INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Structures de données

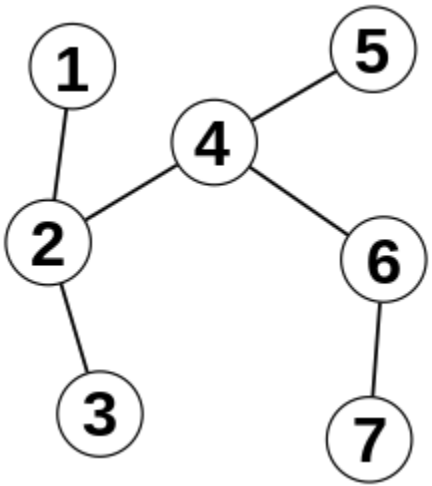
Les arbres (binaires)

Etienne ALLAIN, Hamza AZEROUAL, Maureen BARRAL, Aymen BERRAJAA, Bohan HA, Jean HAUROGNE, Yixuan HU, Alp JAKOP, Yasmine KHALLAAYOUNE, Chenyu LI, Antoine MARCHAL, Guillaume MERCHEZ, Ezgi OZEL

Définitions

Définitions

- arbre orienté

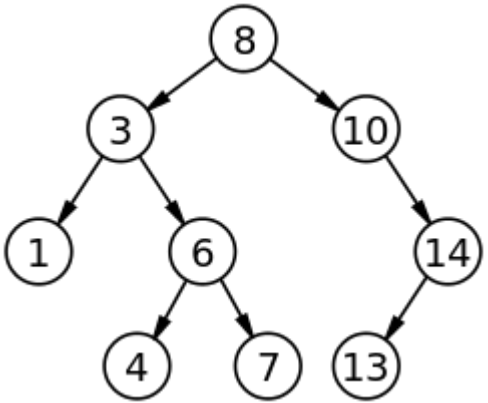


une structure de données telle que:

- un unique élément d'entrée : la racine
- tout élément (sauf la racine) possède un prédécesseur : son père
- on peut atteindre tout élément à partir de la racine par un chemin unique.

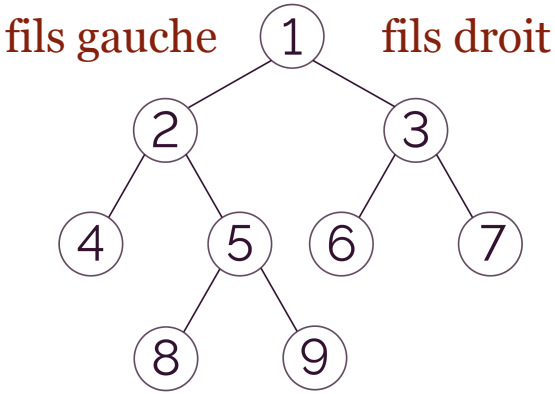
Définitions

- arbre ordonné



tous les successeurs d'un élément sont ordonnés

- arbre binaire

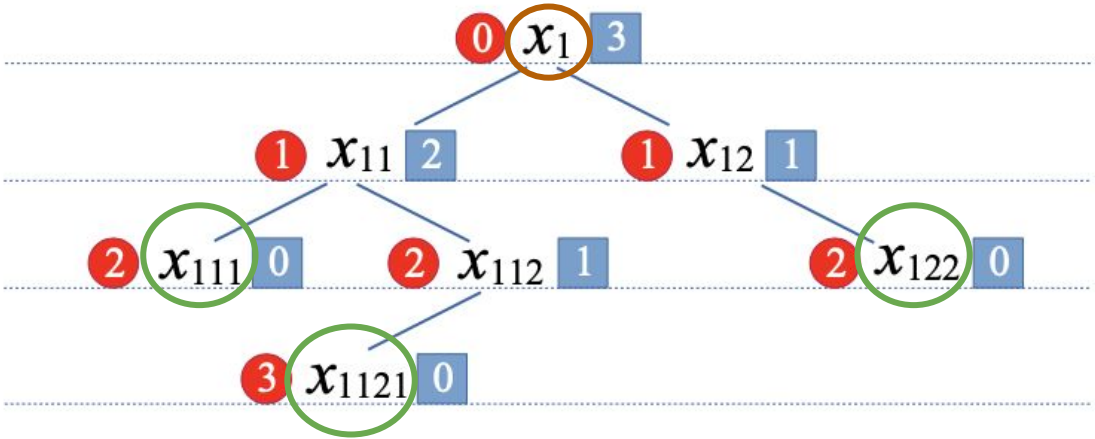


tout élément a, au plus, 2 successeurs

Définitions

- Terminologie usuelle

  - Nœuds de l'arbre: x11, x12, x112…
  - Nœud racine de l'arbre: x1
  - Nœuds feuilles d'arbre: x111, x1121, x122

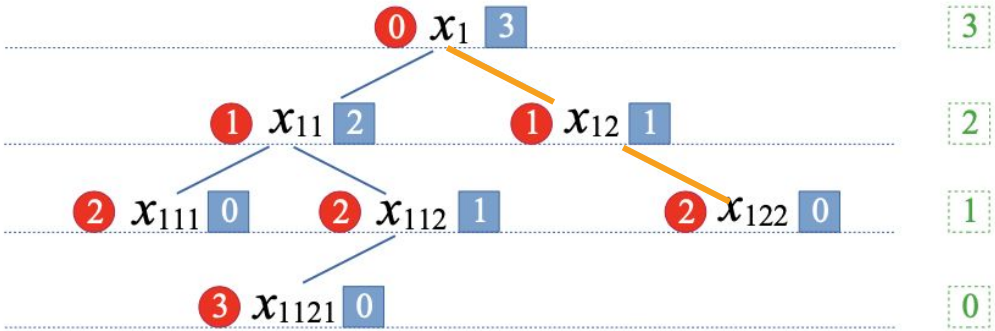


Définitions

- Terminologie usuelle
  - Branche : chemin reliant la racine à une feuille.

