

Parcours préfixe et postfixe d'un arbre général
Parcours en largeur d'un arbre général
Parcours en ordre d'un arbre binaire
Implémentation
Applications
Parcours d'Euler

Algorithmes de parcours

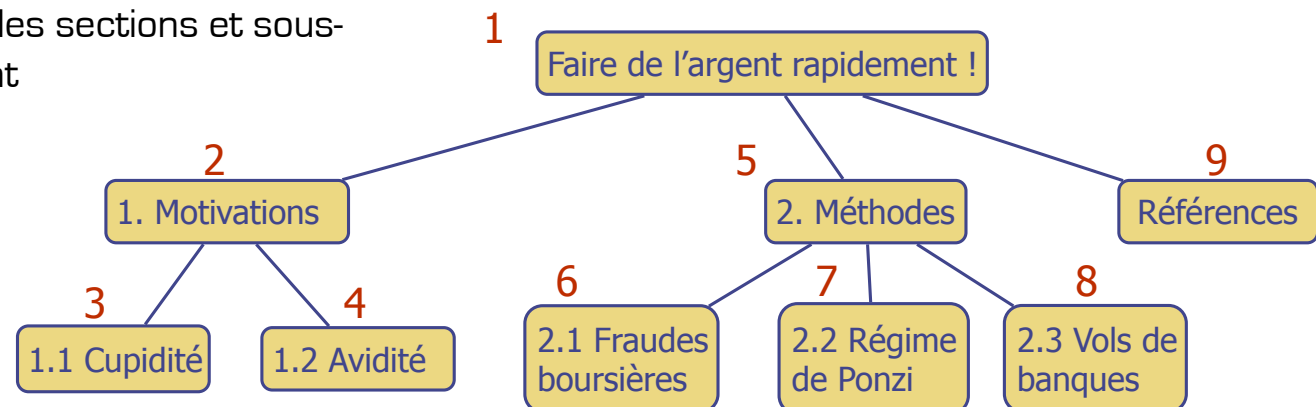
Le parcours d'un arbre T est un moyen systématique d'accéder, ou de “visiter”, toutes ses positions. L'action spécifique associée à la “visite” d'une position p dépend de l'application de ce parcours.

Dans ce sous-module, nous décrivons plusieurs types de parcours d'arbres, nous les implémentons dans le contexte de nos classes d'arbres et nous discutons des applications de parcours d'arbres.

Parcours préfixe

```
Algorithm preorder( p )  
  visit( p )  
  foreach child c in children( p )  
    preorder( c )
```

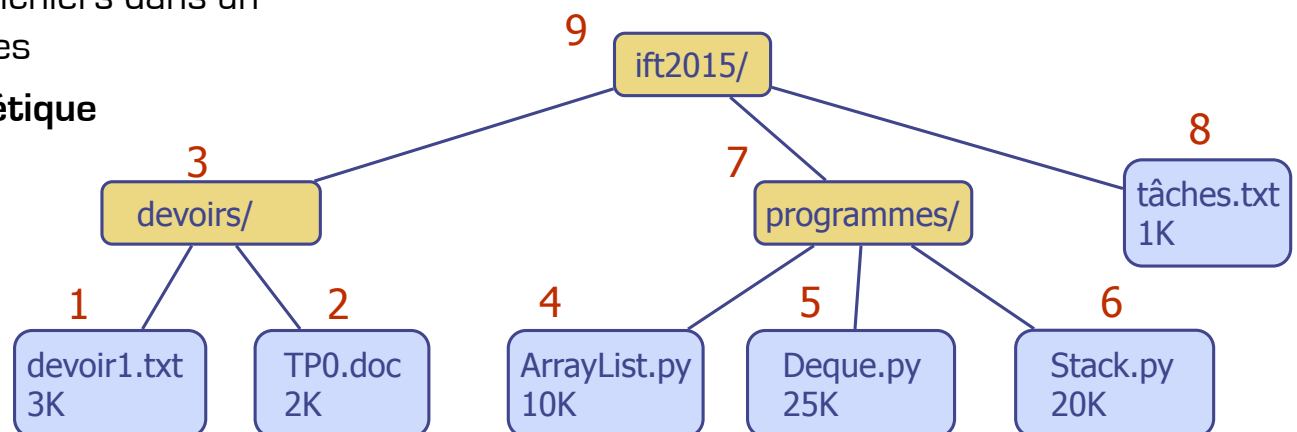
- Dans un parcours de **préfixe**, un noeud est visité avant ses descendants
- On assume que **children**(p) retourne les enfants de gauche à droite
- Application : imprimer les sections et sous-sections d'un document



