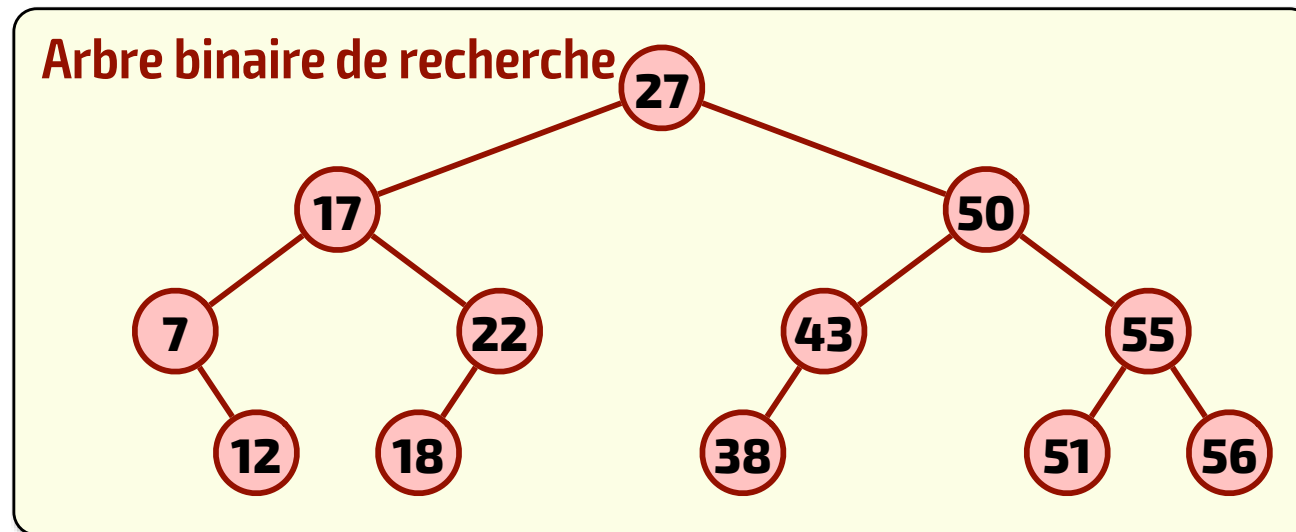


Arbre binaire parfait



Arbre binaire de recherche

-  un arbre binaire tel que **pour tout nœud n** :
 -  l'info. portée par tous les nœuds du **sous-arbre gauche** de **n** est **inférieure ou égale** à l'info portée par **n**
 -  l'info portée par tous les nœuds du **sous-arbre droit** de **n** est **strictement supérieure** à l'info portée par **n**

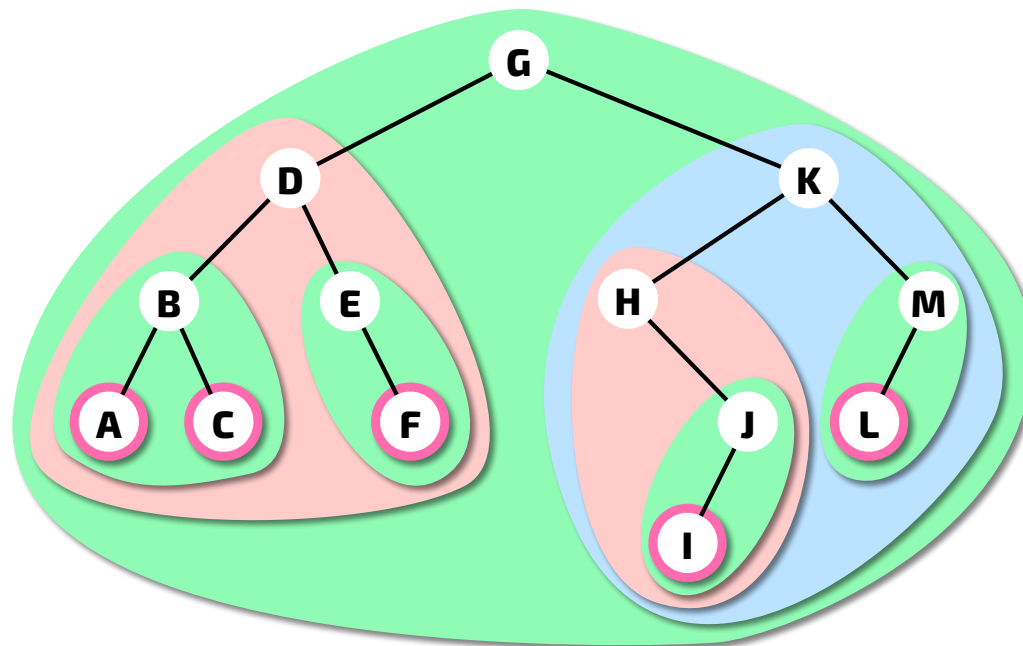


Où je revois la définition inductive d'un type de données !

ARBRE BINAIRE DE RECHERCHE










Le TAD arbre binaire de recherche

- Un ABR est un arbre binaire particulier !
- Dans ABR pour un nœud **n** quelconque,
 - si **n** est une feuille, **n** est un ABR (**A**, **C**, **F**, **I**, **L**)
 - si **n** a seulement un **fil droit**, ce fil droit est **un ABR** (**E**, **H**)
 - il porte des informations **strictement supérieures** à l'information portée par **n**
 - si **n** a seulement un **fil gauche**, ce fil gauche est **un ABR** (**J**, **M**)
 - il porte des informations **inférieures ou égales** à l'information portée par **n**
 - si **n** a **deux fils**, ces deux fils sont **des ABR** (**B**, **D**, **G**, **K**)