    Les nœuds sont reliés par des **arêtes**.
  - Profondeur : longueur du chemin en nombre d'arêtes de la racine au nœud.
  - Hauteur : longueur maximale des chemins allant d'un nœud à une feuille.

    Remarque : la hauteur d'un arbre est la hauteur de son noeud racine
  - Niveau : hauteur du nœud racine - profondeur du nœud.

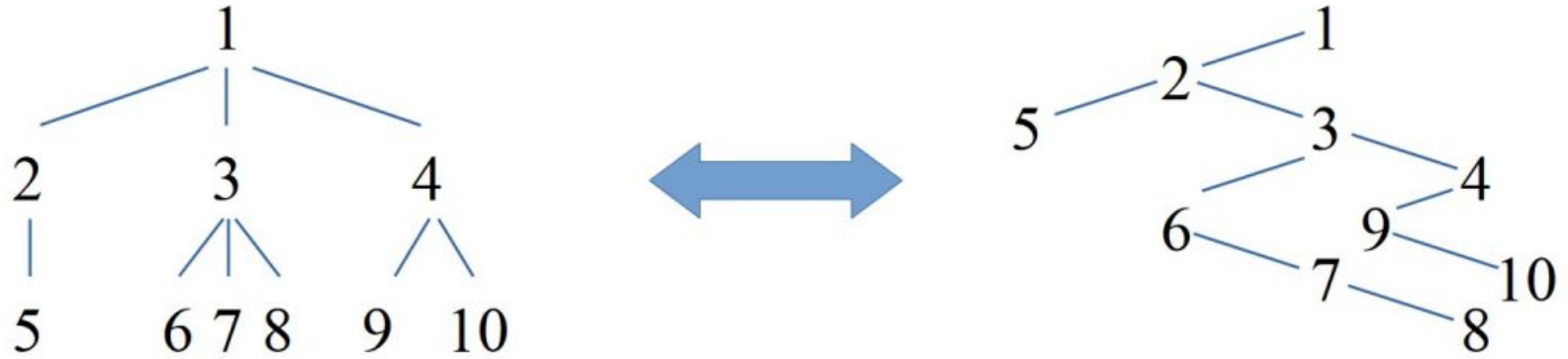


Remarque

- Tout arbre n-aire peut être converti en arbre binaire
    - Il existe un procédé bijectif qui permet de passer de l'un à l'autre
    - On peut limiter l'étude aux arbres binaires

- Procédé à suivre :
    - On place les fils d'un noeud successivement dans le sous-arbre gauche de ce noeud
    - On place les frères successivement dans le sous-arbre droit de ce noeud

Conversion d’un arbre n-aire en binaire

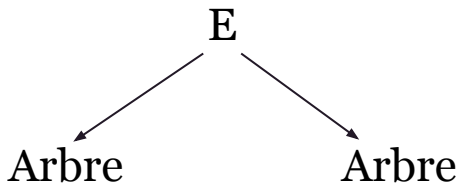


TAD

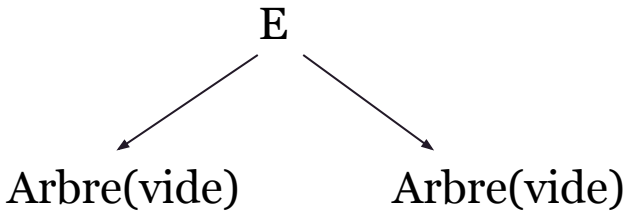
b) TAD Arbre

```
sorte Arbre[E] ⟹  Arbres dont l'élément est le type E
utilise Booléen
fonctions
    new :          ⟶ Arbre[E]                                  } 2 constructeur
    new :          E×Arbre[E]×Arbre[E] ⟶ Arbre[E]              }
    estVide :      Arbre[E] ⟶ Booléen
    vider :        Arbre[E] ⟶ Arbre[E]
    racine :       Arbre[E] -/⟶ E
    arbreG :       Arbre[E] ⟶ Arbre[E]
    arbreD :       Arbre[E] ⟶ Arbre[E]
    modifRacine : Arbre[E]×E ⟶ Arbre[E]
    modifArbreG : Arbre[E]×Arbre[E] ⟶ Arbre[E]
    modifArbreD : Arbre[E]×Arbre[E] ⟶ Arbre[E]

préconditions
    ∀ a de type Arbre[E]
    pré(racine(a)) = non estVide(a)

axiomes
    ∀ a,g,d de type Arbre[E], ∀ r de type E
    racine(new(r,g,d)) = r
    arbreG(new(r,g,d)) = g
    arbreD(new(r,g,d)) = d

    estVide(new) = VRAI
    estVide(new(r,g,d)) = FAUX
    estVide(vider(a)) = VRAI
    estVide(arbreG(new)) = VRAI
    estVide(arbreD(new)) = VRAI

    racine(modifRacine(a,r)) = r
    arbreG(modifArbreG(a,g)) = g
    arbreD(modifArbreD(a,d)) = d
```

Noeud :



Feuille :



Pratique : pas de précondition pour arbreG et arbreD

Variation possible du TAD

- Un troisième constructeur arbre

```
new : E -> Arbre[E]
```

- Fonctions :

```
estUneFeuille : Arbre[E] -> Booléen
```

- Des préconditions ajoutées pour arbreG et arbreD :

```
pré(arbreG(a)) = non estUneFeuille(a)
pré(arbreD(a)) = non estUneFeuille(a)
```

