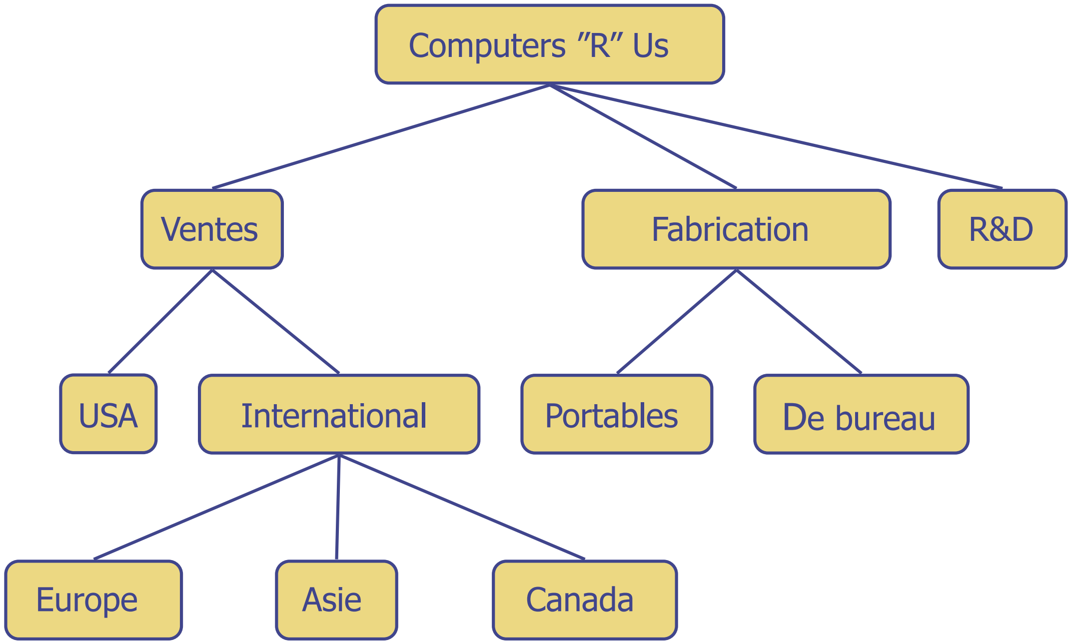
Arbre
Terminologie
ADT *Tree*
Parcours préfixe et postfixe
Parcours en largeur
Arbre binaire
Propriétés de l'arbre binaire
ADT *BinaryTree*
Parcours en ordre
Arbre chaîné
Arbre dans un tableau

Qu'est-ce qu'un arbre ?

- En informatique, un arbre est un modèle abstrait d'une structure hiérarchique
- Un arbre est constitué de noeuds ayant une relation de parent-enfant
- Les applications incluent :
  - les systèmes de fichiers
  - hiérarchies organisationnelles
  - héritage en programmation objet
  - généalogies et phylogénies
  - syntaxe de langages naturels et de programmation
  - expression arithmétiques
  - sections d'un document



Terminologie

<u>Racine</u>: noeud sans parent (A)

<u>Nœud interne</u>: noeud avec au moins un enfant (A, B, C, F)

<u>Nœud externe</u>, ou **<u>feuille</u>**: nœud sans enfants (E, I, J, K, G, H, D)

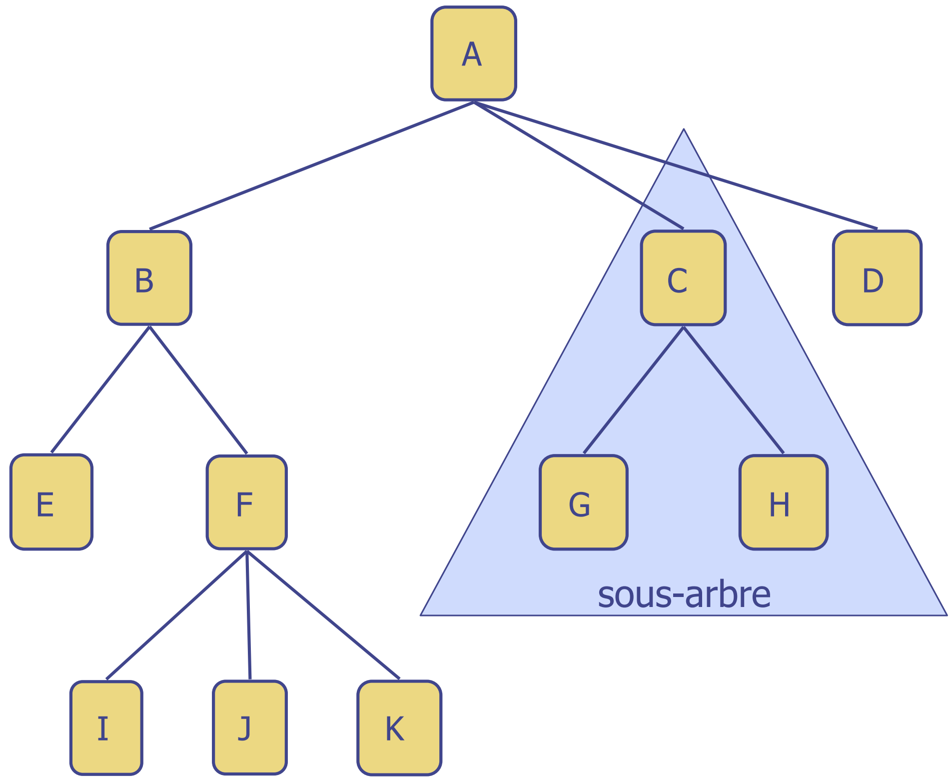
<u>Ancêtres d'un nœud</u>: parent, grand-parent, grand-grand-parent, etc.

<u>Profondeur d'un nœud</u>: nombre d'ancêtres

<u>Hauteur d'un arbre</u>: profondeur maximale d'un de ses nœuds (3)

<u>Descendant d'un nœud</u>: enfant, petit-enfant, arrière-petit-enfant, etc.

<u>Sous-arbre</u>: arbre constitué d'un nœud et de ses descendants

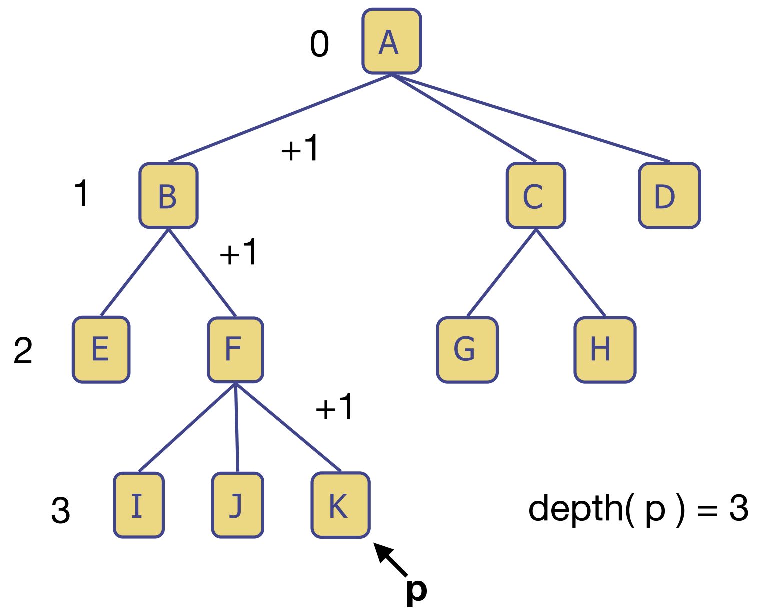


ADT *Tree*

- Nous utilisons des positions pour abstraire des nœuds (comme pour la liste positionnelle)
- Méthodes génériques :
  - *Integer len( )*
  - *Boolean est_vide( )*
  - *Iterator positions( )*
  - *Iterator iter( )*
- Méthodes d'accès :
  - *Position racine( )*
  - *Position parent(p)*
  - *Iterator enfants(p)*
  - *Integer nb_enfants(p)*
- Méthodes requête :
  - *Boolean est_feuille(p)*
  - *Boolean est_racine(p)*
- Méthode de mise à jour :
  - *element remplace(p, o)*
- Des méthodes de mise à jour supplémentaires peuvent être définies par des structures de données implémentant l'ADT Arbre