Le TAD arbre binaire de recherche

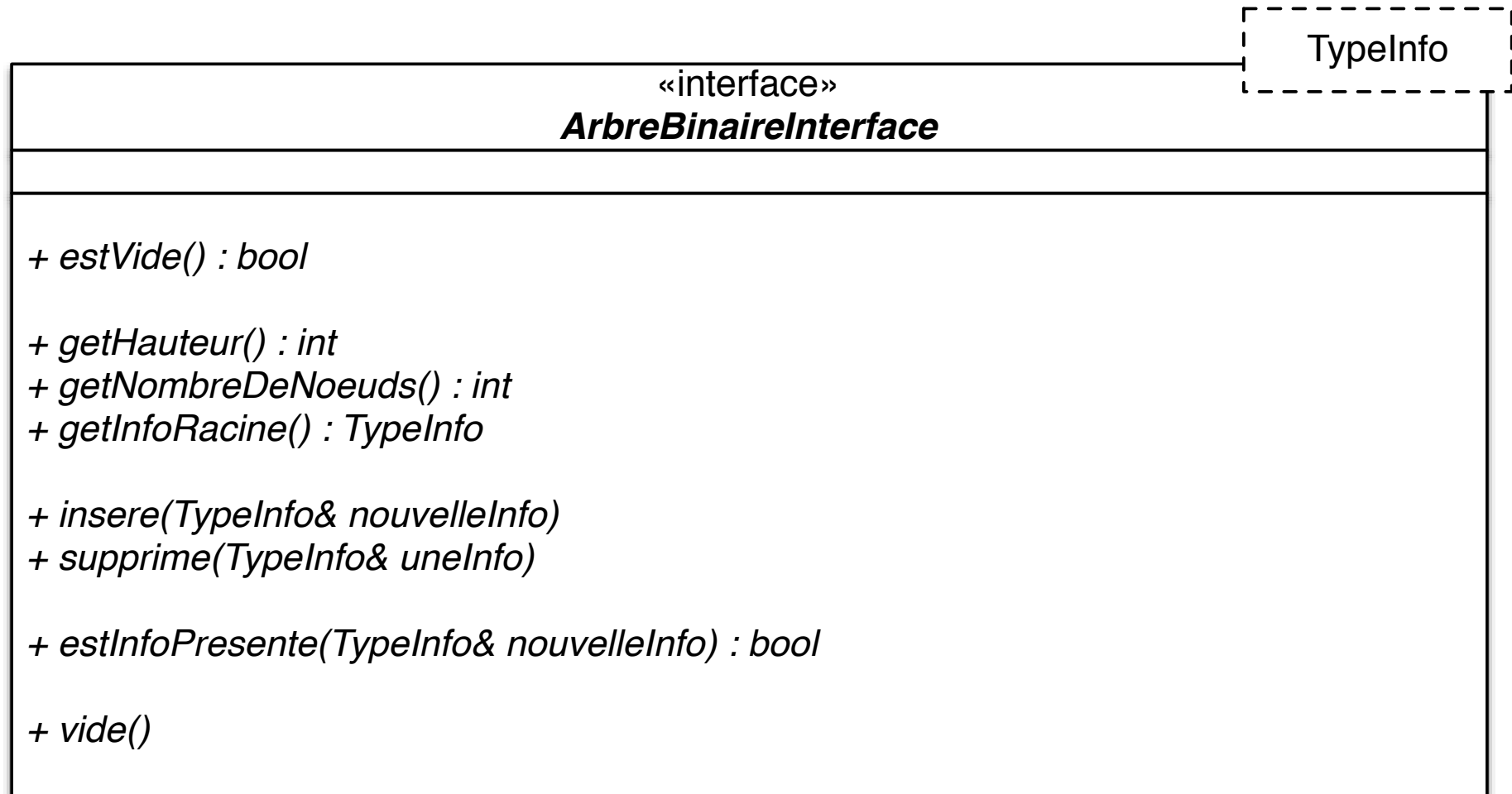
Les opérations associées

-  construit() : ABR
-  estVide() : bool
-  estInfoPresente(unInfo) : bool
-  getHauteur() : int
-  getNombreDeNoeuds() : int
-  getInfoRacine() : info *{exception si impossible}*
-  insere(nouvelleInfo) *{reste un ABR}*
-  supprime(info) *{reste un ABR}*
-  vide()

Le TAD arbre binaire de recherche

-  Le **TAD ABR** sera décrit (*implanté*) au moyen d'une classe abstraite **ArbreBinaireInterface** qui ...
 -  définira uniquement les opérations comme des méthodes virtuelles pures
-  Des classes planteront effectivement un **ABR** en héritant de cette classe abstraite, et ...
 -  définiront la structure de données utilisée pour représenter l'**ABR**
 -  ici le **type arbre dynamique de nœuds binaires**
 -  planteront les opérations d'**ABR** sur la structure de données
 -  ici des **méthodes sur des nœuds binaires**

Le TAD AB : ArbreBinaireInterface




```
template<class TypeInfo>
class ArbreBinaireInterface {
public:
    /** Teste si cet arbre binaire (this) est vide ou non.
     * @return vrai si cet arbre binaire est vide, faux sinon. */
    virtual bool estVide() const = 0;

    /** Retourne la hauteur de cet arbre binaire (this).
     * @return La hauteur de cet arbre binaire. */
    virtual int getHauteur() const = 0;

    /** Retourne le nombre de nœuds de cet arbre binaire (this).
     * @return Le nombre de nœuds de cet arbre binaire. */
    virtual int getNombreDeNoeuds() const = 0;

    /** Retourne l'information disponible à la racine de cet arbre binaire (this).
     * @pre L'arbre binaire n'est pas vide.
     * @post L'information portée par la racine a été retournée, cet arbre binaire est inchangé.
     * @return L'information portée la racine de cet arbre. */
    virtual TypeInfo getInfoRacine() const = 0;

    /** Insère un nouveau nœud contenant la nouvelle information dans cet arbre binaire.
     * @param nouvelleInfo l'information porté par le nouveau nœud.
     * @post Cet arbre binaire contient un nouveau noeud contenant nouvelleInfo.
     * @return vrai si l'ajout est réussit, faux sinon. */
    virtual bool insere(const TypeInfo& nouvelleInfo) = 0;

    ...
}
```

```
...

/** Supprime le premier nœud contenant uneInfo dans cet arbre binaire.
    @param uneInfo La valeur de d'information à supprimer cet arbre binaire.
    @return vrai si la suppression est réussie, faux sinon. */
virtual bool supprime(const TypeInfo& uneInfo) = 0;

/** Supprime tous les nœuds de cet arbre binaire. */
virtual void vide() = 0;

/** Teste si uneInfo est présente dans cet arbre binaire.
    @post L'arbre binaire est inchangé.
    @param uneInfo L'information cherchée.
    @return True si uneInfo est présente dans cet arbre ; false sinon. */
virtual bool estInfoPresente(const TypeInfo& uneInfo) const = 0;

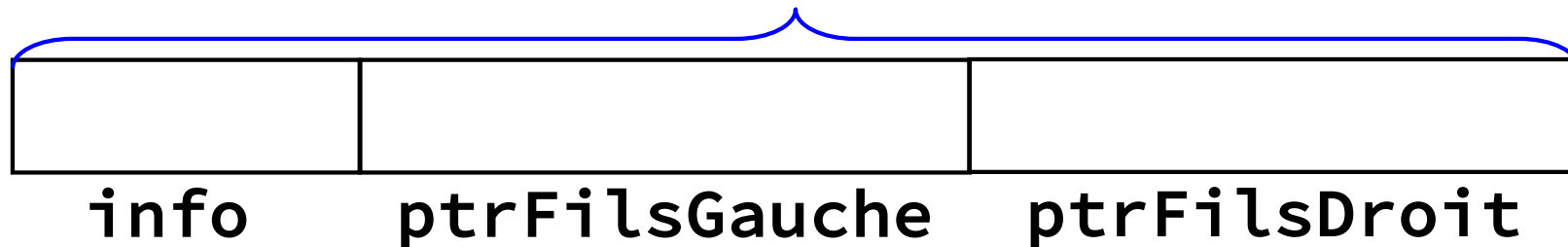
}; // end ArbreBinaireInterface
```

Classe NoeudBinaire

■ On va définir la classe `NoeudBinaire` pour représenter un nœud d'un arbre binaire de recherche