- axiomes modifiés :

```
estVide(new(r)) = FAUX
racine(new(r)) = r
```

~~estVide(arbreG(new)) = VRAI~~
~~estVide(arbreD(new)) = VRAI~~

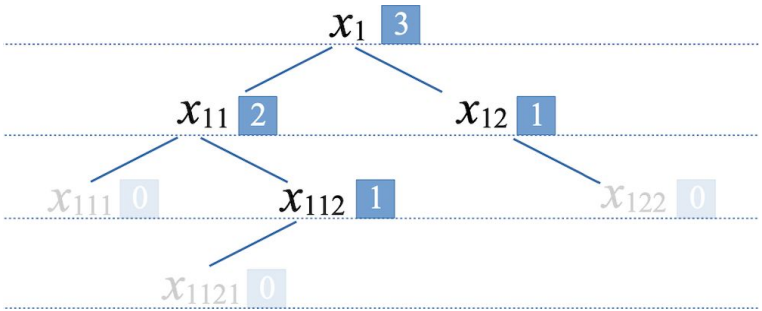
Calcul de la hauteur d’un arbre

Calcul de la hauteur d'un arbre

```
int hauteur(Arbre a) {
  if (estVide(a)) {
     return -1 ;
  } else {
     return max(hauteur(arbreG(a)),hauteur(arbreD(a))) + 1 ;
  }
}
```

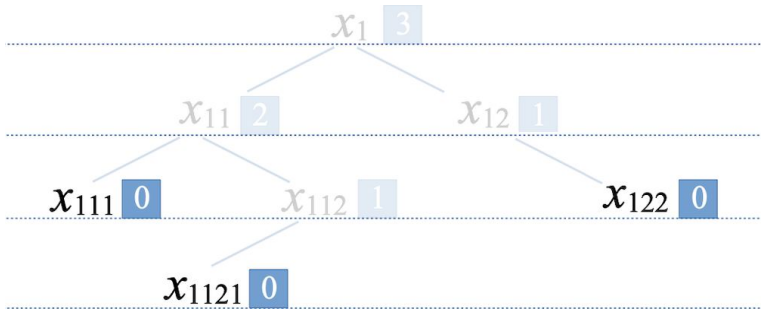
Calcul de la hauteur d'un arbre

```
int hauteur(Arbre a) {
  if (estVide(a)) {
     return -1 ;
  } else {
     return max(hauteur(arbreG(a)),hauteur(arbreD(a))) + 1 ;
  }
}
```



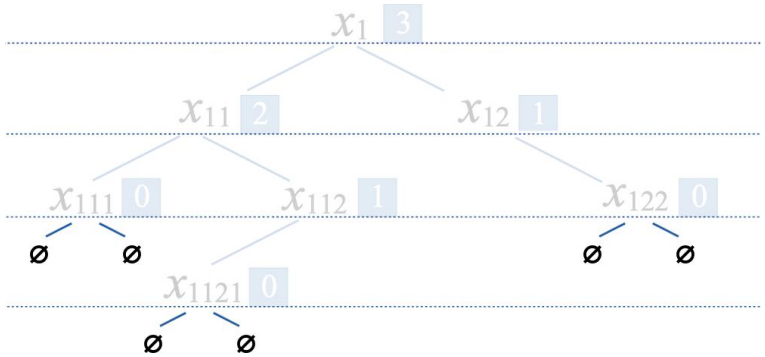
Calcul de la hauteur d'un arbre

```
int hauteur(Arbre a) {
  if (estVide(a)) {
     return -1 ;
  } else {
     return max(hauteur(arbreG(a)),hauteur(arbreD(a))) + 1 ;
  }
}
```



Calcul de la hauteur d'un arbre

```
int hauteur(Arbre a) {
  if (estVide(a)) {
     return -1 ;
  } else {
     return max(hauteur(arbreG(a)),hauteur(arbreD(a))) + 1 ;
  }
}
```



Calcul de la hauteur d'un arbre

```
int hauteur(Arbre a) {...}

int hauteurArbre(Arbre a) catch estVideException {
  int haut = hauteur(a) ;
  if (haut == -1) {
       throw new estVideException("arbre vide") ;
  }
  return haut ;
}
```

Calcul de la hauteur d'un arbre

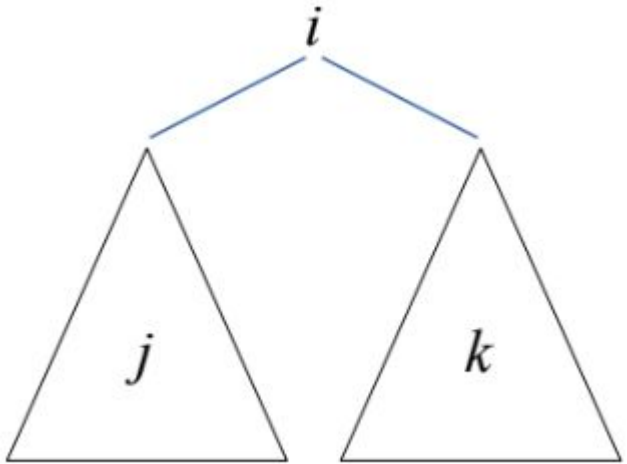
```
int hauteur(Arbre a) {...}

int hauteurArbre(Arbre a) catch estVideException {
  int haut = hauteur(a) ;
  if (haut == -1) {
       throw new estVideException("arbre vide") ;
  }
  return haut ;
}
```

Parcours d’arbres en profondeur

Parcours d’arbres en profondeur

3 parcours classiques d’arbres:

