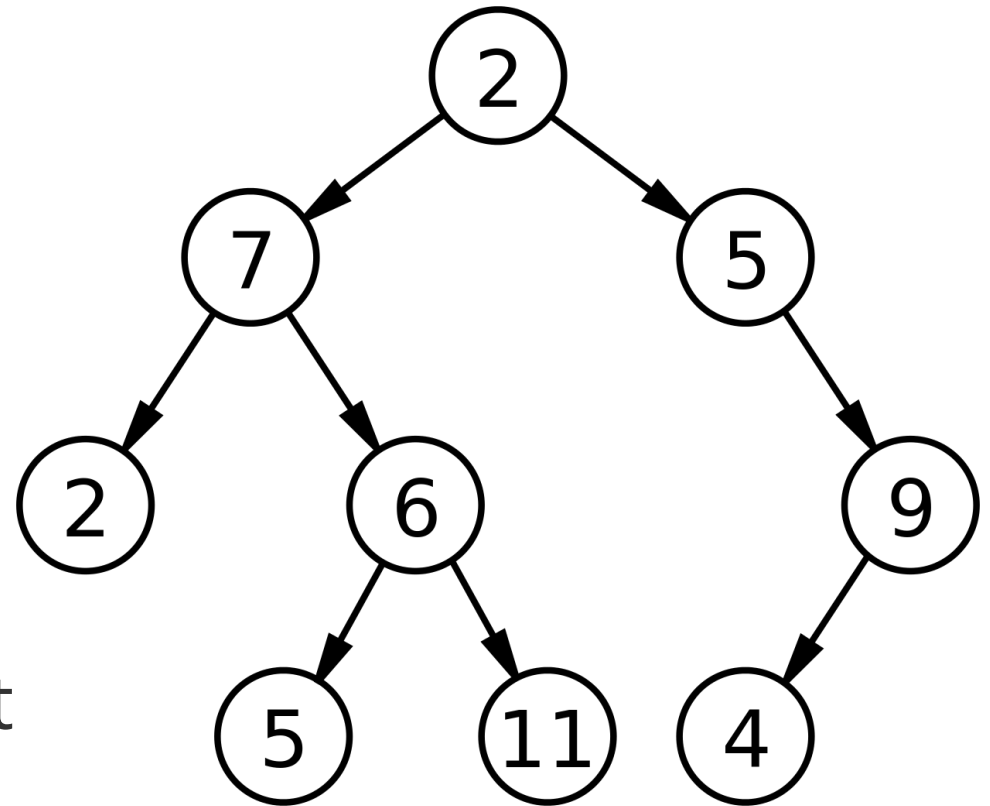


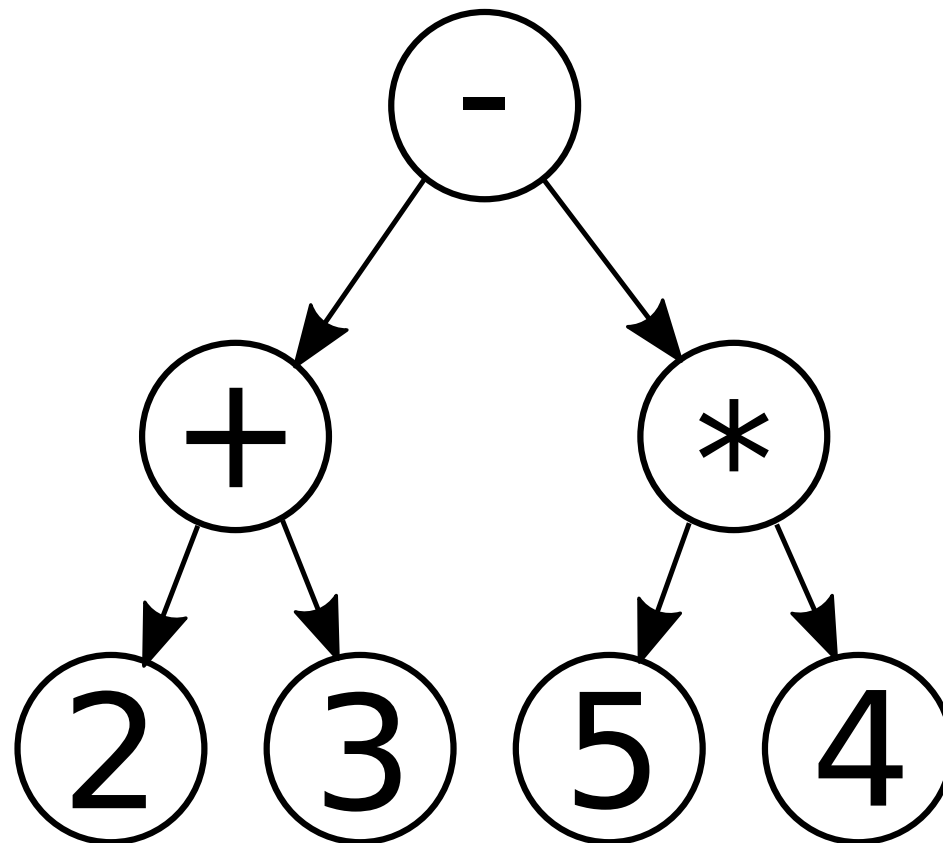
# Arbre

- **Arbre** : ensemble de **sommets** contenant des **données** avec une structure hiérarchique
- Un sommet a  $\leq 1$  parent et des enfants
- Feuille : sommet sans enfant
- Racine : sommet sans parent