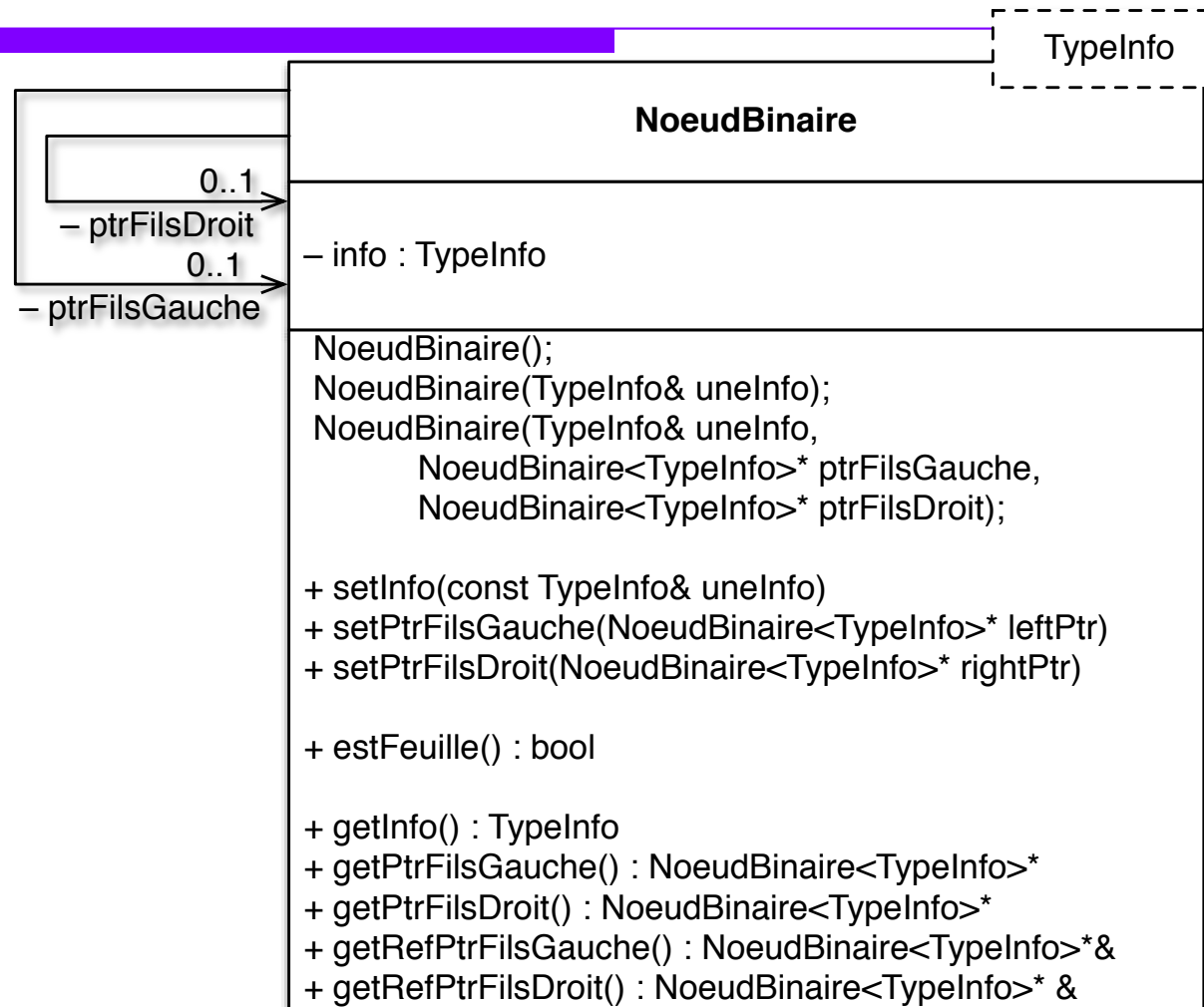
■ Un `NoeudBinaire` à **trois attributs** :

`NoeudBinaire`



■ On propose une classe modèle (template) pour manipuler des arbres binaires de recherche avec n'importe quelle `info` de type comparable

Classe NoeudBinaire



Note



un **NoeudBinaire** à pour attribut 2 pointeurs sur un **NoeudBinaire** gauche (**ptrFilsGauche**) et un **NoeudBinaire** droit (**ptrFilsDroit**)



chacun peuvent être un pointeur nul (nullptr en C++11)

```
template<class TypeInfo>
class NoeudBinaire {
private:
    TypeInfo info; // information portée par le noeud binaire
    NoeudBinaire<TypeInfo>* ptrFilsGauche; // pointeur sur le sous-arbre gauche
    NoeudBinaire<TypeInfo>* ptrFilsDroit; // pointeur sur le sous-arbre droit

public:
    NoeudBinaire(); // Constructeur par défaut de ce Noeud
    // Constructeur avec une information uniquement (une feuille => pas de sous-arbre)
    NoeudBinaire(const TypeInfo& uneInfo);
    // Constructeur avec une information et des sous-arbres gauche et droit
    NoeudBinaire(const TypeInfo& uneInfo,
                 NoeudBinaire<TypeInfo>* ptrFilsGauche,
                 NoeudBinaire<TypeInfo>* ptrFilsDroit);

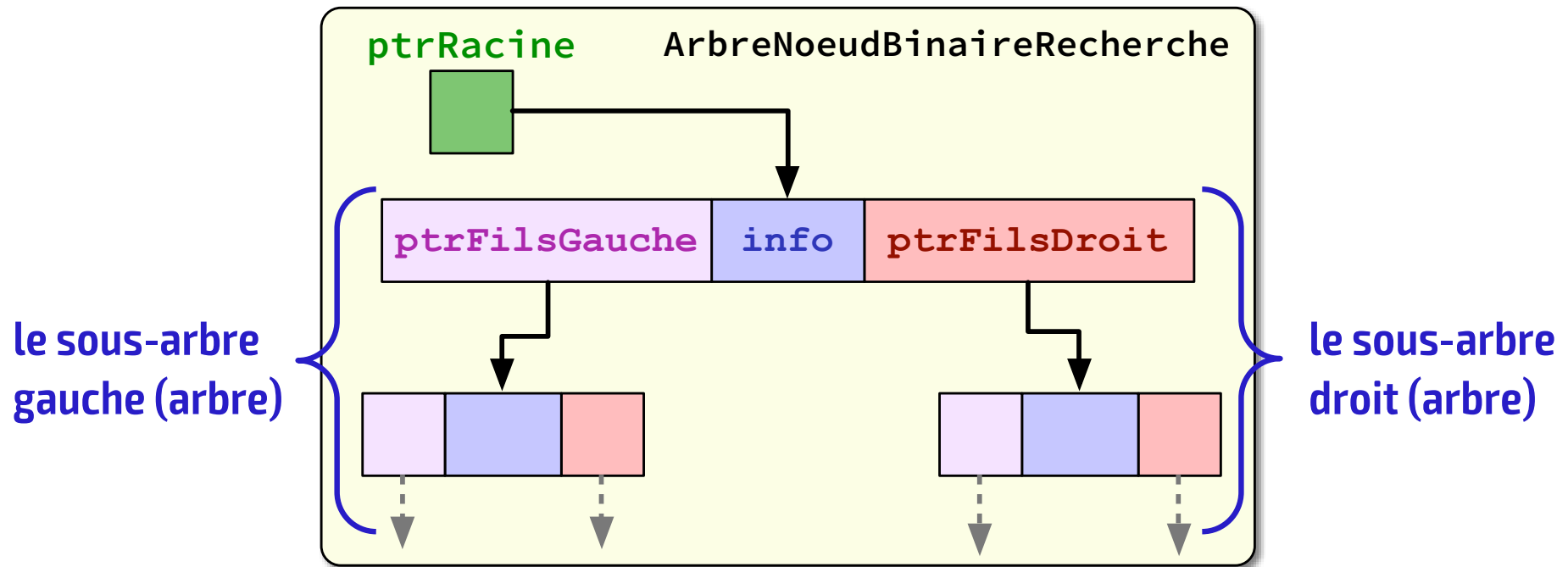
    // Setter mise à jour de l'information de ce noeud
    void setInfo(const TypeInfo& uneInfo);
    // Setters un nouveau sous-arbre gauche ou droit
    void setPtrFilsGauche(NoeudBinaire<TypeInfo>* leftPtr);
    void setPtrFilsDroit(NoeudBinaire<TypeInfo>* rightPtr);

    // Prédicat suis-je une feuille
    bool estFeuille() const;

    // Getter consultation de l'information de ce noeud
    TypeInfo getInfo() const;
    // Getters accès à mon sous-arbre gauche ou droit
    NoeudBinaire<TypeInfo>* getPtrFilsGauche() const;
    NoeudBinaire<TypeInfo>* getPtrFilsDroit() const;
    // Getters accès à une référence sur sous-arbre gauche ou droit (-> workers récursifs)
    NoeudBinaire<TypeInfo>* & getRefPtrFilsGauche();
    NoeudBinaire<TypeInfo>* & getRefPtrFilsDroit();
}; // end NoeudBinaire
```