- préfixe
- infixe
- postfixe

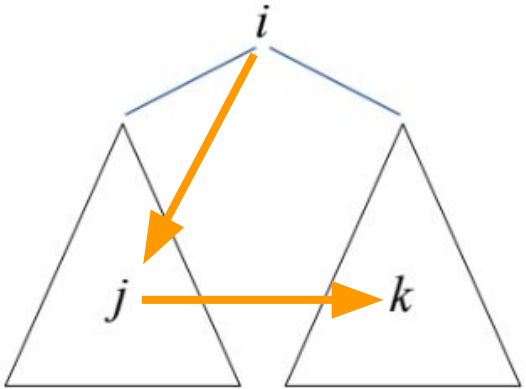


∀ nœud $i \in$ arbre
∀ nœud $j \in$ sous-arbre gauche de i
∀ nœud $k \in$ sous-arbre droit de i

Parcours d'arbres en profondeur

- Le parcours **préfixe**

  visite(i)<visite(j)<visite(k)

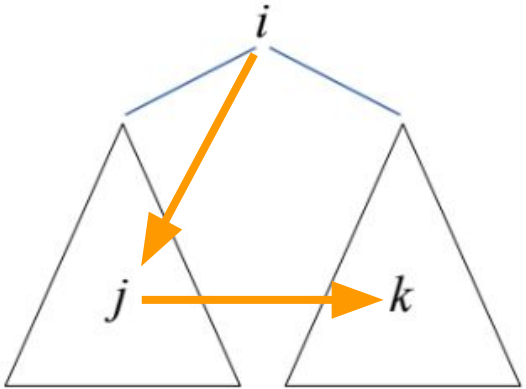


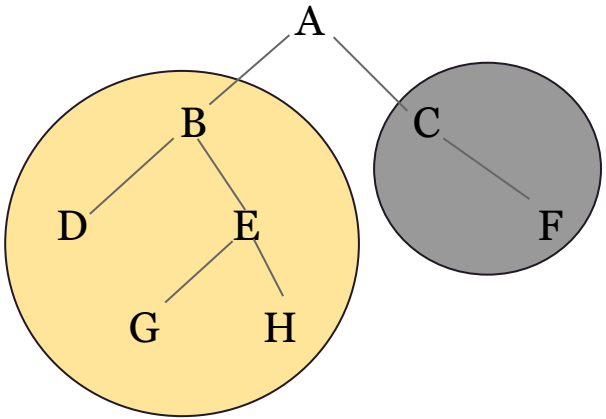

```
algorithme parcoursPrefixe (a : Arbre)
     si non estVide(a) alors
          traiter(racine(a))
          parcoursPrefixe(arbreG(a))
          parcoursPrefixe(arbreD(a))
     fin
fin
```

Parcours d'arbres en profondeur

- Le parcours **préfixe**

  visite(i)<visite(j)<visite(k)



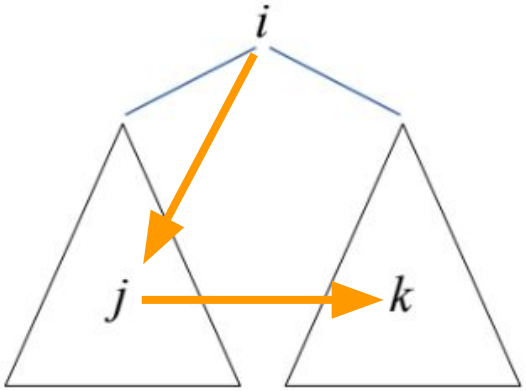


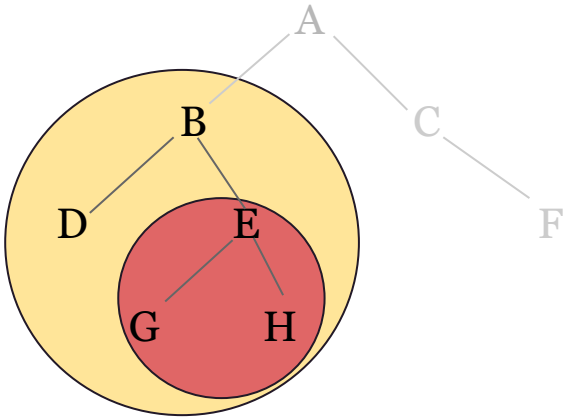
A ( ) ( )

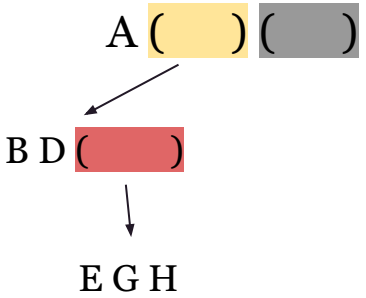
Parcours d'arbres en profondeur

- Le parcours **préfixe**

  visite(i)<visite(j)<visite(k)