Parcours postfixe

```
Algorithm postorder( p )  
  foreach child c in children( p )  
    postorder( c )  
  visit( p )
```

- Dans un parcours postfixe, un noeud est visité après ses descendants
- On assume que **children**(p) retourne les enfants de gauche à droite
- Applications :
 - calculer l'espace utilisé par des fichiers dans un répertoire et ses sous-répertoires
 - **Évaluer une expression arithmétique**

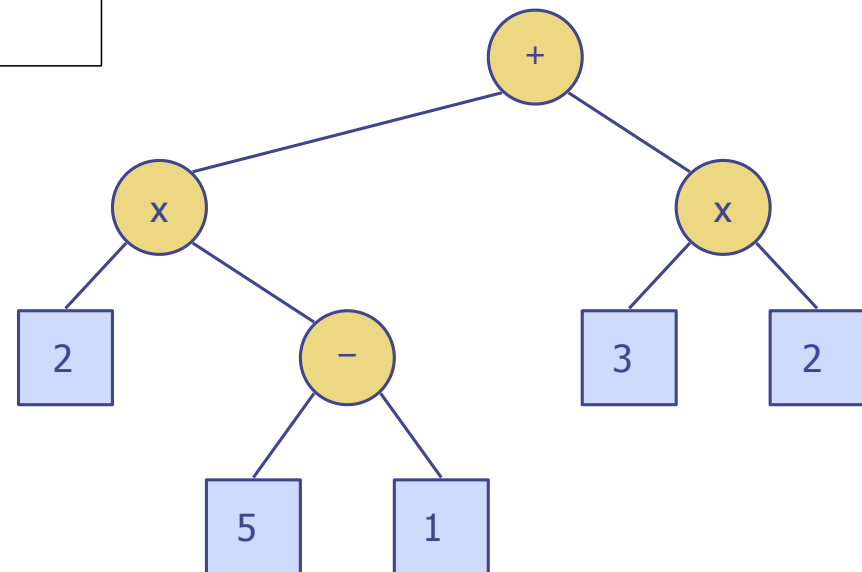


Évaluer une expression arithmétique

```
Algorithm evalExpr( p )  
  if p is a leaf  
    return p.element()  
  else  
    x = evalExpr( left( p ) )  
    y = evalExpr( right( p ) )  
    op = operator in p  
    return x op y
```

Application d'un parcours **postfixe**

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



Parcours en largeur

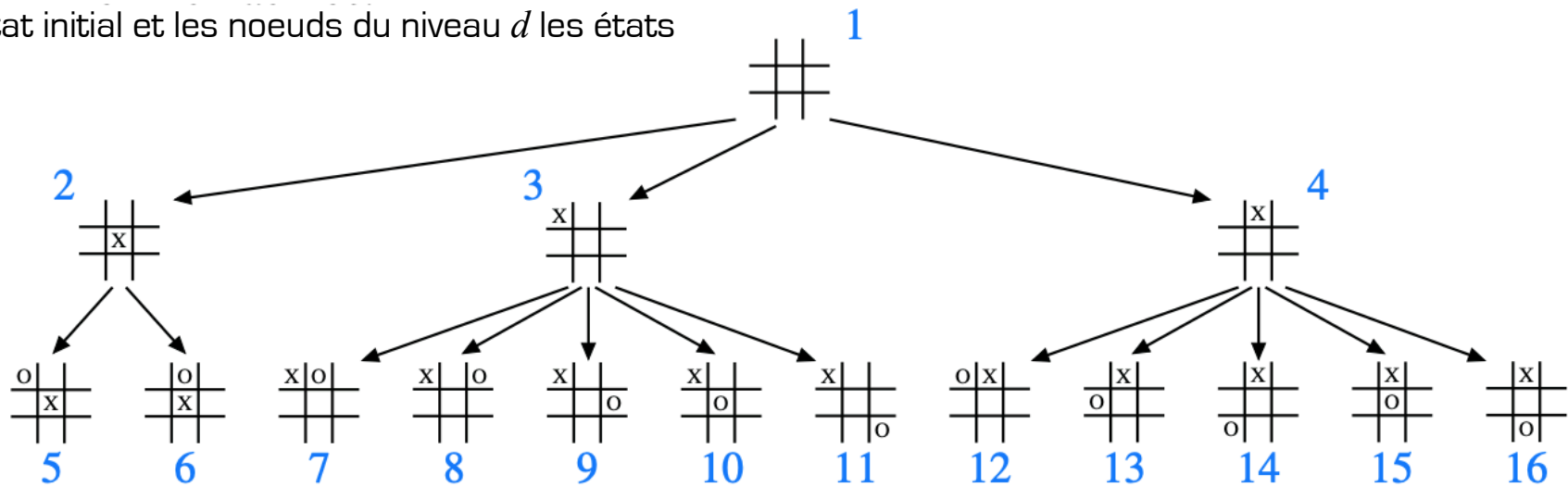
Algorithm **breadthFirst()**

```

initialize queue Q to contain root
while Q is not empty
  p = Q.dequeue()
  visit( p )
  foreach child c in children( p )
    Q.enqueue( c )