# Quelques exemples

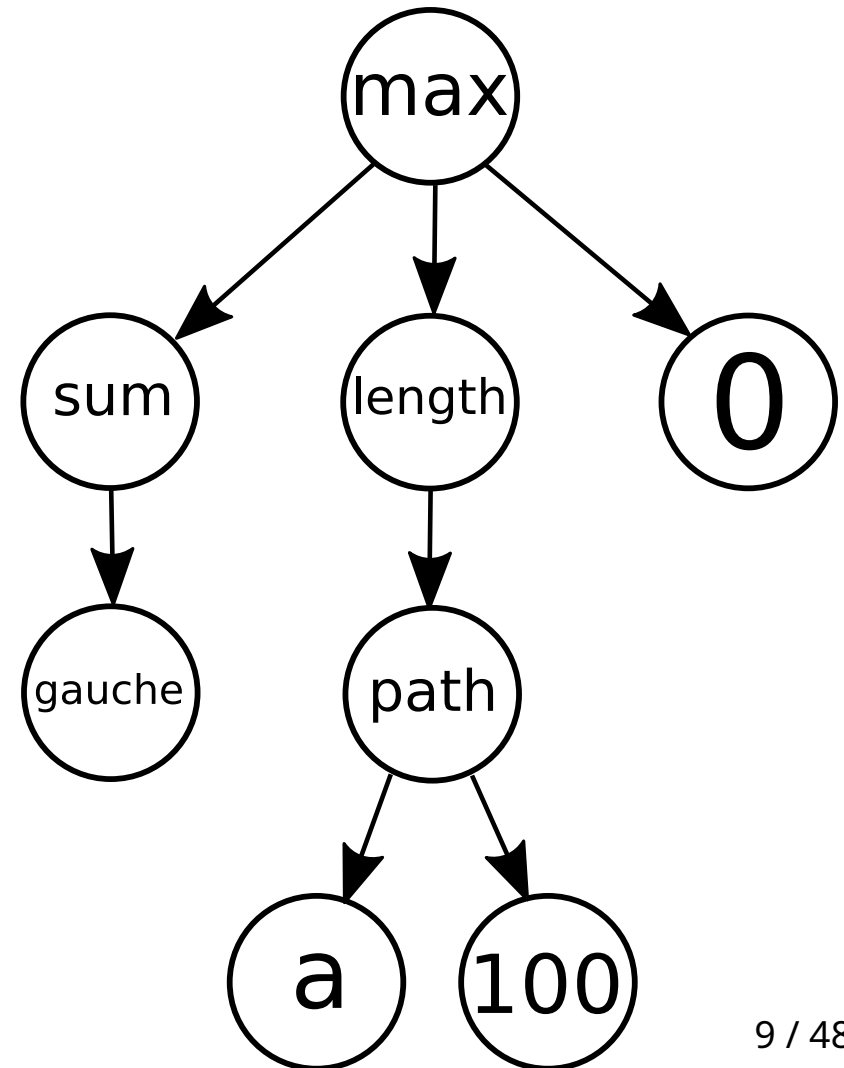
- Une expression arithmétique  $(2+3) - (5*4)$   
ou logique  $(a \text{ and } b) \text{ or } c$



# Quelques exemples

```
int x = Math.max(sum(a.gauche), length(path(a, 100)), 0)
```

- Du code source,  
un fichier UML / HTML...