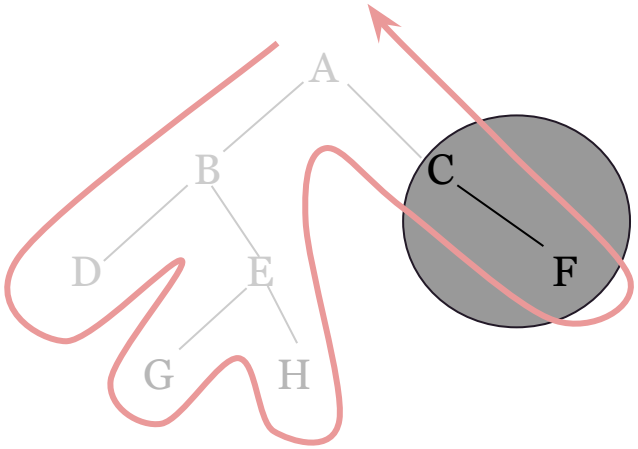


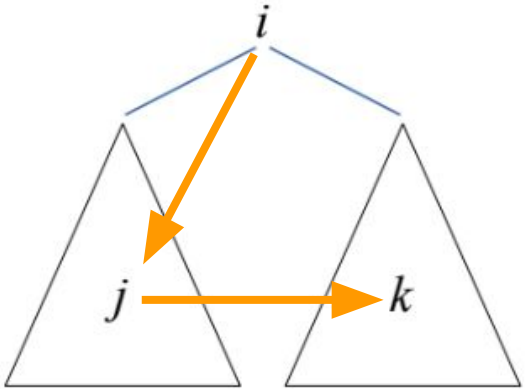


Parcours d'arbres en profondeur

- Le parcours **préfixe**

  visite(i)<visite(j)<visite(k)





A (B D (E G H)) ( )

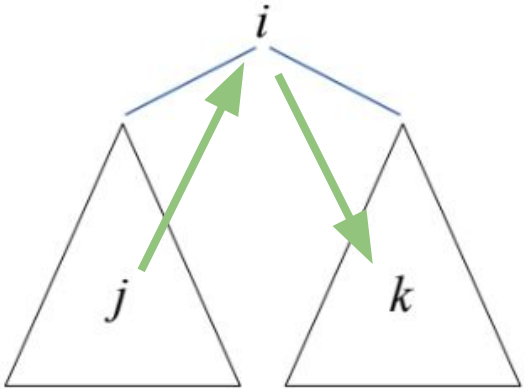
C F

A (B D (E G H)) (C F)

Parcours d'arbres en profondeur

- Le parcours **infixe**

  visite(j)<visite(i)<visite(k)

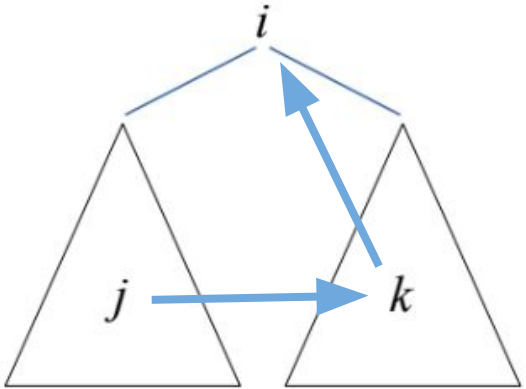



```
algorithme parcoursInfixe (a : Arbre)
     si non estVide(a) alors
          parcoursInfixe(arbreG(a))
          traiter(racine(a))
          parcoursInfixe(arbreD(a))
     fin
fin
```

Parcours d'arbres en profondeur

- Le parcours **postfixe**

  visite(j)<visite(k)<visite(i)

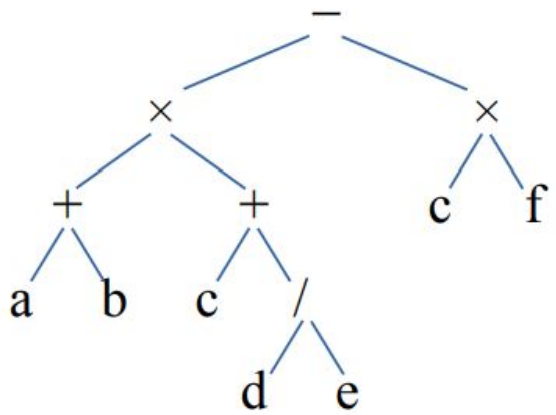



```
algorithme parcoursPostfixe (a : Arbre)
     si non estVide(a) alors
          parcoursPostfixe(arbreG(a))
          parcoursPostfixe(arbreD(a))
          traiter(racine(a))
     fin
fin
```

Exercices sur les expressions

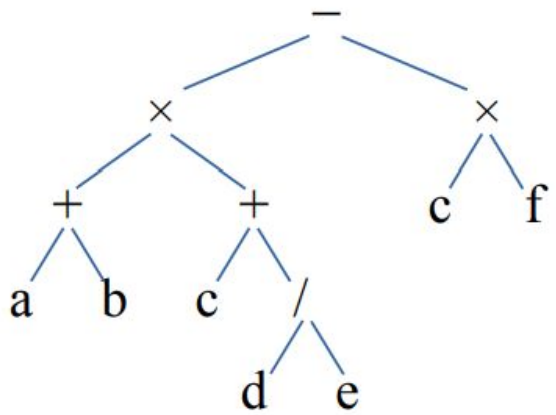
Exercices sur les expressions

Traduire l'expression **((a+b)*(c+(d/e)))-(c*f)** sous forme d'arbre binaire :