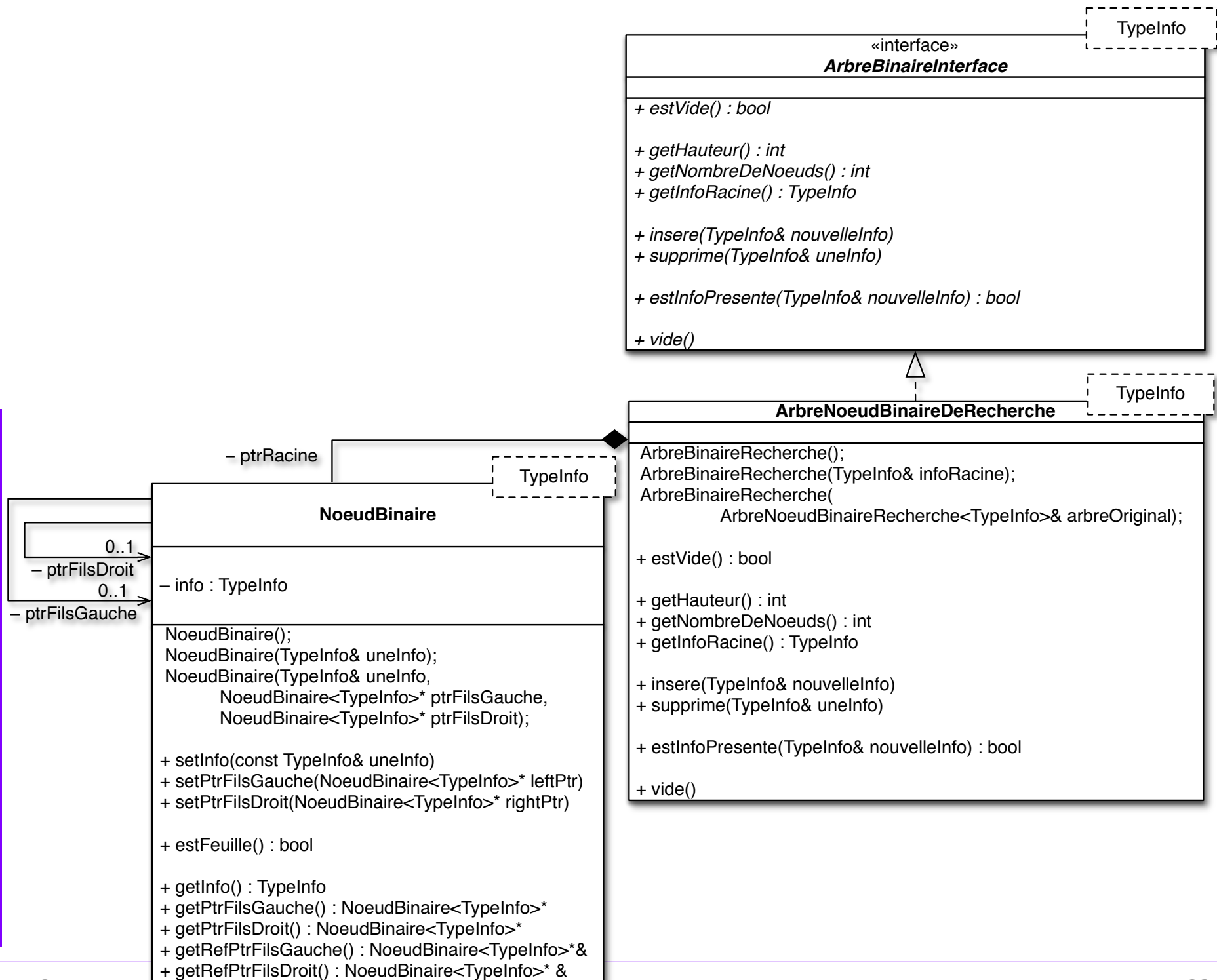
Classe ArbreNoeudBinaireRecherche

🧩 TAD implanté avec des NoeudBinaires chaînés



- 🧩 **ptrRacine** est un pointeur sur le **Noeud** racine
- 🧩 chaque **Noeud** pointe sur ses deux fils
- 🧩 un fils est éventuellement vide (**nullptr**)

La classe ArbreNoeudBinaireRecherche



```

template<class TypeInfo>
class ArbreNoeudBinaireRecherche: public ArbreInterface<TypeInfo> {
private:
    //structure de données pour implanter effectivement la liste
    Noeud<TypeInfo>* ptrRacine; // pointeur sur le Noeud racine de cet ABR

    //méthodes utilitaires privées utilisées par les méthodes publiques
    ...

public:
    ArbreNoeudBinaireRecherche();
    ArbreNoeudBinaireRecherche(const TypeInfo& infoRacine);
    ArbreNoeudBinaireRecherche(const ArbreBinaireRecherche<TypeInfo>& arbreOriginal);

    //-----
    // Méthodes publiques : définition des méthodes virtuelles de ArbreInterface
    //-----
    bool estVide() const;

    int getHauteur() const;
    int getNombreDeNoeuds() const;
    TypeInfo getInfoRacine() const throw (PrecondViolatedExcep);

    bool insere(const TypeInfo& nouvelleInfo);
    bool supprime(const TypeInfo& uneInfo);

    void vide();

    bool estInfoPresente(const TypeInfo& uneInfo) const;
};

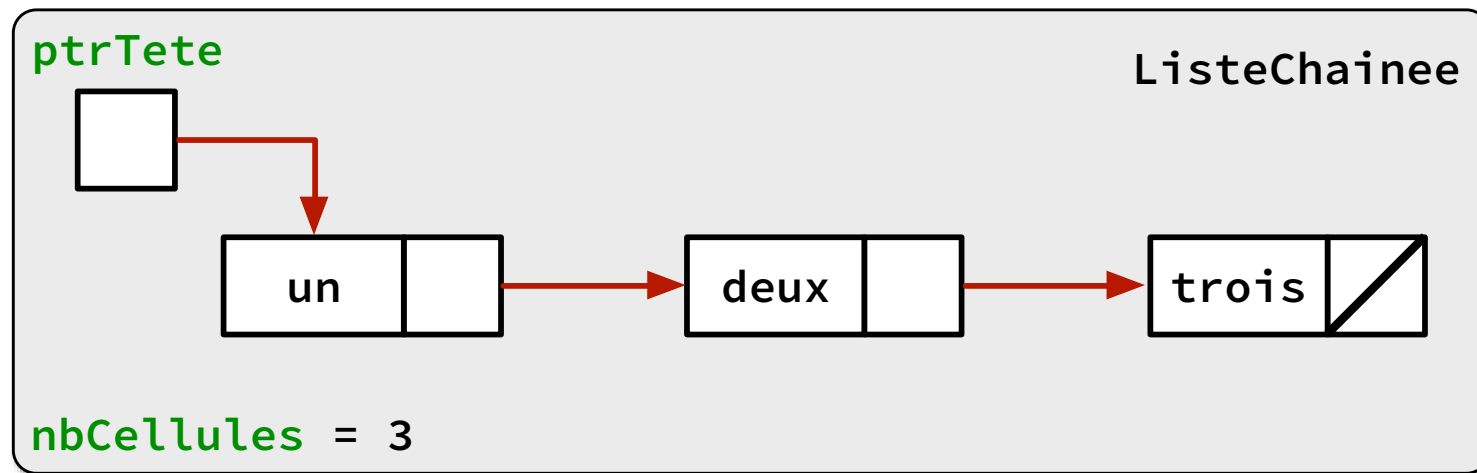
```


Où je me promène dans un arbre !