```

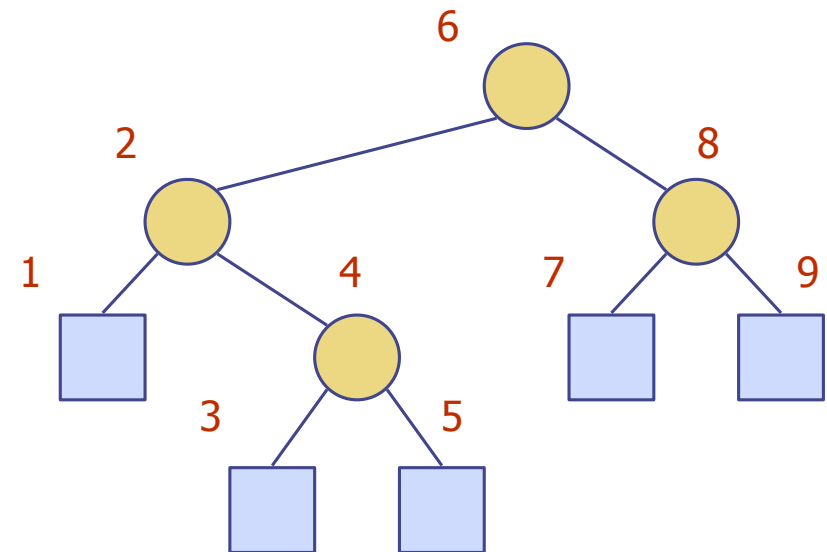
- Dans un parcours en largeur, les noeuds du niveau d sont visités avant les noeuds du niveau $d+1$
- Application : parcourir les états d'un jeu, où la racine représente l'état initial et les noeuds du niveau d les états après d tours



Parcours dans l'ordre d'un arbre binaire

```
Algorithm inorder( p )  
  if p has a left child lc  
    inorder( lc )  
  visit( p )  
  if p has a right child rc  
    inorder( rc )
```

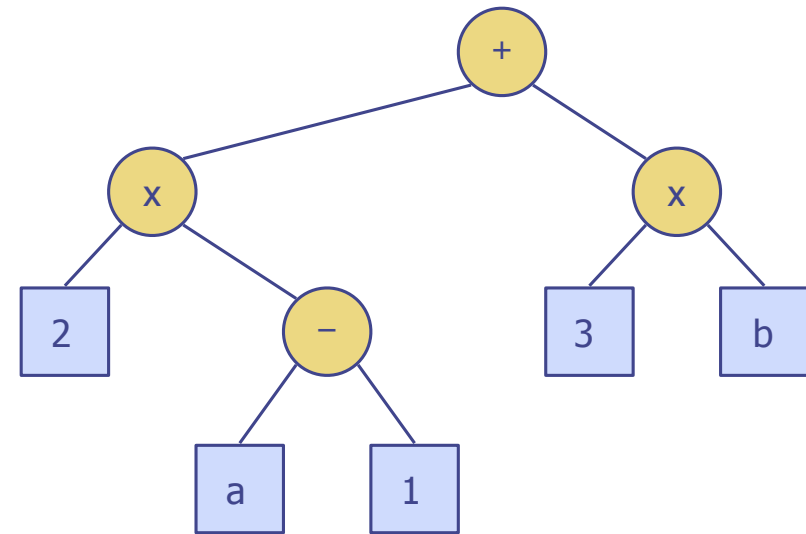
- Dans un parcours dans l'ordre, on visite une position entre les parcours récur­sifs des sous-arbres gauche et droit, respectivement
- Ceci correspond à visiter les noeuds dans l'ordre de gauche à droite
- Application : plusieurs applications, dont **imprimer une expression arithmétique**



Imprimer une expression arithmétique

Allons voir le code...