Exercices sur les expressions

Lecture de cet arbre en mode infixe, préfixe et postfixe

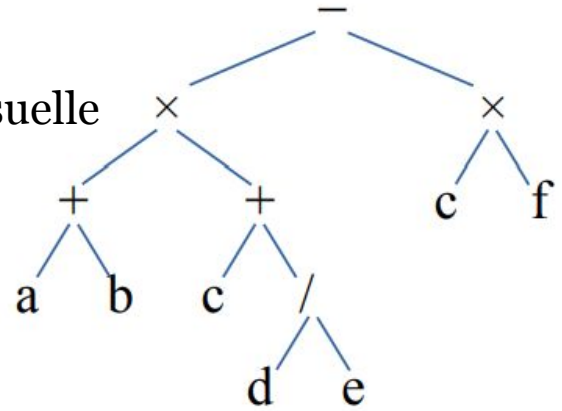


Exercices sur les expressions

Lecture de cet arbre en mode infixe, préfixe et postfixe

infixe -> **(a+b)*(c+(d/e))-c*f**

Notation mathématique usuelle

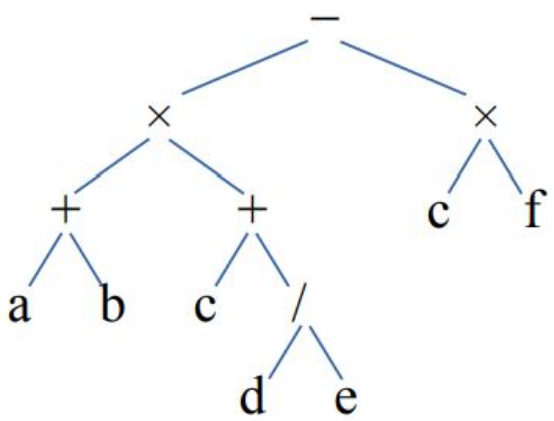


Exercices sur les expressions

Lecture de cet arbre en mode infixe, préfixe et postfixe

préfixe -> **-*+ab+c/de*cf**

Notation fonctionnelle usuelle:
sub(mul(add(a, b), add(c, div(d, e))), mul(c, f))

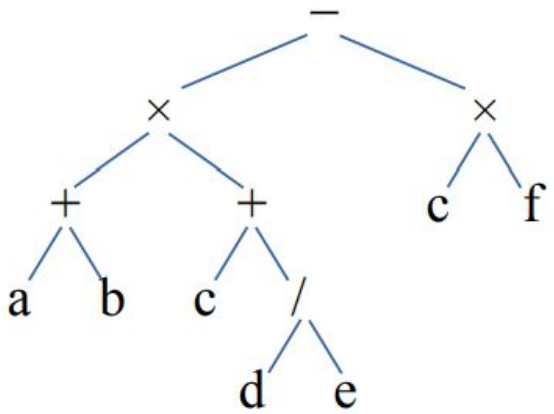


Exercices sur les expressions

Lecture de cet arbre en mode infixe, préfixe et postfixe

postfixe -> **ab+cde/+*cf*-**

Notation polonaise inverse



Un peu de programmation:

Représentation d’expressions arithmétiques en Java (préparation au TP)

Mise en place de notre structure de données

Création d'une interface *INoeud* représentant un noeud de l'arbre:

```
public interface INoeud {
    int evaluer();
}
```

Mise en place de notre structure de données

Création d'une classe abstraite NoeudBinaire représentant une opération binaire:

```java
public abstract class NoeudBinaire implements INoeud {
    private INoeud gauche;
    private INoeud droit;

    public NoeudBinaire(INoeud gauche, INoeud droit) {
        this.gauche = gauche;
        this.droit = droit;
    }

    public INoeud getGauche() {
        return gauche;
    }

    public INoeud getDroite() {
        return droit;
    }
}
```

Mise en place de notre structure de données

Création des opérations binaires: les classes NoeudAddition et NoeudMultiplication

```java
public class NoeudAddition extends NoeudBinaire {
    public NoeudAddition(INoeud gauche, INoeud droite) {
        super(gauche, droite);
    }

    @Override
    public int evaluer() {
        int g = getGauche().evaluer();
        int d = getDroite().evaluer();

        return g + d;
    }
}
```

```java
public class NoeudMultiplication extends NoeudBinaire {
    public NoeudMultiplication(INoeud gauche, INoeud droite) {
        super(gauche, droite);
    }

    @Override
    public int evaluer() {
        int g = getGauche().evaluer();
        int d = getDroite().evaluer();

        return g * d;
    }
}
```

Mise en place de notre structure de données

Création des constantes: la classe NoeudNombre

```java
public class NoeudNombre implements INoeud {
    private int valeur;

    public NoeudNombre(int valeur) {
        this.valeur = valeur;
    }

    @Override
    public int evaluer() {
        return valeur;
    }
}
```

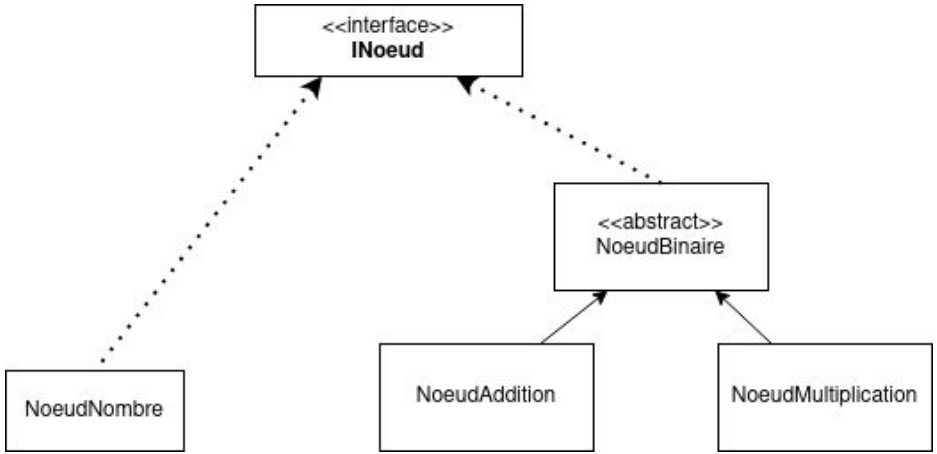
Mise en place de notre structure de données

Création de la structure représentant notre arbre: la classe Expression

```
public class Expression implements INoeud {
    private INoeud racine;

    public Expression(INoeud racine) {
        this.racine = racine;
    }

    @Override
    public int evaluer() {
        return racine.evaluer();
    }
}
```