PARCOURS EN PROFONDEUR

Affichage d'un arbre binaire

- On ne parle pas spécifiquement d'un ABR
 - un ABR n'est qu'un arbre binaire particulier !
- Sur la **ListeChaine**, on a vu deux parcours

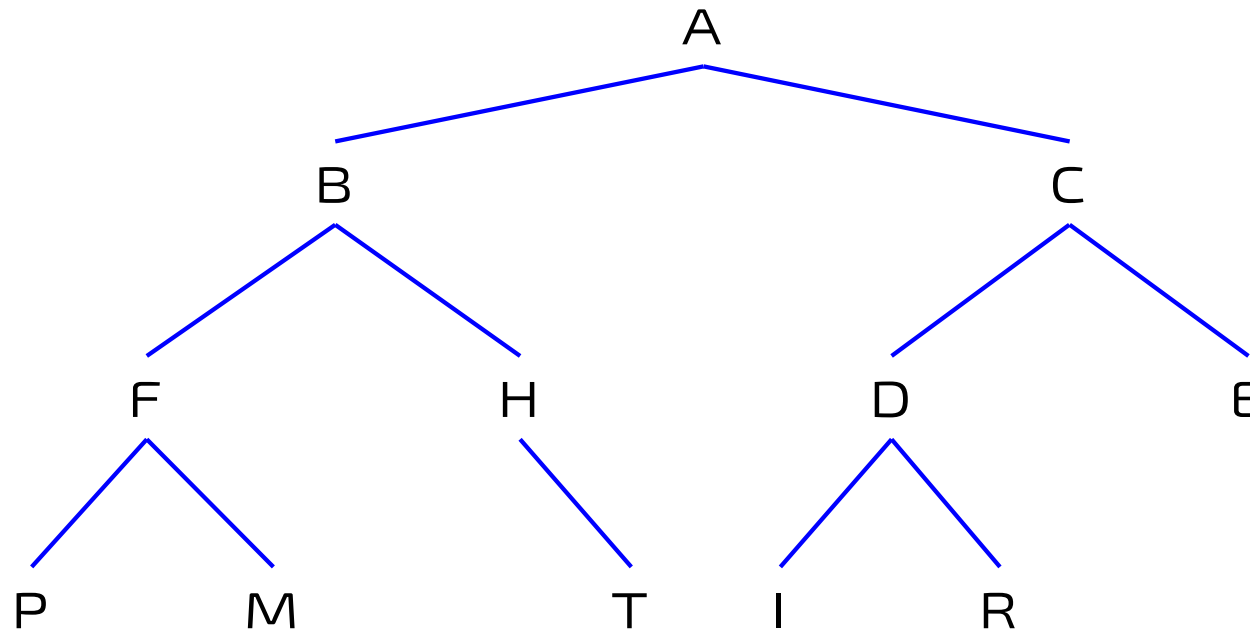


de gauche à droite	de droite à gauche
un deux trois	trois deux un

- Pas le choix, une **Cellule** n'a qu'un successeur !

Affichage d'un arbre binaire

Que peut-on faire sur un arbre binaire ?



On veut seulement afficher toutes les informations
pas faire un joli dessin !

... il faut donc examiner chaque **NoeudBinaire**

une information

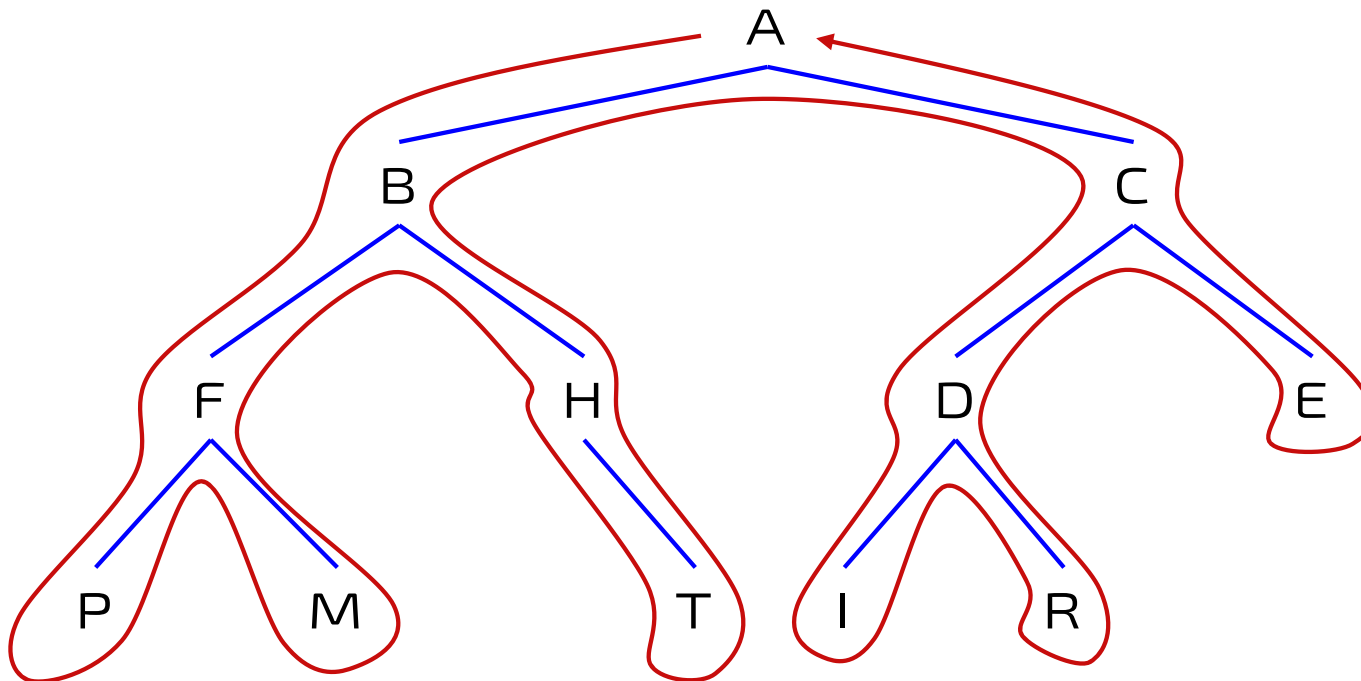
deux fils : gauche et droit (**2 descendants**)

Affichage d'un arbre binaire

Examiner chaque **NoeudBinaire**

l'information

les fils gauche et droit (2 successeurs)