# Comment ça s'utilise

- Un arbre est une classe abstraite ; on peut l'implémenter de différentes manières.
- **Création** d'un nouvel arbre (à un seul sommet) :  
`Tree<String> A = new Tree<>("Salut");`
- **Ajout** d'un nouvel enfant à la racine :  
`A.addchild(new Tree<>("World"));`

# Comment ça s'utilise

- **Récupérer** les données / enfants / parents de la racine :

`A.data(); A.child(2); A.parent();`

- **Modifier** ces mêmes valeurs :

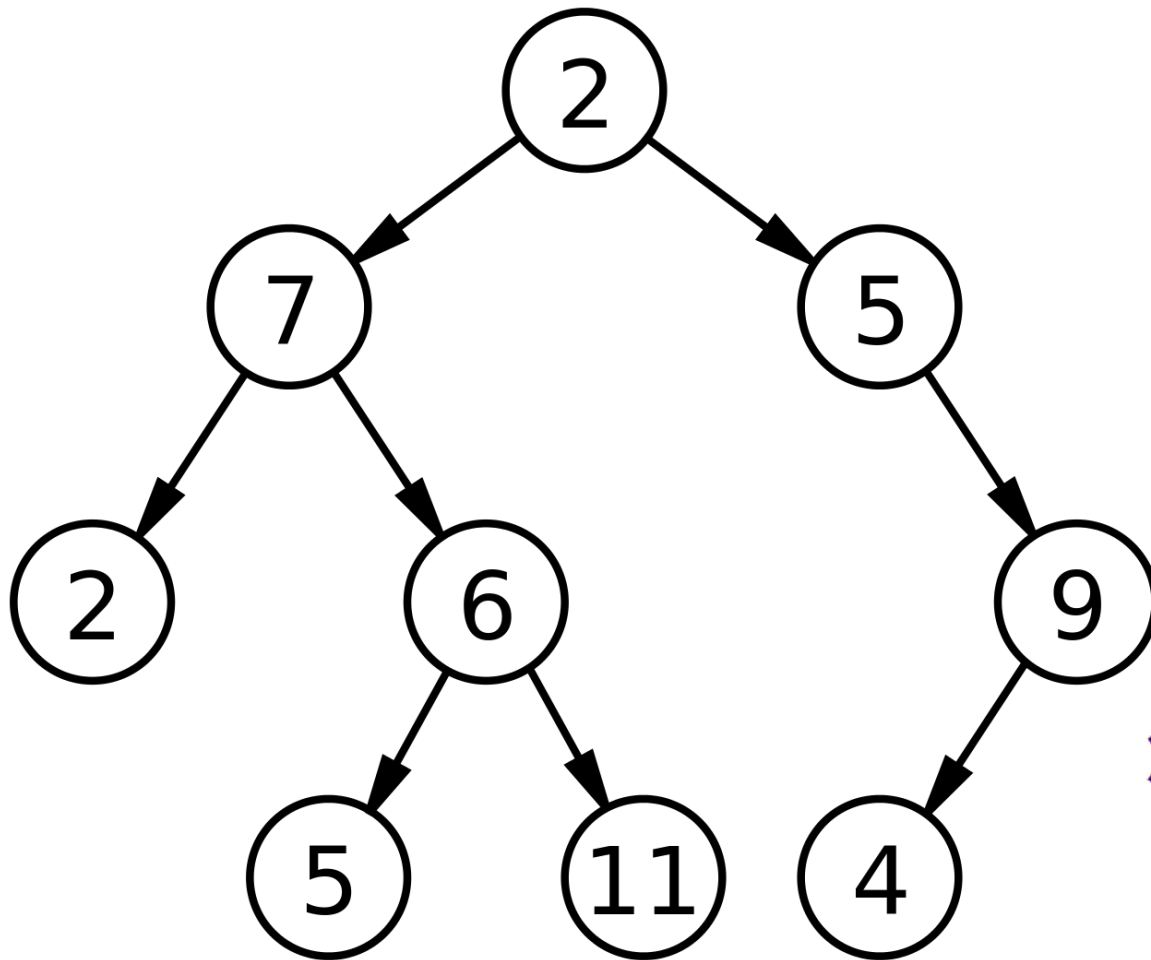
`A.setdata(5); A.setchild(2, T);`

- Si les valeurs ne sont pas renseignées, la réponse sera une valeur `null`

+ les fonctions que vous ajouterez !

# Exemple de définition

```
Tree<Integer> a = new Tree<>(2,  
    new Tree<>(7,  
        new Tree<>(2),  
        new Tree<>(6,  
            new Tree<>(5),  
            new Tree<>(11)  
        )  
    ),  
    new Tree<>(5,  
        new Tree(9,  
            new Tree(4)  
        )  
    )  
);
```



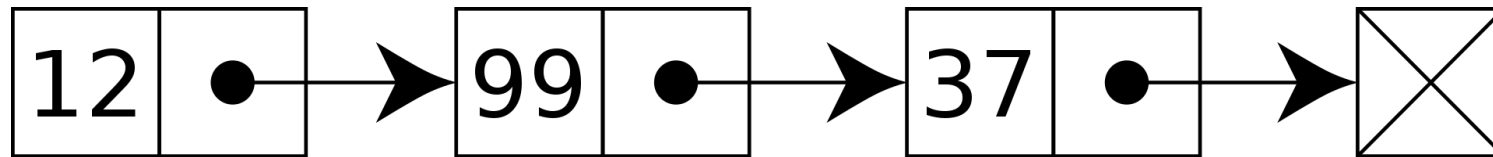
# Cas particulier : arbre binaire

- Jamais plus de deux enfants :  
on parle d'**enfant gauche** et d'**enfant droit**