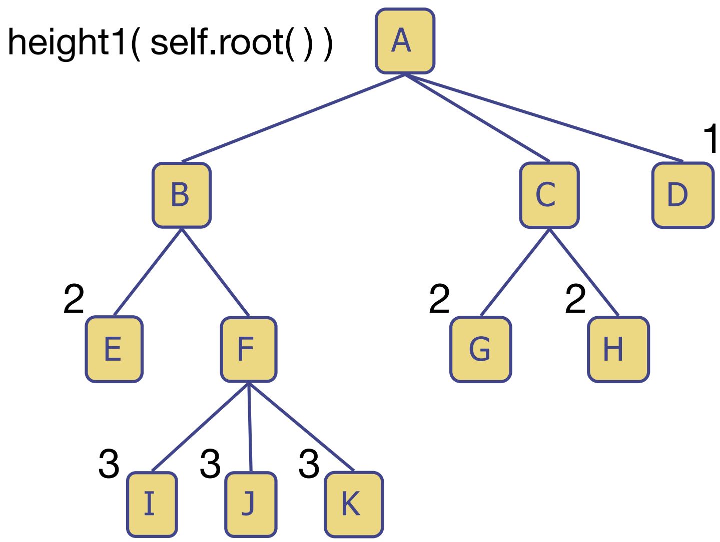
```python
# utilise ListQueue
from ListQueue import ListQueue

#ADT Tree (Classe de base)
class Tree:

    #inner class Position
    class Position:

        def element( self ):
            pass

        def __eq__( self, other ):
            pass

        def __ne__( self, other):
            return not( self == other )
```

```
# retourne la racine
def root( self ):
    pass

# retourne le parent d'une Position
def parent( self, p ):
    pass

# retourne le nombre d'enfants d'une Position
def num_children( self, p ):
    pass

# retourne les enfants d'une Position
def children( self, p ):
    pass

# retourne le nombre de noeuds
def __len__( self ):
    pass

# demande si une Position est la racine
def is_root( self, p ):
    return self.root() == p

# demande si une Position est une feuille
def is_leaf( self, p ):
    return self.num_children( p ) == 0

# demande si un arbre est vide
def is_empty( self ):
    return len( self ) == 0

# retourne la profondeur d'une Position
def depth( self, p ):
    # retourne le nombre d'ancêtres d'une Position
    if self.is_root( p ):
        return 0
    else:
        return 1 + self.depth( self.parent() )
```

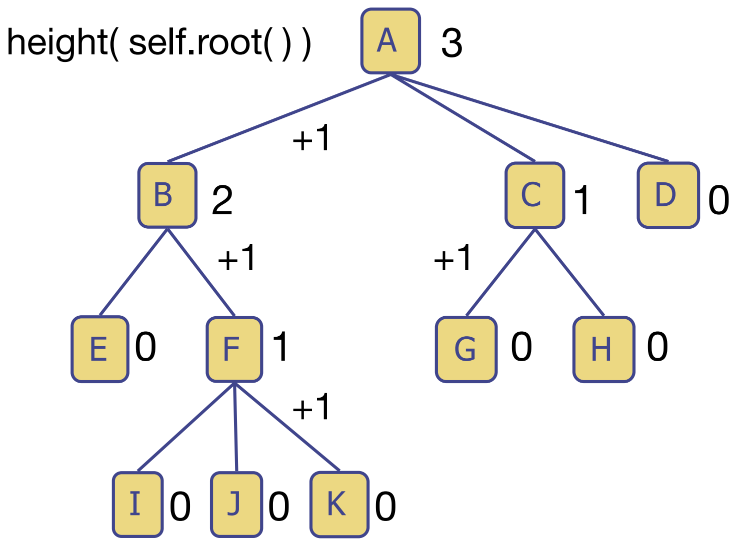


Considérons 2 définitions pour la hauteur

```
# retourne la hauteur d'une Position avec depth (non efficace)
def height1( self, p ):
    # retourne la profondeur maximum des feuilles sous une Position
    # positions n'est pas implanté et se fait en O(n)
    return max( self.depth( p ) for p in self.positions() if self.is_leaf( p ))

# retourne la hauteur d'une Position en descendant l'arbre (efficace)
def height( self, p ):
    # retourne la hauteur d'un sous-arbre à une Position
    if self.is_leaf( p ):
        return 0
    else:
        return 1 + max( self.height( c ) for c in self.children( p ) )
```



nb noeuds visités = 16



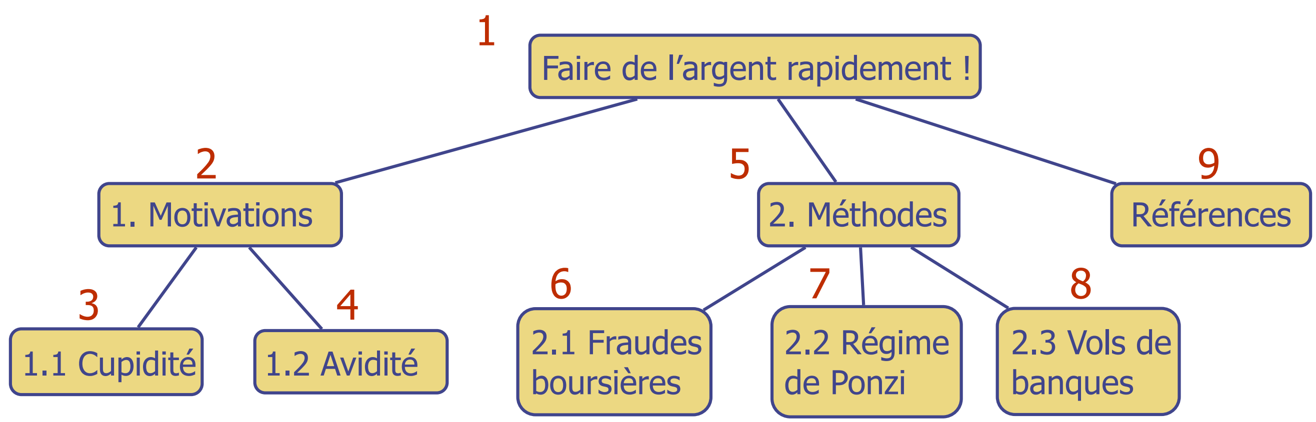
nb noeuds visités = 11

Parcours préfixe

- Une parcours visite les noeuds d'un arbre de manière systématique
- Dans un parcours de préfixe, un noeud est visité avant ses descendants
- Application : imprimer un document

Algorithme *préfixe(v)*
    visite(v)
    pour chaque enfant *w* de *v*
        préfixe(w)

```
# imprime le sous-arbre dont la racine est la Position p
# utilise un parcours préfixé
def preorder_print( self, p, indent = "" ):
    # on traite le noeud courant
    print( indent + str( p ) )
    # et par la suite les enfants, récursivement
    for c in self.children( p ):
        self.preorder_print( c, indent + "    " )
```

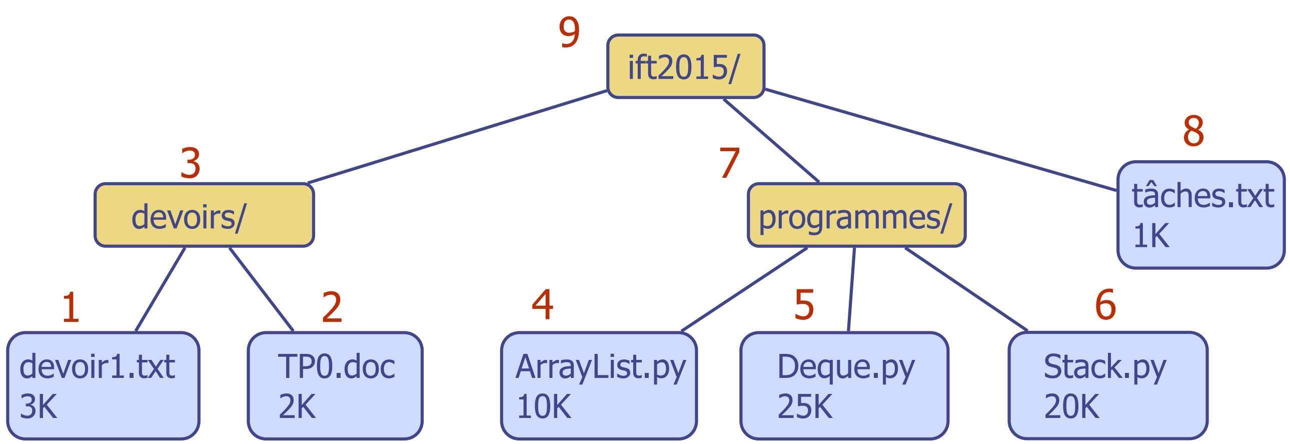


Parcours postfixe

- Dans un parcours postfixe, un noeud est visité après ses descendants
- Application : calcul de l'espace utilisé par des fichiers dans un répertoire et ses sous-répertoires