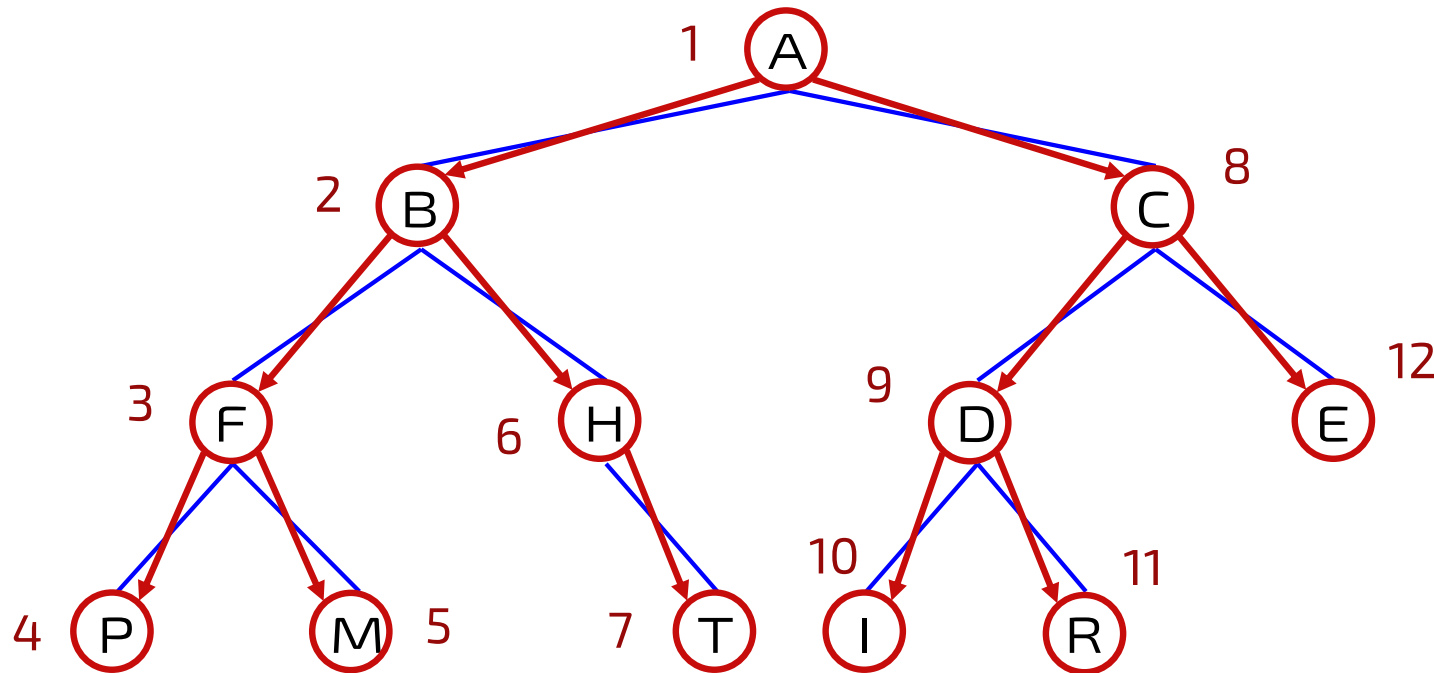


... donc choisir dans quel ordre traiter

l'information et les 2 fils

Affichage d'un arbre binaire

- Avec un parcours préfixé (en préordre)
 - afficher l'information portée par la racine
 - afficher l'arbre de racine le fils gauche
 - afficher l'arbre de racine le fils droit

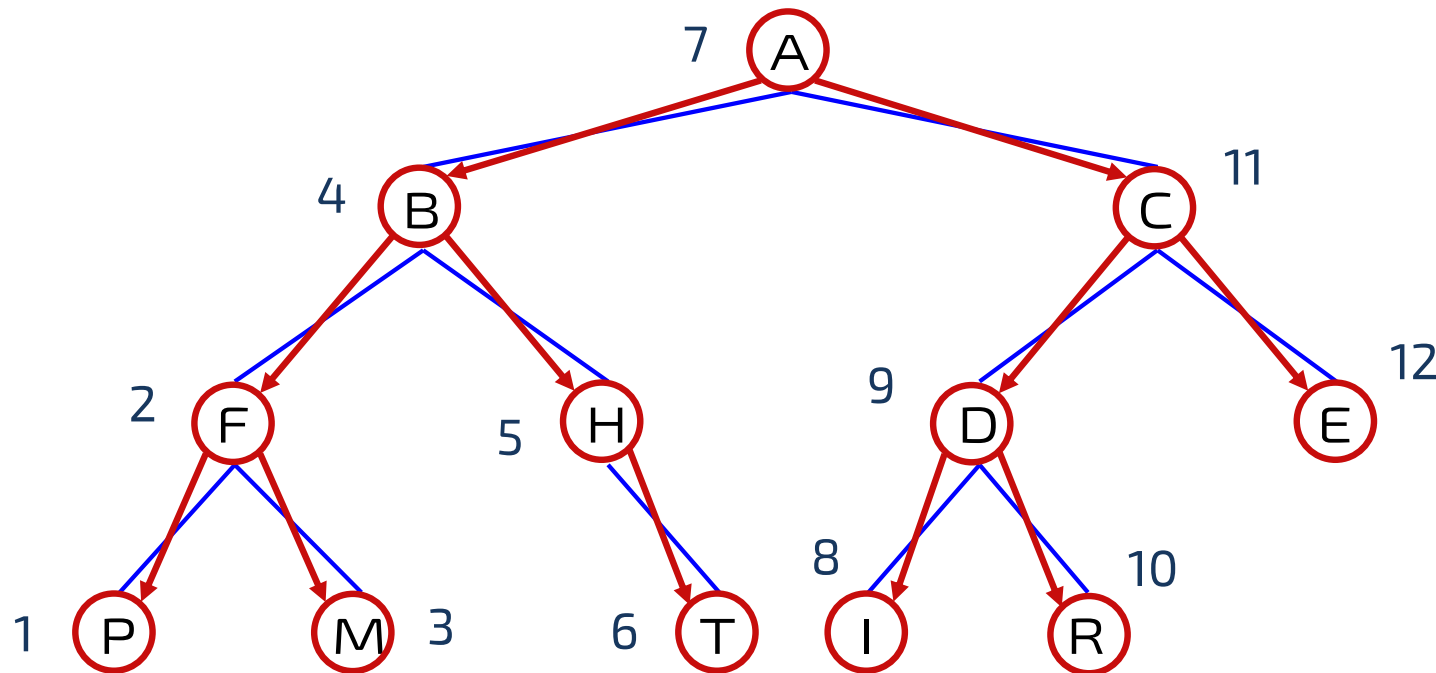


■ Exemple : A B F P M H T C D I R E

Affichage d'un arbre binaire

■ Avec un parcours infixé (projectif, symétrique)

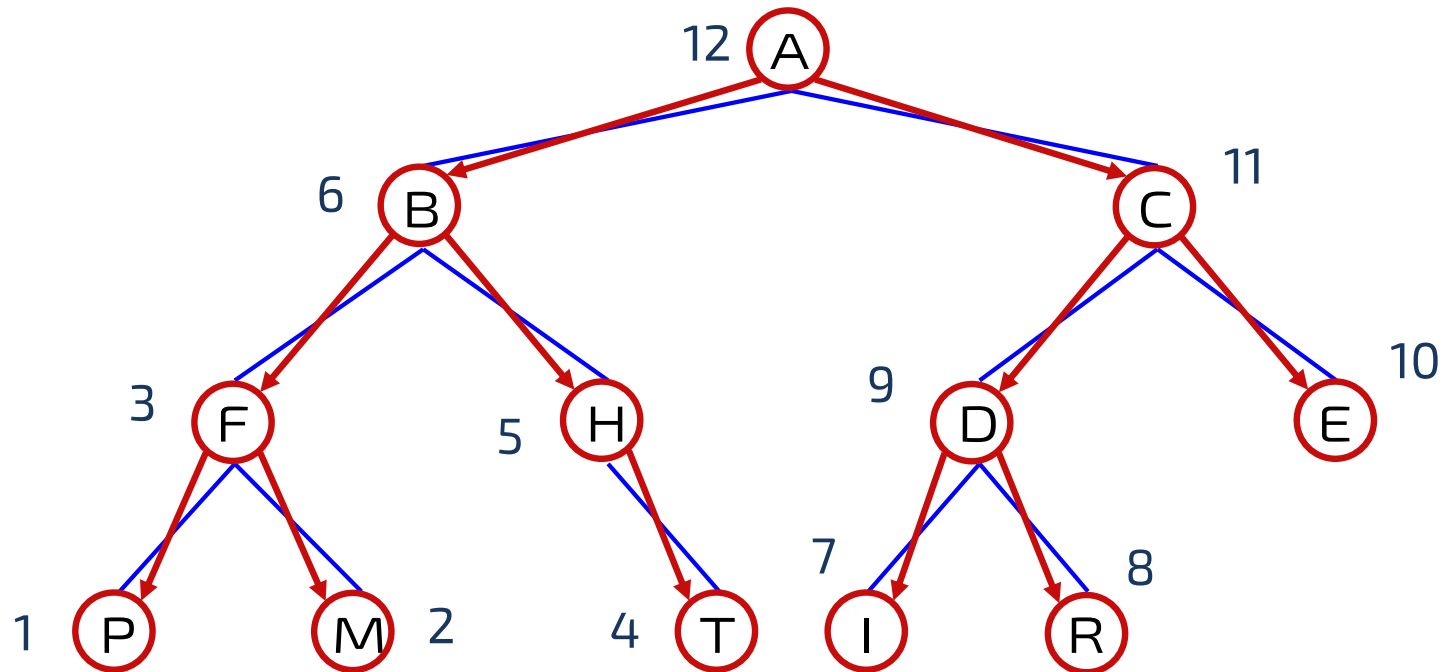
- afficher l'arbre de racine le fils gauche
- afficher l'information portée par la racine
- afficher l'arbre de racine le fils droit