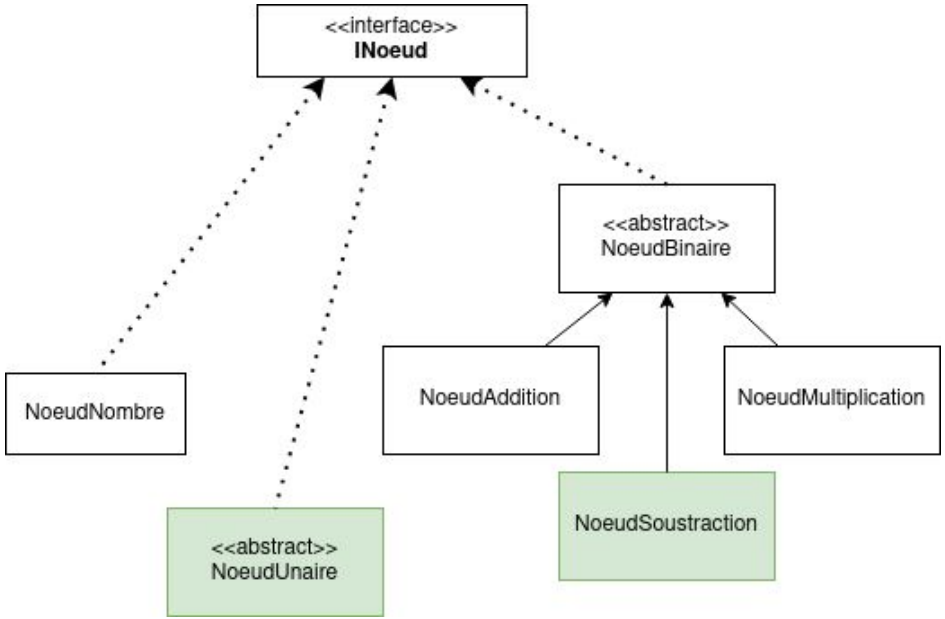
Mise en place de notre structure de données

Le diagramme de classes de notre code



Mise en place de notre structure de données

La qualité de notre solution



Utilisation de notre structure de données

Création de l'expression "(2+3) * 4"

Code

```
Expression e = new Expression(

);
```

Arbre correspondant

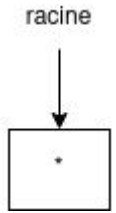
Utilisation de notre structure de données

Création de l'expression "(2+3) * 4"

Code

```
Expression e = new Expression(
    new NoeudMultiplication(

    )
);
```

Arbre correspondant



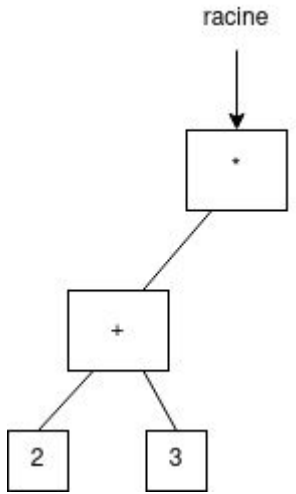
Utilisation de notre structure de données

Création de l'expression "(2+3) * 4"

Code

```
Expression e = new Expression(
    new NoeudMultiplication(
        new NoeudAddition(
            new NoeudNombre(2),
            new NoeudNombre(3)
        )
    )
);
```

Arbre correspondant



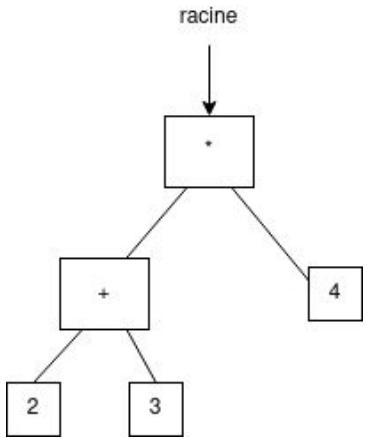
Utilisation de notre structure de données

Création de l'expression "(2+3) * 4"

Code

```
Expression e = new Expression(
    new NoeudMultiplication(
        new NoeudAddition(
            new NoeudNombre(2),
            new NoeudNombre(3)
        ),
        new NoeudNombre(4)
    )
);
```

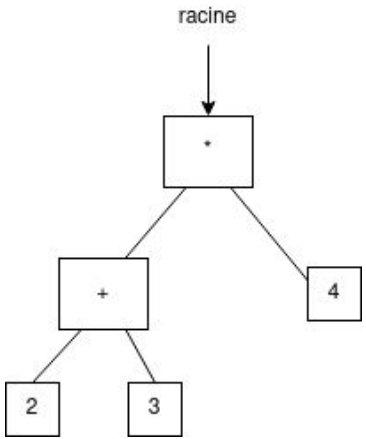
Arbre correspondant



Utilisation de notre structure de données

Affichage du résultat

```
System.out.println(e.evaluer()); //Affiche 20
```



Ce que vous aurez à faire…

- Affichage du calcul représenté par l'arbre (par exemple, afficher "((2+3)*4) dans la console)
- Gestion des erreurs (que se passe-t-il si l'une des opérandes est *null* ?)
- Simplification de l'arbre (par exemple, un NoeudAddition(3, 0) peut être transformé en NoeudNombre(3)
- Possibilité d'utiliser des symboles (comme π, e)