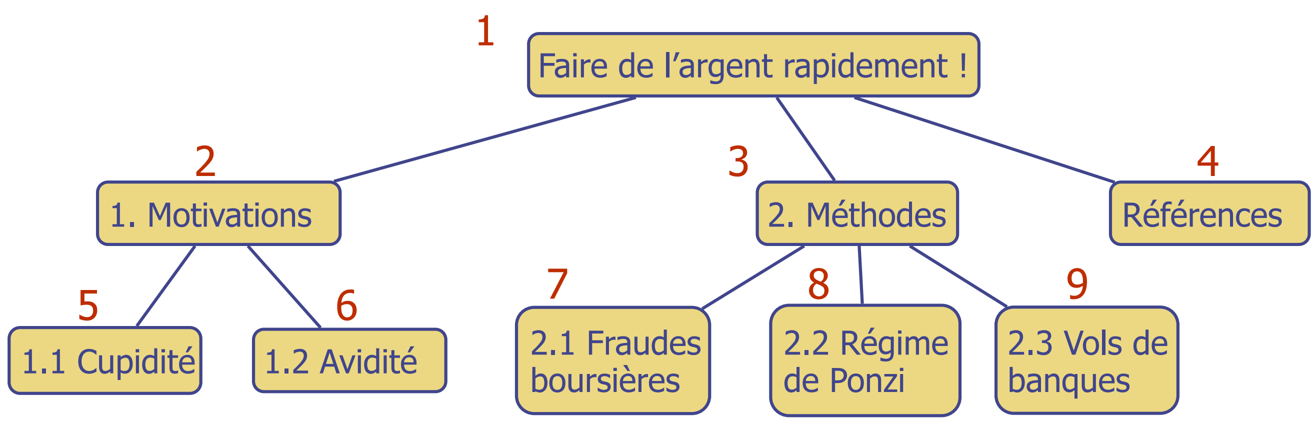
Algorithme *postfixe*(*v*)
    pour chaque enfant *w* de *v*
        postfixe(*w*)
    visite(*v*)

```
# imprime le sous-arbre dont la racine est la Position p
# utilise un parcours postfixé
def postorder_print( self, p ):
    # on traite les enfants
    for c in self.children( p ):
        self.postorder_print( c )
    # et par la suite le parent
    print( p )
```



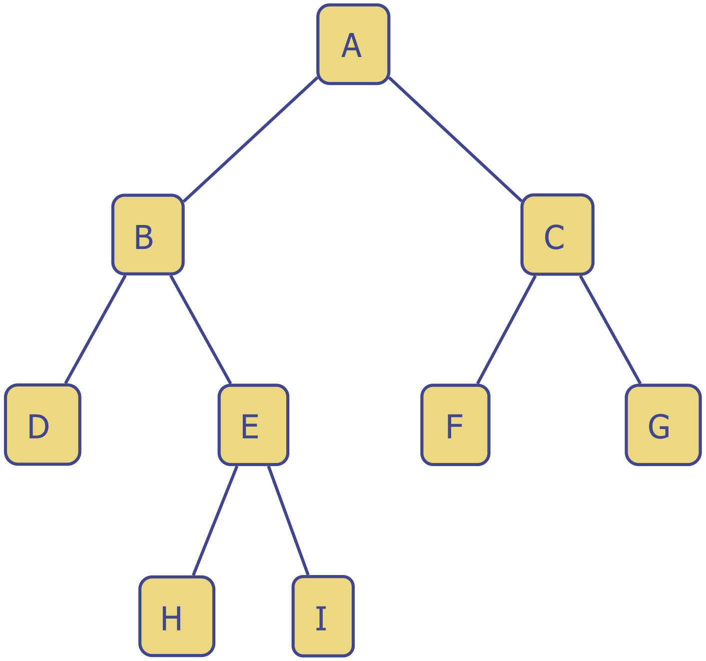
Parcours en largeur

```
# imprime le sous-arbre dont la racine est la Position p
# utilise un parcours en largeur, utilisant une File
def breadth_first_print( self, p ):
    Q = ListQueue()
    # on enqueue la Position p
    Q.enqueue( p )
    # tant qu'il y a des noeuds dans la File
    while not Q.is_empty():
        # prendre le suivant et le traiter
        q = Q.dequeue()
        print( q )
        # enqueuer les enfants du noeud traité
        for c in self.children( q ):
            Q.enqueue( c )
```



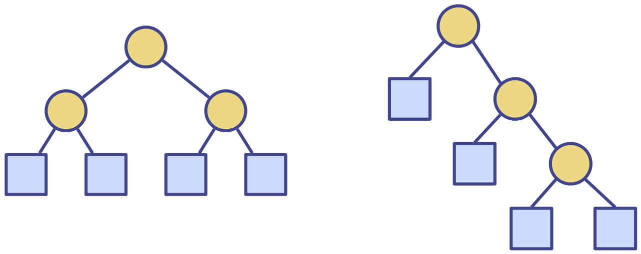
Arbre binaire

- Un **arbre binaire** est un arbre avec les propriétés suivantes :
  - Chaque nœud interne a au plus deux enfants (exactement deux pour un arbre binaire plein)
  - Les enfants d'un nœud sont une paire ordonnée
- Nous appelons les enfants d'un nœud interne l'enfant de gauche et l'enfant de droite
- Une définition récursive alternative est : un **arbre binaire** est soit
  - un arbre constitué d'un seul nœud, ou
  - un arbre dont la racine a une paire ordonnée d'enfants, dont chacun est un arbre binaire
- Applications:
  - expressions arithmétiques
  - processus de décision
  - recherche

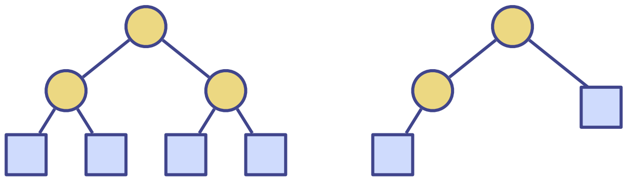


Arbres binaires plein et complet

Un **<u>arbre binaire plein</u>** ("full" en anglais) est un arbre binaire dont tous les noeuds internes possèdent 2 enfants. Un arbre binaire plein est aussi dit **<u>propre</u>** ou **<u>stricte</u>**.



Un **<u>arbre binaire complet</u>** ("complete" en anglais) est un arbre binaire dont tous les niveaux sauf possiblement le dernier sont complètement remplis et tous les noeuds sont le plus à gauche possible. *Un arbre binaire complet n'est pas nécessairement plein* !

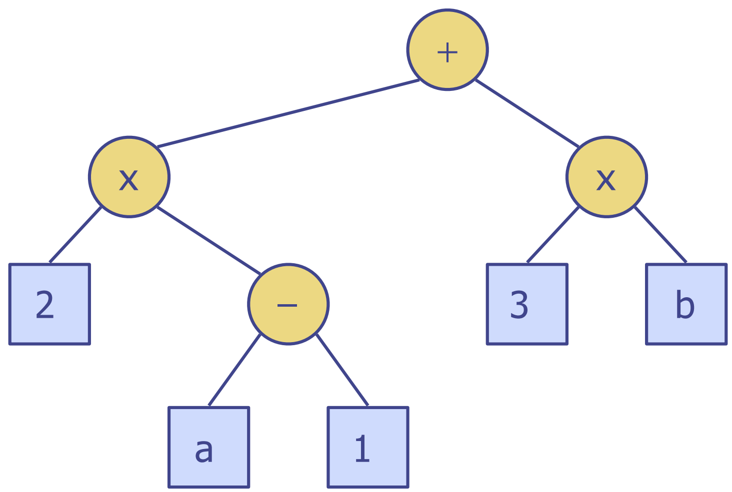


Arbre binaire pour une expression arithmétique

Les nœuds internes sont les opérateurs

Les noeuds externes sont les opérandes

Exemple : (2 x (a - 1) + (3 x b))