■ Exemple : P F M B H T A I D R C E

Affichage d'un arbre binaire

- Avec un parcours postfixé (terminal)
 - afficher l'arbre de racine le fils gauche
 - afficher l'arbre de racine le fils droit
 - afficher l'information portée par la racine



■ Exemple : P M F T H B I R D E C A

Affichage d'un arbre binaire

 On va avoir 3 méthodes publiques d'affichage d'un arbre binaire et donc d'un

ArbreNoeudBinaireRecherche

 **affichePrefixe()**

 **afficheInfixe()**

 **affichePostfixe()**

 Comme pour la **ListeChaine**, ces 3 méthodes vont faire appel à des procédures **workers récurives** qui réaliserons l'affichage de la descendance en **NoeudBinaires** de la racine

Affichage de NoeudBinaires

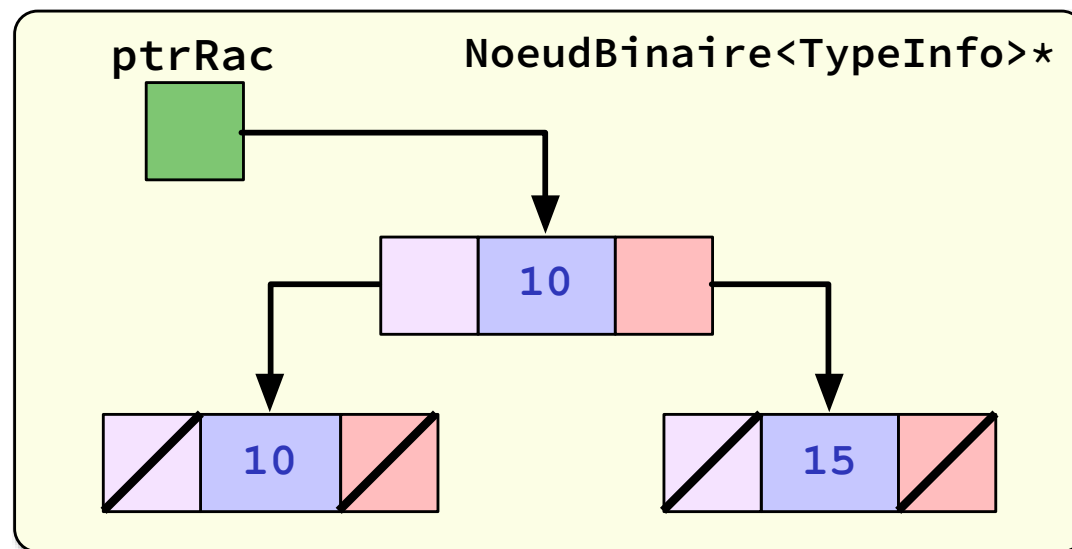
- Prototypes des Worker privés qui affichent une descendance de NoeudsBinaire :

```
void affichePrefixeWorker(const NoeudBinaire<TypeInfo>* ptrRac) const;
```

```
void afficheInfixeWorker(const NoeudBinaire<TypeInfo>* ptrRac) const;
```








```
void affichePostfixeWorker(const NoeudBinaire<TypeInfo>* ptrRac) const;
```

- Le paramètre effectif des procédures sera par exemple



Affichage descendance de NoeudBinaire

Préfixé

-  Réfléchissons à l'écriture de la méthode `affichePrefixeWorker(NoeudBinaire<TypeInfo>* ptrRacine)` en supposant qu'elle existe
-  l'arbre de Noeuds est vide
 -  `ptrRac == nullptr` → il n'y a rien à faire *
-  l'arbre de Noeuds n'est pas vide
 -  `ptrRac != nullptr` →
 1. afficher l'info de la racine : `ptrRac->getInfo()`
 2. afficher en parcours préfixé l'arbre de racine le fils gauche `ptrRac->getFilsGauche()`
 -  en utilisant la méthode `affichePrefixeWorker()` faite pour ça !!
 3. afficher en parcours préfixé l'arbre de racine le fils droit `ptrRac->getFilsDroit()`
 -  en utilisant la méthode `affichePrefixeWorker()` faite pour ça !!

Affichage descendance de NoeudBinaire

Préfixé

En résumé

- `ptrRac == nullptr` ➡ c'est terminé *
- `ptrRac != nullptr` ➡

```
cout << ptrRac->getInfo() << " ";
affichePrefixeWorker(ptrRac->getPtrFilsGauche());
affichePrefixeWorker(ptrRac->getPtrFilsDroit());
```

Procédure

```
template<class TypeInfo>
void ArbreNoeudBinaireRecherche<TypeInfo>::
    affichePrefixeWorker(const NoeudBinaire<TypeInfo>* ptrRac) const {
    if (ptrRac) {
        cout << ptrRac->getInfo() << " ";
        affichePrefixeWorker(ptrRac->getPtrFilsGauche());
        affichePrefixeWorker(ptrRac->getPtrFilsDroit());
    } // sinon rien à faire, donc on ne fait rien !!!
}
```

Affichage descendance de NoeudBinaire

Infixé

 On raisonne de la même manière

- `ptrRac == nullptr` ➡ c'est terminé *
- `ptrRac != nullptr` ➡

```
    afficheInfixeWorker(ptrRac->getPtrFilsGauche());  
    cout << ptrRac->getInfo() << " ";  
    afficheInfixeWorker(ptrRac->getPtrFilsDroit());
```

 Procédure

```
template<class TypeInfo>  
void ArbreNoeudBinaireRecherche<TypeInfo>::  
afficheInfixeWorker(const NoeudBinaire<TypeInfo>* ptrRac) const {  
    if (ptrRac) {  
        afficheInfixeWorker(ptrRac->getPtrFilsGauche());  
        cout << ptrRac->getInfo() << " ";  
        afficheInfixeWorker(ptrRac->getPtrFilsDroit());  
    } // sinon rien à faire, donc on ne fait rien !!!  
}
```

Affichage descendance de NoeudBinaire

Postfixé

 On raisonne de la même manière

- `ptrRac == nullptr` ➡ c'est terminé *
- `ptrRac != nullptr` ➡

```
    affichePostfixeWorker(ptrRac->getPtrFilsGauche());  
    affichePostfixeWorker(ptrRac->getPtrFilsDroit());  
    cout << ptrRac->getInfo() << " ";
```