Arbres de recherche

Concept

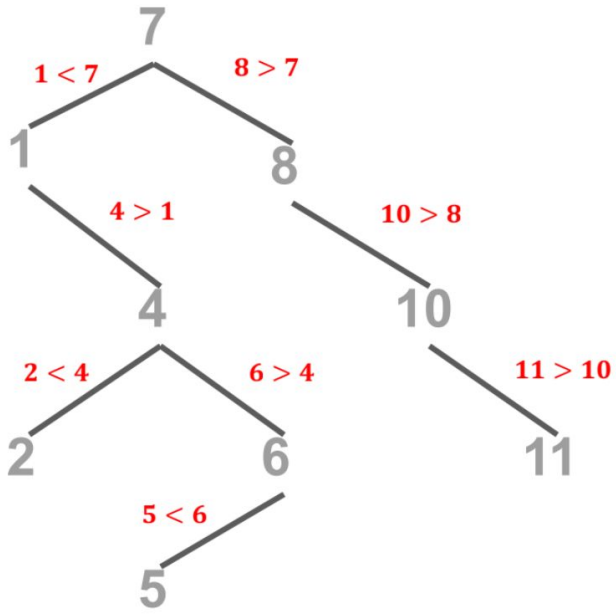
Utiliser les arbres pour stocker des données de manière ordonnée, en suivant une certaine relation d'ordre qui permet de se placer aux noeuds de l'arbre en respectant la règle suivante :

$\forall$ nœud x $\in$ arbre :

- $x_i \in$ sous-arbre gauche de $x \Rightarrow x_i < x$
- $x_j \in$ sous-arbre droit de $x \Rightarrow x_j \geqslant x$

Exemple

Par exemple, l'ajout successif des valeurs 7, 1, 8, 4, 6, 10, 11, 2, 5, donne :



L'ordre de l'entrée est important

Exercice

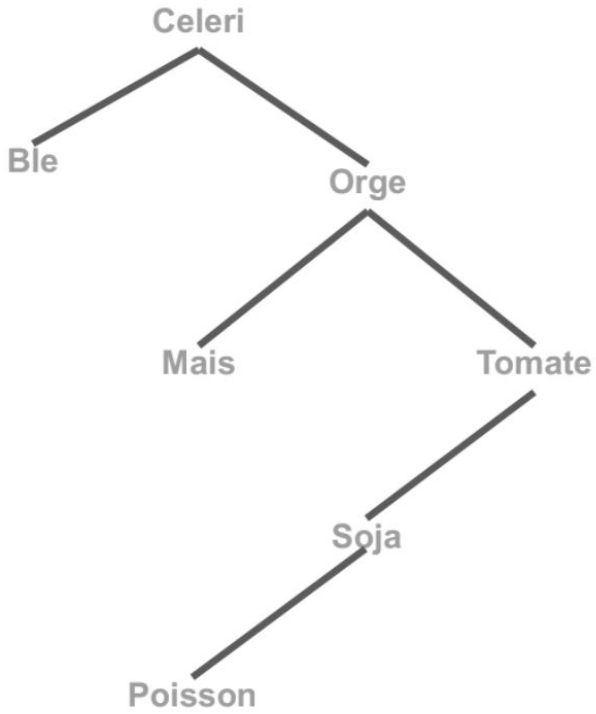
Transformer cette entrée en un arbre de String

Entrée : ["celeri","orge", "mais","ble","tomate","soja", "poisson"]

Réponse

Entrée : ["celeri","orge", "mais","ble","tomate","soja", "poisson"]

Ici l’ordre est donc l ’ordre alphabétique



Utilité

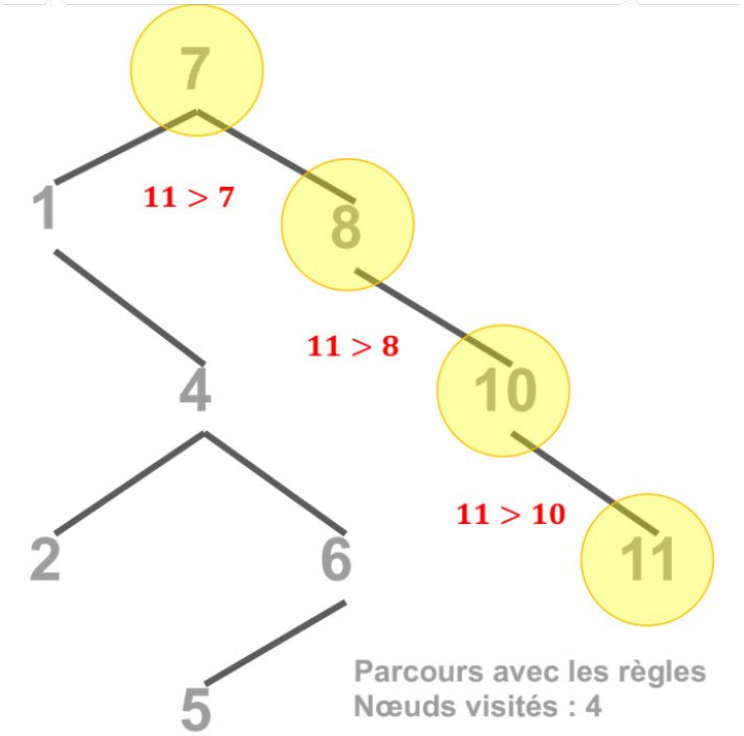
Recherches plus rapides que dans un arbre "classique"

Réduit le nombre de nœuds à visiter

Exemple:

On cherche 11

dans l'arbre



Question

Est-il possible d’obtenir une liste triée à partir d’un arbre de recherche ?

Réponse

Oui, un simple parcours infixe d'un arbre de recherche fournit les valeurs triées dans l'ordre.

Sortie : [1, 2, 4, 5, 6, 7, 8, 10, 11]