LinkedBinaryTree.java
PrintExpression.java



$((2 \times (a - 1)) + (3 \times b))$

Application d'un parcours **inorder**

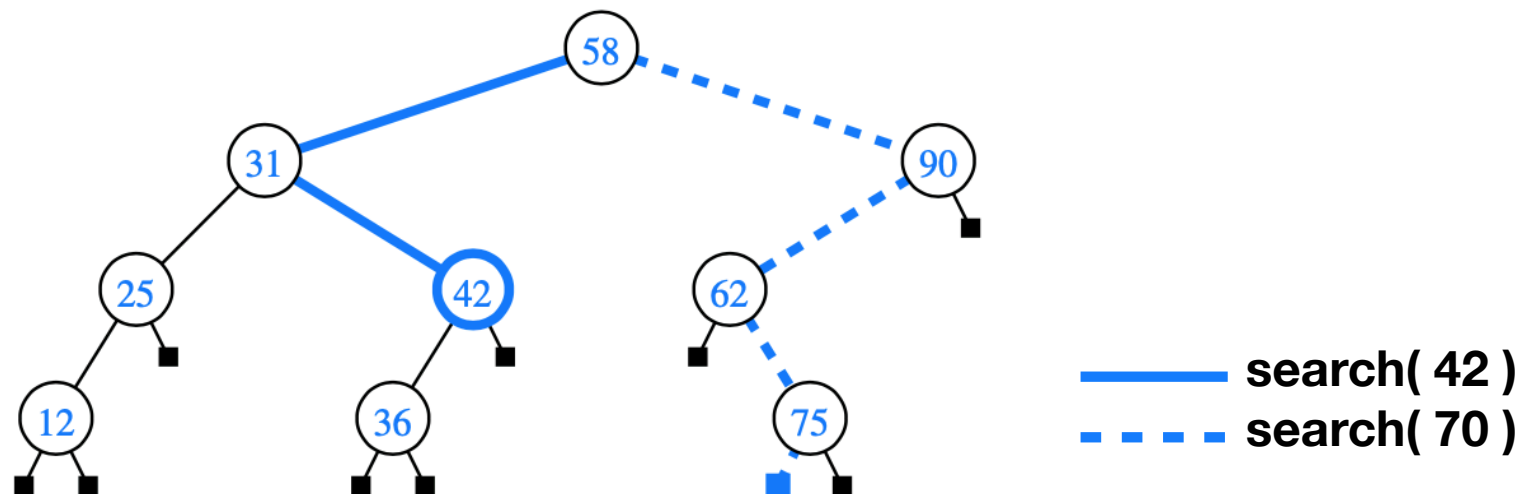
- imprimer l'expression de gauche entre parenthèses
- imprimer l'opération
- imprimer l'expression de droite entre parenthèses

Parcourir (partiellement) un arbre binaire de recherche

```

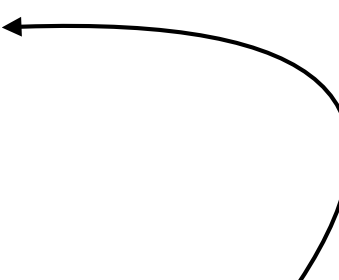
Algorithm contains( p, v )
  if p == null return false
  if v == p.element() return true
  if v < p.element() return contains( getLeft( p ), v )
  return contains( getRight( p ), v )
  
```

- Un arbre binaire de recherche S est un arbre binaire tel que pour chaque noeud interne p :
 - La position p stocke un élément, $p.element()$
 - Les éléments stockés dans le sous-arbre gauche de p sont inférieurs à $p.element()$
 - Les éléments stockés dans le sous-arbre droite de p sont supérieurs à $p.element()$