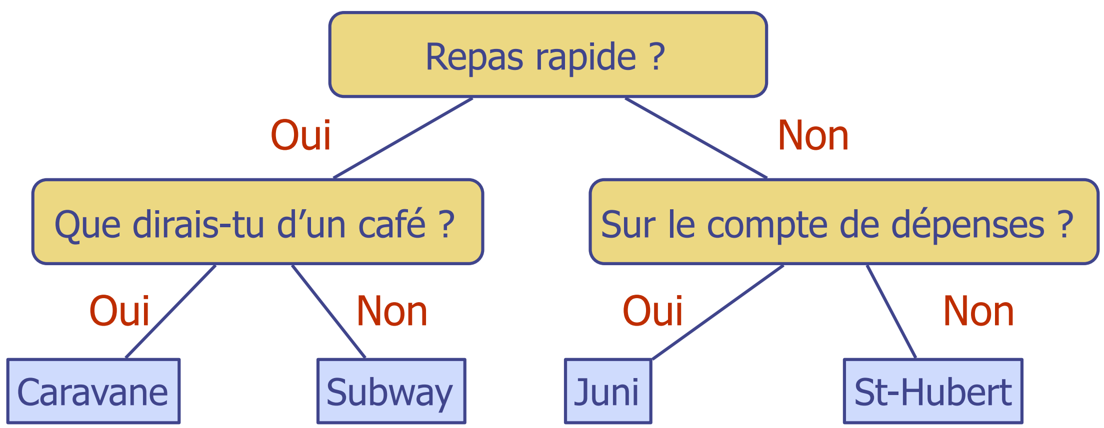


Arbre de décision

Les nœuds internes sont des questions à réponses oui/non

Les nœuds externes sont des décisions

Exemple : décider quoi manger



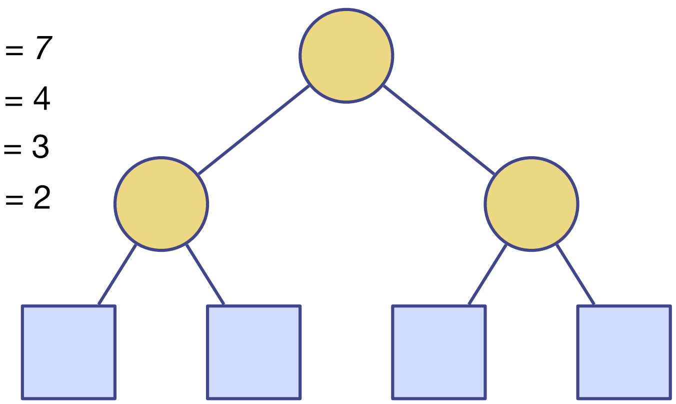
Propriétés d'un arbre binaire plein

(tous les noeuds internes possèdent 2 enfants)

- Notation :
  - n nombre de noeuds
  - e nombre de noeuds externes
  - i nombre de noeuds internes
  - h hauteur h

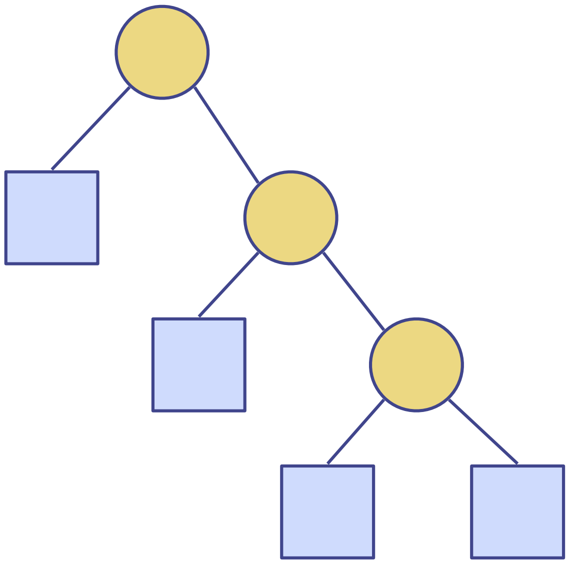
- Propriétés :
  - $\boldsymbol{e} = \boldsymbol{i} + 1$
  - $\boldsymbol{n} = 2\boldsymbol{e} - 1$
  - $\boldsymbol{h} \leq \boldsymbol{i}$
  - $\boldsymbol{h} \leq (\boldsymbol{n} - 1)/2$
  - $\boldsymbol{e} \leq 2^{\boldsymbol{h}}$
  - $\boldsymbol{h} \geq \log_2 \boldsymbol{e}$
  - $\boldsymbol{h} \geq \log_2 (\boldsymbol{n} + 1) - 1$

$n = 7$
$e = 4$
$i = 3$
$h = 2$



$\boldsymbol{e} \leq 2^{h} \leq 4$

$n = 7$
$e = 4$
$i = 3$
$h = 3$



$\boldsymbol{e} = \boldsymbol{i} + 1 = 3 + 1 = 4$
$\boldsymbol{n} = 2\boldsymbol{e} - 1 = 8 - 1 = 7$
$\boldsymbol{h} \leq \boldsymbol{i} \leq 3$
$\boldsymbol{h} \leq (\boldsymbol{n} - 1)/2 \leq 6/2 \leq 3$
$\boldsymbol{e} \leq 2^{h} \leq 8$
$\boldsymbol{h} \geq \log_2 \boldsymbol{e} \geq 2$
$\boldsymbol{h} \geq \log_2 (\boldsymbol{n} + 1) - 1 \geq 3 - 1 \geq 2$

ADT *BinaryTree*

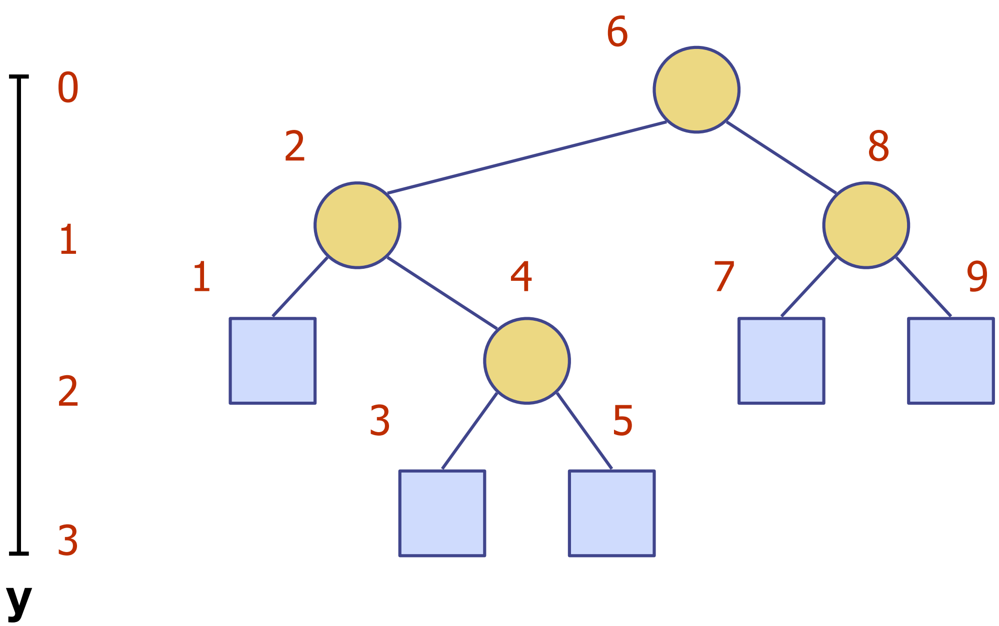
- L'ADT BinaryTree étend l'ADT Tree, c'est-à-dire qu'il hérite de toutes ses méthodes
- Méthodes supplémentaires :
  - position gauche(p)
  - position droite(p)
  - position adelphe(p)
- D'autres méthodes supplémentaires peuvent être définies pour des structures de données spécifiques qui implantent l'ADT BinaryTree

```python
# utilise Tree (Tree.py)
from Tree import Tree

# Classe de base pour arbres binaires
class BinaryTree( Tree ):

    # retourne l'enfant de gauche d'une Position
    def left( self, p ):
        pass

    # retourne l'enfant de gauche d'une Position
    def right( self, p ):
        pass

    # retourne l'adelphe d'une Position
    def sibling( self, p ):
        # on passe par le parent
        parent = self.parent( p )
        # si le parent n'existe pas, p est la racine, pas d'adelphe
        if parent is None:
            return None
        # sinon, si p est l'enfant gauche, on retourne l'enfant droit
        #        si p est l'enfant droit, on retourne l'enfant gauche
        else:
            if p == self.left( parent ):
                return self.right( parent )
            else:
                return self.left( parent )

    # retourne un générateur des enfants dans l'ordre gauche-droit
    def children( self, p ):
        if self.left( p ) is not None:
            yield self.left( p )
        if self.right( p ) is not None:
            yield self.right( p )

    #print the subtree rooted by position p
    #using an inorder traversal
    def inorder_print( self, p ):
        if self.left( p ) is not None:
            self.inorder_print( self.left( p ) )
        print( p )
        if self.right( p ) is not None:
            self.inorder_print( self.right( p ) )
```