 Procédure

```
template<class TypeInfo>  
void ArbreNoeudBinaireRecherche<TypeInfo>::  
affichePostfixeWorker(const NoeudBinaire<TypeInfo>* ptrRac) const {  
    if (ptrRac) {  
        affichePostfixeWorker(ptrRac->getPtrFilsGauche());  
        affichePostfixeWorker(ptrRac->getPtrFilsDroit());  
        cout << ptrRac->getInfo() << " ";  
    }    // sinon rien à faire, donc on ne fait rien !!!  
}
```

Affichage d'un ArbreNoeudBinaireRecherche

 3 nouvelles méthodes publiques

+ .h

```
// Affichages récurifs de cet arbre binaire ordonné
// parcours préfixé, infixé et suffixé
void affichePrefixe() const;
void afficheInfixe() const;
void affichePostfixe() const;
```

+ .cpp

```
template<class TypeInfo>
void ArbreNoeudBinaireRecherche<TypeInfo>::affichePrefixe() const
{
    cout << "En parcours préfixé, l'ABO contient -> ";
    affichePrefixeWorker(ptrRacine);
    cout << endl;
}

// même modèle pour afficheInfixe() et affichePostfixe()
```

Remarque



Ces parcours sont dits en profondeur ...



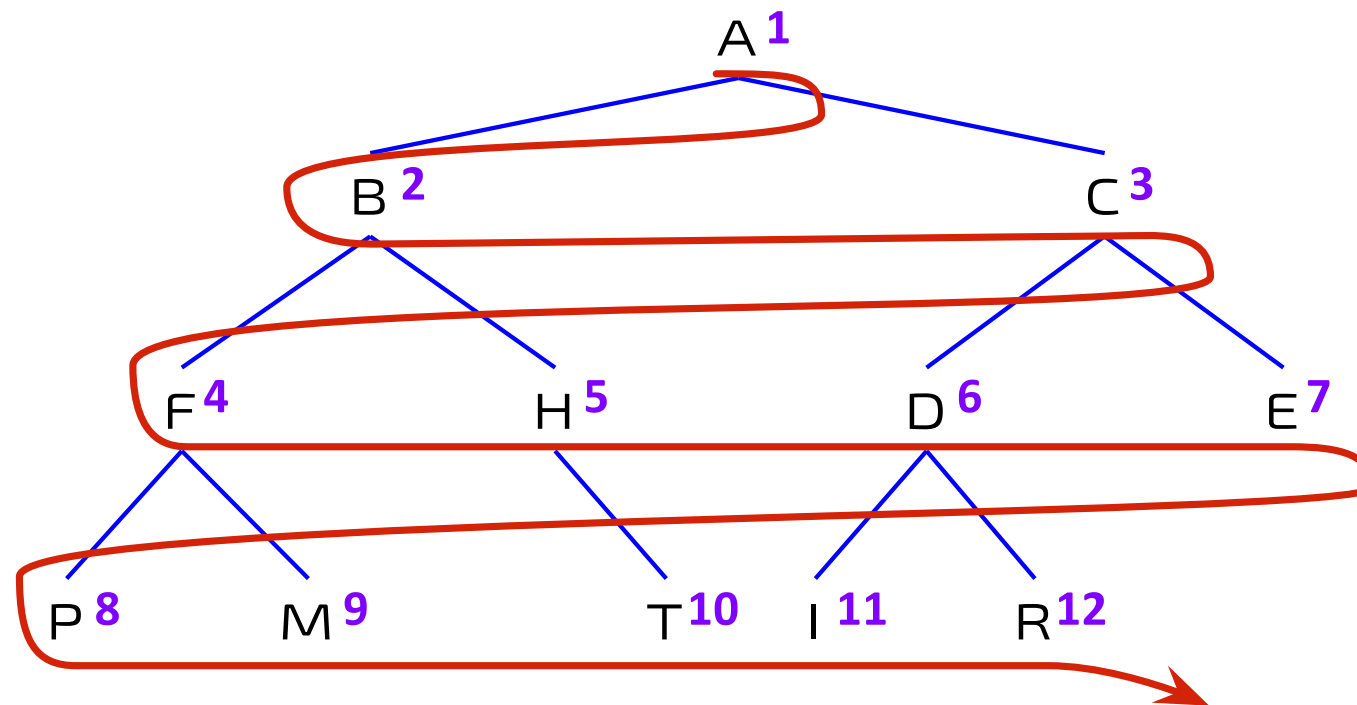
on descend sur le nœud le plus en bas à gauche avant de remonter



Il existe aussi un parcours en largeur ...



qui consiste à parcourir l'arbre niveau par niveau




A B C F H D E P M T I R

Où je comprends l'intérêt des ABO pour la recherche !

RECHERCHE DANS UN ABO

Recherche dans un ABO : présence

 Soit à écrire une **méthode publique** (prédicat) de la classe **ArbreNoeudBinaireRecherche**

 **bool** **estInfoPresente**(**const** TypeInfo& uneInfo)
const;

 Ce prédicat ne peut être récursif car il s'applique sur un objet de type **NoeudBinaire**

 ce que l'on veut parcourir la descendance en **NoeudBinaires** via **ptrRacine**

 Nous allons faire comme pour l'affichage en utilisant un worker

Recherche dans un ABO : présence

 La méthode publique `estInfoPresente()` utilisera une **fonction privée réursive** (un worker fonction)

```
bool estInfoPresenteWorker(  
    const NoeudBinaire<TypeInfo>* ptrRac,  
    const TypeInfo& infoCible) const;
```

 qui retourne

 **true** si l'information `infoCible` est présente dans la descendance en `NoeudBinaires` de `ptrRac`

 **false** sinon

```
template<class TypeInfo>  
bool ArbreNoeudBinairerecherche<TypeInfo>::estInfoPresente (  
    const TypeInfo& uneInfo) const {  
    return estInfoPresenteWorker(ptrRacine, uneInfo);  
}
```

Recherche dans un ABO : présence

```
bool estInfoPresenteWorker(  
    const NoeudBinaire<TypeInfo>* ptrArbre,  
    const TypeInfo& infoCible) const;
```

Schéma récursif

```
> ptrArbre == nullptr → return false; *
```

```
> ptrArbre != nullptr →  
    >> ptrArbre->getInfo() == infoCible → return true; *  
    >> ptrArbre->getInfo() > infoCible →  
        // le résultat est celui de la recherche à gauche  
        return estInfoPresenteWorker(ptrRacine->getPtrFilsGauche(),  
                                       infoCible);  
    >> ptrArbre->getInfo() < infoCible →  
        // le résultat est celui de la recherche à droite  
        return estInfoPresenteWorker(ptrRacine->getPtrFilsDroit(),  
                                       infoCible);
```

Recherche dans un ABO : présence



Nouvelle méthode privée dans `ArbreNBRecherche`

.h

```
private:  
    bool estInfoPresenteWorker(...) const;
```

.cpp

```
template<class TypeInfo>  
bool ArbreBinaireRecherche<TypeInfo>::estInfoPresenteWorker(  
    const NoeudBinaire<TypeInfo>* ptrRacine,  
    const TypeInfo& infoCible) const {  
    if (ptrRacine == nullptr)  
        return false; // non trouvé  
    else if (ptrRacine->getInfo() == infoCible)  
        return true; // trouvé  
    else if (ptrRacine->getInfo() > infoCible)  
        // chercher dans le sous-arbre gauche  
        return estInfoPresenteWorker(ptrRacine->getPtrFilsGauche(),  
                                     infoCible);  
    else // ptrRacine->getInfo() < infoCible  
        // chercher dans le sous-arbre droit  
        return estInfoPresenteWorker(ptrRacine->getPtrFilsDroit(),  
                                     infoCible);  
}
```