Implémentation

```
public interface Tree<E> extends Iterable<E> {  
    Position<E> root();  
    Position<E> parent( Position<E> p ) throws IllegalArgumentException;  
    Iterable<Position<E>> children( Position<E> p ) throws IllegalArgumentException;  
    int numChildren( Position<E> p ) throws IllegalArgumentException;  
    boolean isInternal( Position<E> p ) throws IllegalArgumentException;  
    boolean isExternal( Position<E> p ) throws IllegalArgumentException;  
    boolean isRoot( Position<E> p ) throws IllegalArgumentException;  
    int size();  
    boolean isEmpty();  
    Iterator<E> iterator();  
    Iterable<Position<E>> positions();  
}
```



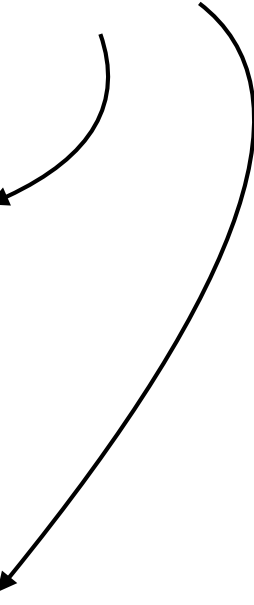
Rappel des méthodes `iterator()` et `positions()` dans l'interface `Tree`

On pourra changer `positions()` en fonction du parcours désiré

```
//----- inner element iterator class
private class ElementIterator implements Iterator<E> {
    Iterator<Position<E>> posIterator = positions().iterator();
    public boolean hasNext() { return posIterator.hasNext(); }
    public E next() { return posIterator.next().getElement(); }
    public void remove() { posIterator.remove(); }
} //----- end inner element iterator class

// return an iterator of the elements stored in the tree
public Iterator<E> iterator() { return new ElementIterator(); }

// return a position iterable of the list
public Iterable<Position<E>> positions() { return preorder(); }
```



Parcours préfixe

On met les positions dans une liste (snapshot) qu'on retourne.

👉 une liste qui implémente l'interface `List<E>` est itérable !

```
// add the positions of the subtree rooted at p to the given snapshot
private void preorderSubtree( Position<E> p, List<Position<E>> snapshot ) {
    snapshot.add( p ); // for preorder, we add position p before exploring subtrees
    for( Position<E> c : children( p ) )
        preorderSubtree( c, snapshot );
}

// return an iterable collection of positions of the tree, reported in preorder
public Iterable<Position<E>> preorder() {
    List<Position<E>> snapshot = new ArrayList<>();
    if( !this.isEmpty() )
        preorderSubtree( this.root(), snapshot ); // fill the snapshot recursively
    return snapshot;
}
```

Parcours postfixe

On met les positions dans une liste (snapshot) qu'on retourne.

☞ une liste qui implémente l'interface `List<E>` est itérable !

Si on veut changer pour le parcours postorder...

```
// add the positions of the subtree rooted at p to the given snapshot
private void postorderSubtree( Position<E> p, List<Position<E>> snapshot ) {
    for( Position<E> c : children( p ) )
        postorderSubtree( c, snapshot );
    snapshot.add( p ); // for postorder, we add position p after exploring subtrees
}
// return an iterable collection of positions of the tree, reported in postorder
public Iterable<Position<E>> postorder() {
    List<Position<E>> snapshot = new ArrayList<>();
    if( !this.isEmpty() )
        postorderSubtree( this.root(), snapshot ); // fill the snapshot recursively
    return snapshot;
}
```

Parcours en largeur

```

public Iterable<Position<E>> breadthfirst() {
    List<Position<E>> snapshot = new ArrayList<>();
    if( !this.isEmpty() ) {
        Queue<Position<E>> fringe = new LinkedList<>();
        fringe.enqueue( this.root() ); // start with root
        while( !fringe.isEmpty() ) {
            Position<E> p = fringe.dequeue(); // remove from front of the queue
            snapshot.add( p ); // report this position
            for( Position<E> c : children( p ) )
                fringe.enqueue( c ); // add children to back of queue
        }
    }
    return snapshot;
}