Parcours dans l'ordre

```python
# traverse le sous-arbre dont la racine est une Position
# utilise un parcours dans l'ordre gauche-racine-droit
def inorder_print( self, p ):
    # si l'enfant gauche existe, on le traite, récursivement
    if self.left( p ) is not None:
        self.inorder_print( self.left( p ) )
    # on traite p
    print( p )
    # si l'enfant droit existe, on le traite, récursivement
    if self.right( p ) is not None:
        self.inorder_print( self.right( p ) )
```

Algorithme *dansOrdre(v)*
    si *v* **possède un enfant gauche**
        dansOrdre(gauche(v))
    visite(v)
    si *v* **possède un enfant droit**
        dansOrdre(droit(v))



Dans un parcours dans l'ordre, un noeud est visité après son sous-arbre gauche et avant son sous-arbre droit

Application : dessiner un arbre binaire

x(v) = rang de v

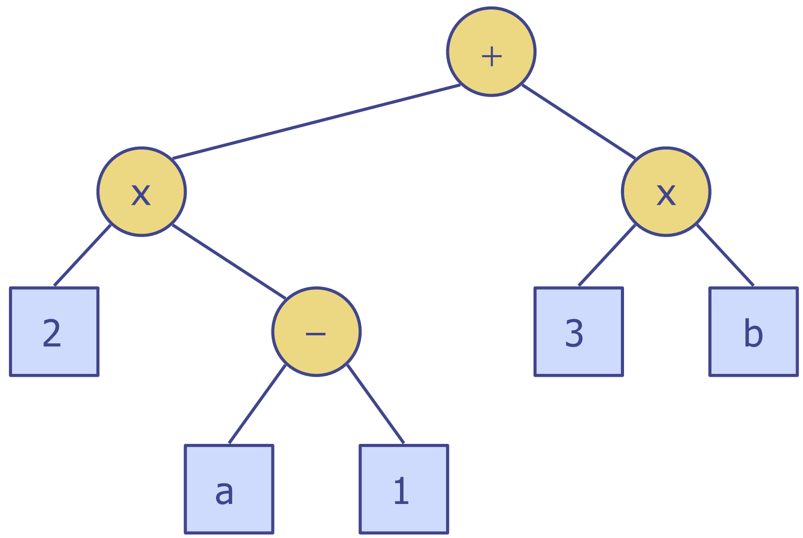
y(v) = profondeur de v

Imprimer des expressions arithmétiques

Spécialisation d'un parcours *inorder*

- imprimer l'expression de gauche entre parenthèses
- imprimer l'opération
- imprimer l'expression de droite entre parenthèses

Algorithme *imprimeExpression(v)*
    si *v* **possède un enfant gauche**
        imprimer("(")
        imprimeExpression(gauche(v))
    imprimer(v.element())
    si *v* **possède un enfant droit**
        imprimeExpression(droit(v))
        imprimer(")")



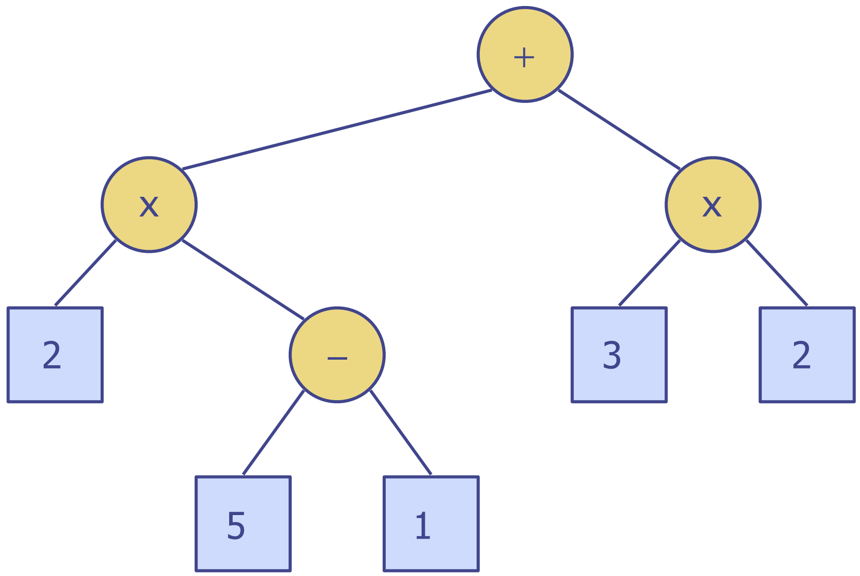
((2x(a – 1))+(3xb))

```python
def printExpression( self, p ):
    # imprime expression gauche
    if self.left( p ) is not None:
        print( '(' )
        self.printExpression( self.left( p ) )
    # imprime opération
    print( p )
    # imprime expression droite
    if self.right( p ) is not None:
        self.printExpression( self.right( p ) )
        print( ')' )
```

Évaluer les expressions arithmétiques

Spécialisation d'un parcours *postorder*

- méthode récursive renvoyant la valeur d'un sous-arbre
- lors de la visite d'un noeud interne, combinez les valeurs des sous-arbres

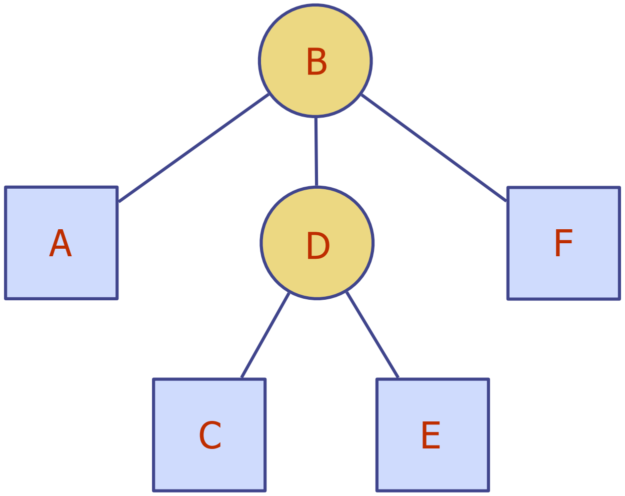


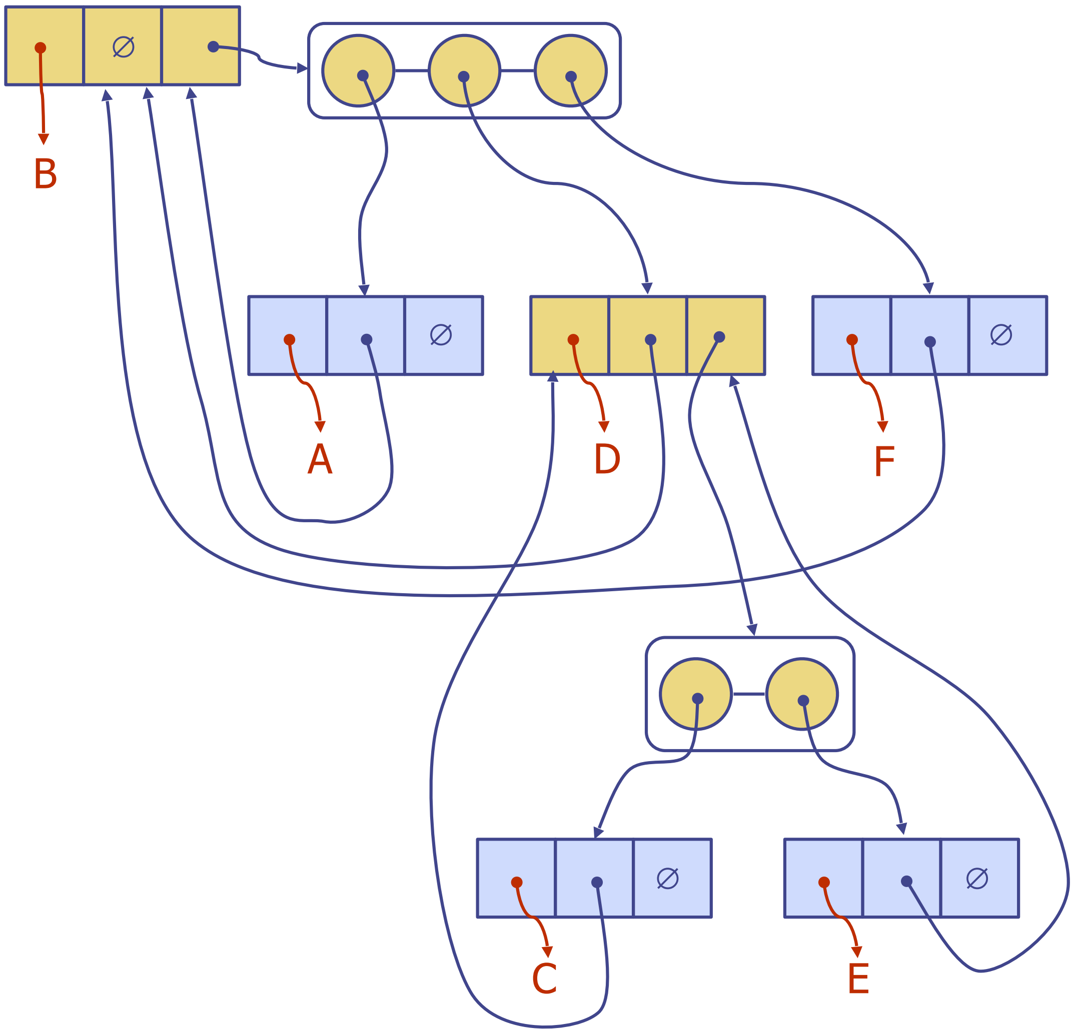
Algorithme *evalExpr*(*v*)
  si *est_feuille*(*v*)
    return *v.element*()
  sinon
    x = ***evalExpr***(***gauche***(***v***))
    y = ***evalExpr***(***droite***(***v***))
    op = opérateur stocké à ***v***
    return *x* **op** *y*

Structure chaînée pour un arbre

Un nœud est représenté par un objet contenant :

- un élément
- son parent
- une séquence de nœuds enfants

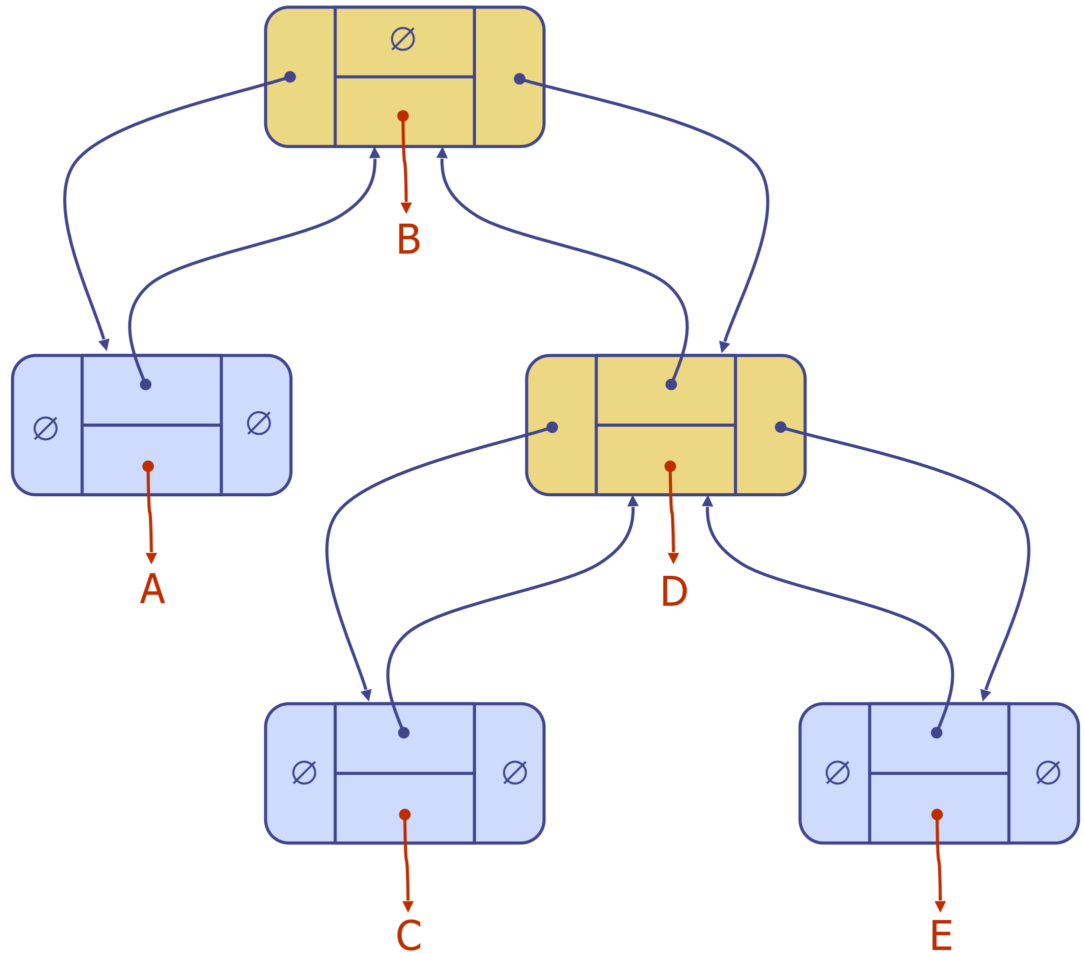


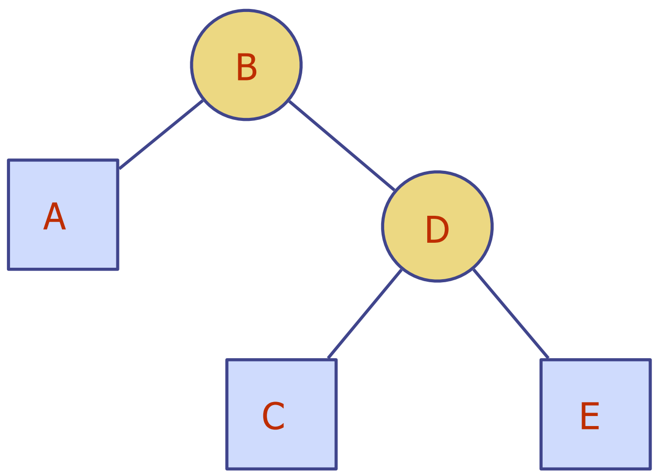


Structure chaînée pour un arbre binaire

Un nœud est représenté par un objet contenant :

- un élément
- son parent
- son enfant gauche
- son enfant droit





```python
# utilise BinaryTree (BinaryTree.py)
from BinaryTree import BinaryTree

# implémentation de BinaryTree avec des noeuds chaînés
class LinkedBinaryTree( BinaryTree ):

    # classe imbriquée _Node
    class _Node:
        # crée une structure statique pour _Node utilisant __slots__
        __slots__ = '_element', '_parent', '_left', '_right'
        def __init__( self, element,
                      parent = None,
                      left = None,
                      right = None ):
            self._element = element
            self._parent = parent
            self._left = left
            self._right = right
```

```python
# classe imbriquée Position, une sous-classe de BinaryTree.Position
class Position( BinaryTree.Position ):

    # constructeur
    # le container (l'arbre) et une référence au noeud sont requis
    # le noeud est de type _Node
    def __init__( self, container, node ):
        self._container = container
        self._node = node

    def __str__( self ):
        return str( self._node._element )

    def element( self ):
        return self._node._element

    # deux Positions sont équivalente si elles sont du même type
    # et réfèrent au même noeud
    def __eq__( self, other ):
        return type( other ) is type( self ) and other._node is self._node

# retourne le noeud d'une Position si valide
# soit, une instance de Position du même container existante
def _validate( self, p ):
    if not isinstance( p, self.Position ):
        raise TypeError( 'p must be proper Position type' )
    if p._container is not self:
        raise ValueError( 'p does not belong to this container' )
    # si p a été deleté (_parent pointe à lui-même: see _delete plus bas)
    if p._node._parent is p._node:
        raise ValueError( 'p is no longer valid' )
    return p._node

#retourne une instance de Position pour un noeud donné (None sinon)
def _make_position( self, node ):
    return self.Position( self, node ) if node is not None else None
```