```

Parcours dans l'ordre

```
// add positions of the subtree rooted at p to the given snapshot
private void inorderSubtree( Position<E> p, List<Position<E>> snapshot ) {
    if( left( p ) != null )
        inorderSubtree( left( p ), snapshot );
    snapshot.add( p );
    if( right( p ) != null )
        inorderSubtree( right( p ), snapshot );
}

// return an iterable collection of positions of the tree, reported in inorder
public Iterable<Position<E>> inorder() {
    List<Position<E>> snapshot = new ArrayList<>();
    if( !this.isEmpty() )
        inorderSubtree( this.root(), snapshot );
    return snapshot;
}

// override positions to make inorder the default for binary trees
public Iterable<Position<E>> positions() { return inorder(); }
```

Parcours d'Euler

(unification des algorithmes de parcours dans un cadre unique : traversée du tour d'Euler)

Algorithm **eulerTour(T, p)**

previsit action for p

foreach child c in T.children(p)

eulerTour(T, c) // recursively tour the subtree rooted at c

postvisit action for p

La traversée du tour d'Euler d'un arbre T peut être définie de manière informelle comme une "marche" autour de T , en suivant l'enveloppe bleue dans la figure, en partant vers la gauche.

Algorithm **eulerTourBinary(T, p)**

previsit action for p

if p has left child lc

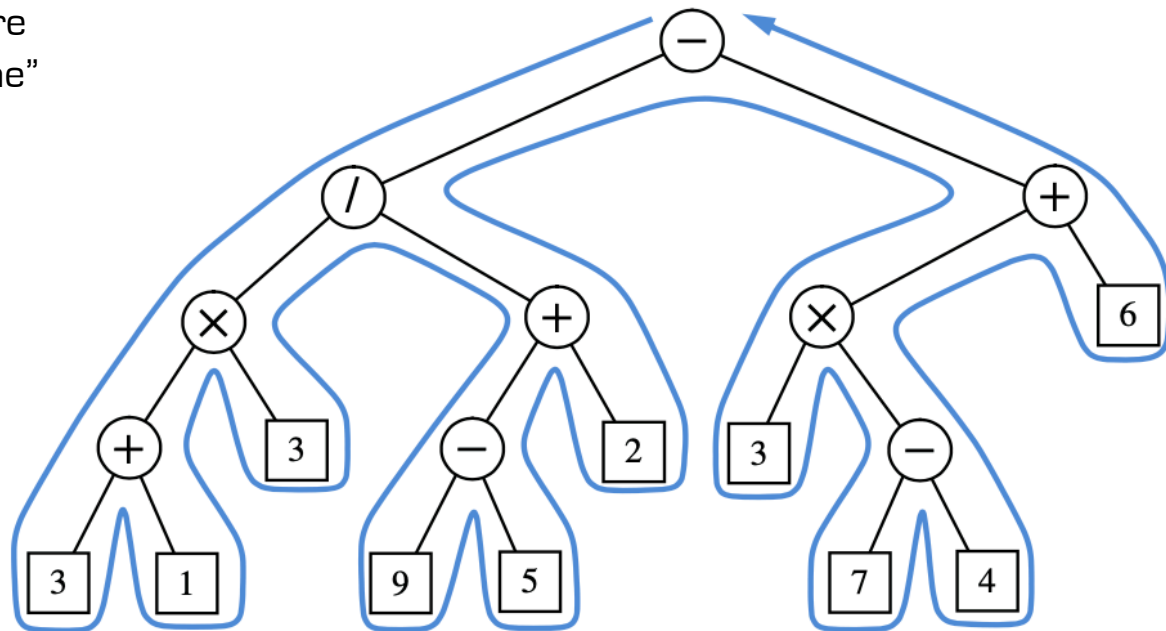
eulerTourBinary(T, lc)

invisit action for p

if p has right child rc

eulerTourBinary(T, rc)

postvisit action for p



Conclusions du module

- Nous avons regardé les méthodes de parcours préfixe et postfixe d'un arbre général
- Nous avons regardé le parcours en largeur
- Nous avons regardé le parcours dans l'ordre d'un arbre binaire
- Nous avons implémenté les parcours d'arbres généraux dans `AbstractTree` et le parcours dans l'ordre d'arbres binaires dans `AbstractBinaryTree`
- Nous avons regardé quelques applications
- Nous avons regardé un cadre général qui unifie les parcours d'arbres: la traversée du tour d'Euler