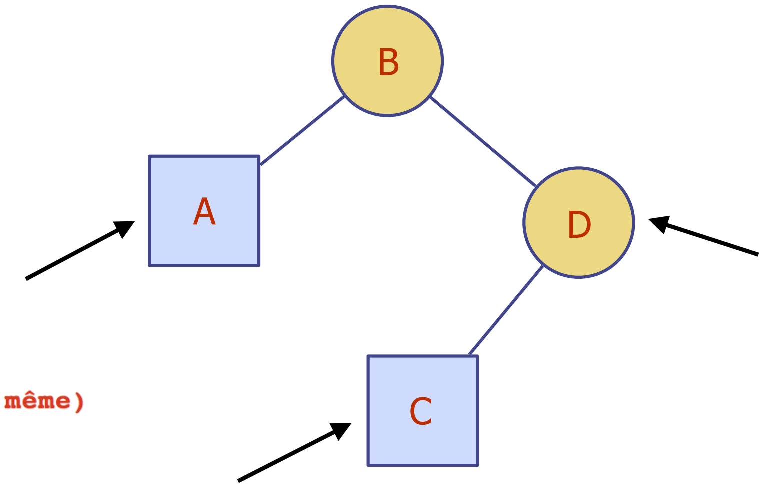
```python
# constructeur d'un BinaryTree
# crée un arbre binaire vide
def __init__( self ):
    self._root = None
    self._size = 0

# retourne la taille
def __len__( self ):
    return self._size

# retourne la racine
def root( self ):
    return self._make_position( self._root )

# retourne le parent d'une Position si valide
def parent( self, p ):
    node = self._validate( p )
    return self._make_position( node._parent )

# retourne l'enfant gauche d'une Position si valide
def left( self, p ):
    node = self._validate( p )
    return self._make_position( node._left )

# retourne l'enfant droit d'une Position si valide
def right( self, p ):
    node = self._validate( p )
    return self._make_position( node._right )

# retourne le nombre d'enfants d'une Position si valide
def num_children( self, p ):
    node = self._validate( p )
    count = 0
    if node._left is not None:
        count += 1
    if node._right is not None:
        count += 1
    return count
```

Méthodes du niveau “développeur”…

```
# ajoute la racine avec valeur e, si elle n'existe pas déjà
# retourne sa Position
def _add_root( self, e ):
    if self._root is not None: raise ValueError( 'Root exists' )
    # taille devient 1
    self._size = 1
    # on crée un noeud pour la racine et retourne sa Position
    self._root = self._Node( e )
    return self._make_position( self._root )

# ajoute un enfant à gauche de valeur e à une Position
# si elle est valide et si cet enfant n'existe pas déjà
def _add_left( self, p, e ):
    node = self._validate( p )
    if node._left is not None: raise ValueError( 'Left child exists' )
    # on incrémente la taille
    self._size += 1
    # on crée un noeud pour l'enfant et on retourne sa Position
    node._left = self._Node( e, node )
    return self._make_position( node._left )

# ajoute un enfant à droite de valeur e à une Position
# si elle est valide et si cet enfant n'existe pas déjà
def _add_right( self, p, e ):
    node = self._validate( p )
    if node._right is not None: raise ValueError( 'Right child exists' )
    # on incrémente la taille
    self._size += 1
    # on crée un noeud pour l'enfant et on retourne sa Position
    node._right = self._Node( e, node )
    return self._make_position( node._right )

# remplace l'élément d'une Position si valide
# retourne l'ancien élément
def _replace( self, p, e ):
    node = self._validate( p )
    old = node._element
    node._element = e
    return old
```

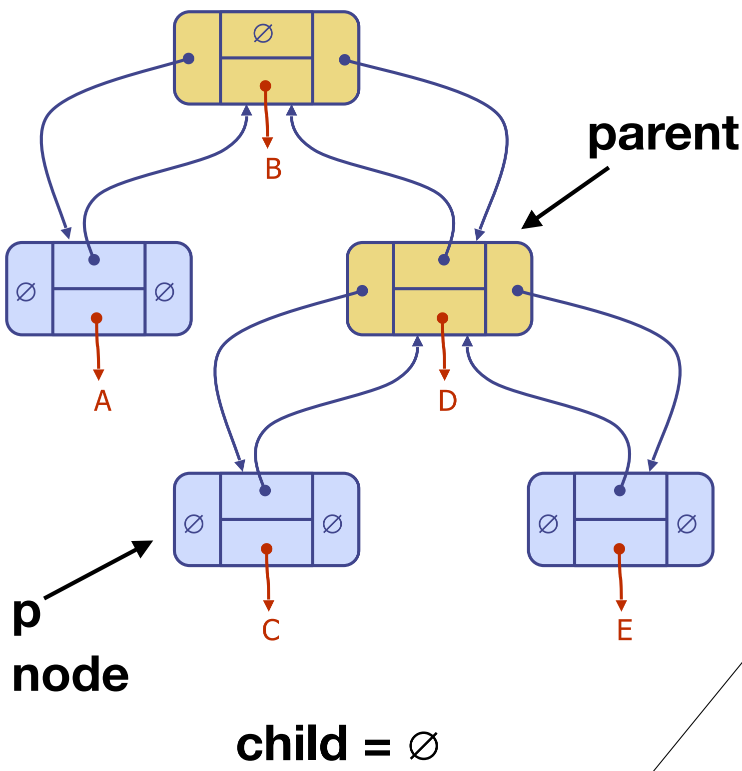
Méthodes du niveau “développeur”…

```
# delete une Position si valide
# la remplace par son enfant s'il y en a un (mais pas 2!)
# retourne l'élément deleté
def _delete( self, p ):
    # validation de la Position
    node = self._validate( p )
    # doit avoir au plus 1 enfant
    if self.num_children( p ) == 2: raise ValueError( 'p has two children' )
    # on prend l'enfant existant ou None s'il n'y en a aucun
    child = node._left if node._left else node._right
    # s'il y a un enfant, il est adopté par son grand-parent
    # ou par personne s'il n'en a pas
    if child is not None:
        child._parent = node._parent
    # si la Position était la racine, la nouvelle racine
    # devient l'enfant
    if node is self._root:
        self._root = child
    # sinon, on remplace le noeud par son enfant
    # de gauche s'il était enfant gauche
    # de droite s'il était enfant droit
    else:
        parent = node._parent
        if node is parent._left:
            parent._left = child
        else:
            parent._right = child
    # on décrémente la taille
    self._size -= 1
    # on rend la Position invalide (en mettant parent sur lui même)
    node._parent = node
    # on retourne l'élément deleté
    return node._element
```

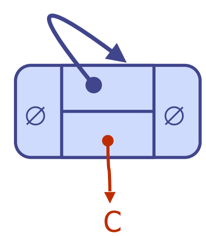


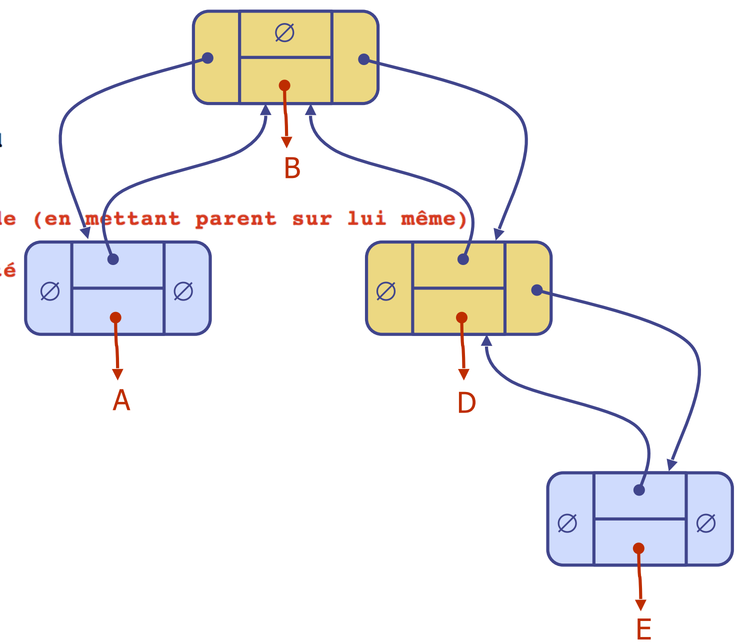
_delete fonctionne sur les noeuds qui possèdent au plus 1 enfant (A, D et C)

_delete(feuille)

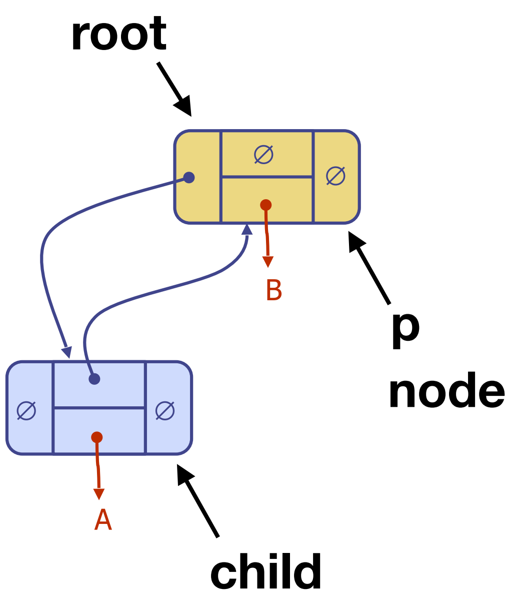



```python
def _delete( self, p ):
    # validation de la Position
    node = self._validate( p )
    # doit avoir au plus 1 enfant
    if self.num_children( p ) == 2: raise ValueError( 'p has two children' )
    # on prend l'enfant existant ou None s'il n'y en a aucun
    child = node._left if node._left else node._right
    # s'il y a un enfant, il est adopté par son grand-parent
    # ou par personne s'il n'en a pas
    if child is not None:
        child._parent = node._parent
    # si la Position était la racine, la nouvelle racine
    # devient l'enfant
    if node is self._root:
        self._root = child
    # sinon, on remplace le noeud par son enfant
    # de gauche s'il était enfant gauche
    # de droite s'il était enfant droit
    else:
        parent = node._parent
        if node is parent._left:
            parent._left = child
        else:
            parent._right = child
    # on décrémente la taille
    self._size -= 1
    # on rend la Position invalide (en mettant parent sur lui même)
    node._parent = node
    # on retourne l'élément deleté
    return node._element
```

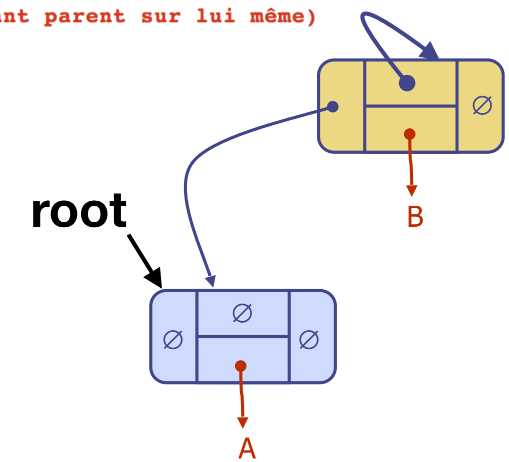




_delete(racine)

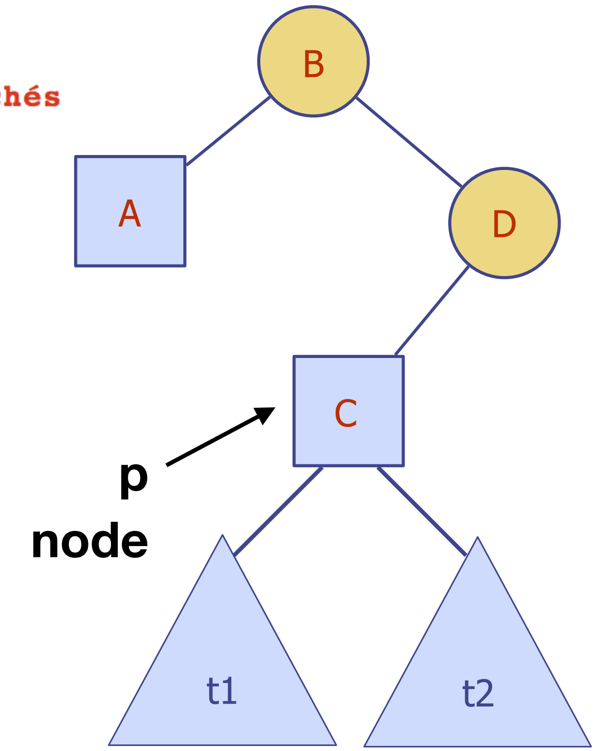



```python
def _delete( self, p ):
    # validation de la Position
    node = self._validate( p )
    # doit avoir au plus 1 enfant
    if self.num_children( p ) == 2: raise ValueError( 'p has two children' )
    # on prend l'enfant existant ou None s'il n'y en a aucun
    child = node._left if node._left else node._right
    # s'il y a un enfant, il est adopté par son grand-parent
    # ou par personne s'il n'en a pas
    if child is not None:
        child._parent = node._parent
    # si la Position était la racine, la nouvelle racine
    # devient l'enfant
    if node is self._root:
        self._root = child
    # sinon, on remplace le noeud par son enfant
    # de gauche s'il était enfant gauche
    # de droite s'il était enfant droit
    else:
        parent = node._parent
        if node is parent._left:
            parent._left = child
        else:
            parent._right = child
    # on décrémente la taille
    self._size -= 1
    # on rend la Position invalide (en mettant parent sur lui même)
    node._parent = node
    # on retourne l'élément deleté
    return node._element
```



Méthodes du niveau “développeur”

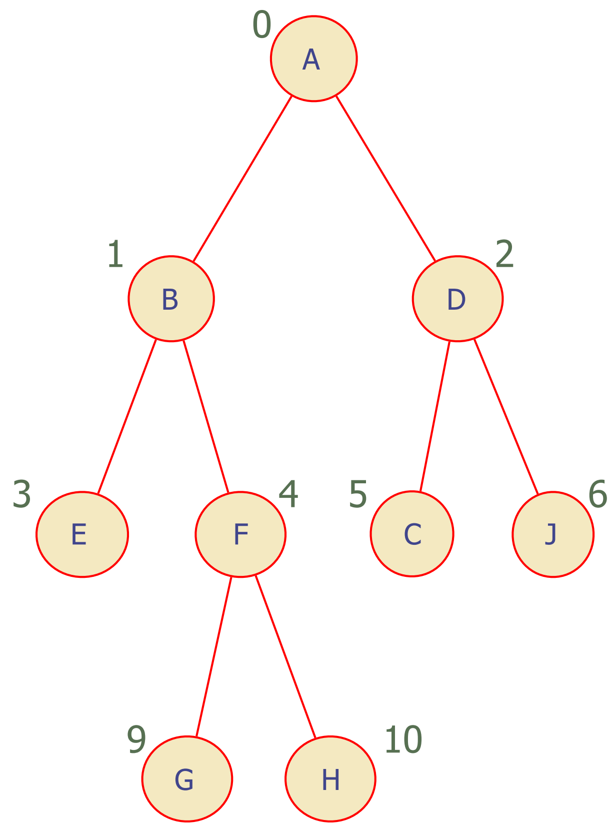
```python
# attache des sous-arbres gauche et droit à une Position feuille si valide
def _attach( self, p, t1, t2 ):
    # validation de p
    node = self._validate( p )
    # s'assurer que c'est une feuille
    if not self.is_leaf( p ): raise ValueError( 'position must be leaf' )
    # s'assurer que les types des sous-arbres sont compatibles
    if not type( self ) is type( t1 ) is type( t2 ):
        raise TypeError( 'Tree types must match' )
    # augmenter la taille de celles des deux sous-arbres attachés
    self._size += len( t1 ) + len( t2 )
    # on attache un sous-arbre non vide
    # en mettant son parent à la feuille d'attache
    # en mettant à None sa racine et à 0 sa taille
    # cet arbre n'existera plus de manière individuelle
    if not t1.is_empty():
        t1._root._parent = node
        node._left = t1._root
        t1._root = None
        t1._size = 0
    if not t2.is_empty():
        t2._root._parent = node
        node._right = t2._root
        t2._root = None
        t2._size = 0
```



_attach fonctionne sur les feuilles uniquement

Tableau pour un arbre binaire

Les nœuds sont stockés dans un tableau A



Le nœud *v* est stocké à *A[index(v)]*

- *index(racine) = 0*
- si le nœud est l'enfant gauche :

  *index(nœud) = 2 * index(parent(nœud)) + 1*
- si le nœud est l’enfant droit :

  *index(nœud) = 2 * index(parent(nœud)) + 2*

Conclusions du module

- Nous avons exploré des structures récursives d'arbre et d'arbre binaire.
- Nous avons regardé la terminologie utilisée pour décrire les noeuds d'un arbre et leurs relations et leurs propriétés.
- Nous avons décrit des méthodes pour parcourir les noeuds d'un arbre et d'un arbre binaire.
- Nous avons défini les ADT pour un arbre (*Tree*) et un arbre binaire (*BinaryTree*) ainsi que des implantations chaînées (*LinkedBinaryTree*) et dans un tableau, dans le cas de l'arbre binaire.
- Nous avons implanté la notion de *Position*, comme nous l'avions fait pour la liste positionnelle.
- Nous nous sommes intéressé à la hauteur d'un arbre binaire et en particulier en pire et meilleur cas.