`A.left()`      `A.right()`

- L'un ou l'autre peut être `null`

# Cas particulier : liste



- **Liste (chaînée)** : arbre avec un seul enfant à chaque pas
- **Tête** : équivalent de data
- **Queue** : l'enfant (liste privée du premier élément)  
`A.tail()`





# Usage des arbres

- Un tableau, une liste, une chaîne de caractères, un fichier texte sont des structures **linéaires**
- Ces fichiers représentent souvent des objets qui ont une structure **arborescente**
- **Parser**, c'est construire l'**arbre syntaxique** pour pouvoir calculer, compiler...



# Algorithmes récurifs

# Algorithmes et structures récursives

- On peut faire des algorithmes impératifs (« normaux ») mais ça se complique vite
- Nouvel outil : **algorithmes récursifs** !
- Pour calculer la fonction  $f$ ,
  1. On appelle  $f$  sur les descendants
  2. On récupère ces résultats
  3. On utilise ces résultats pour calculer notre résultat

Variations suivant le type de problème

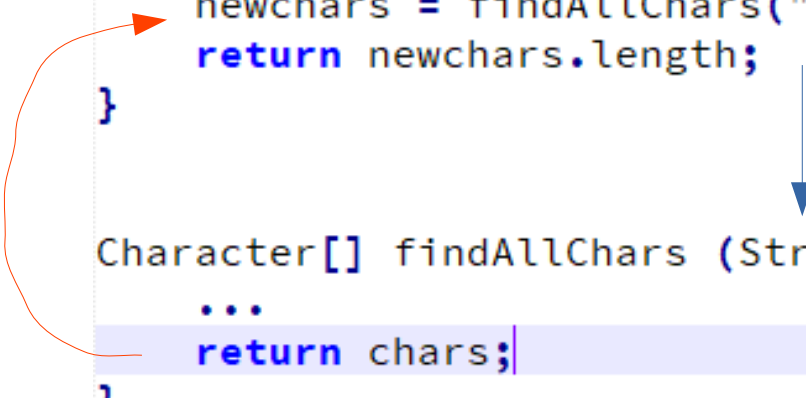


# Fonction qui renvoie une valeur

Exemple : compter la somme des éléments d'un arbre binaire.

Démo : Sum.java

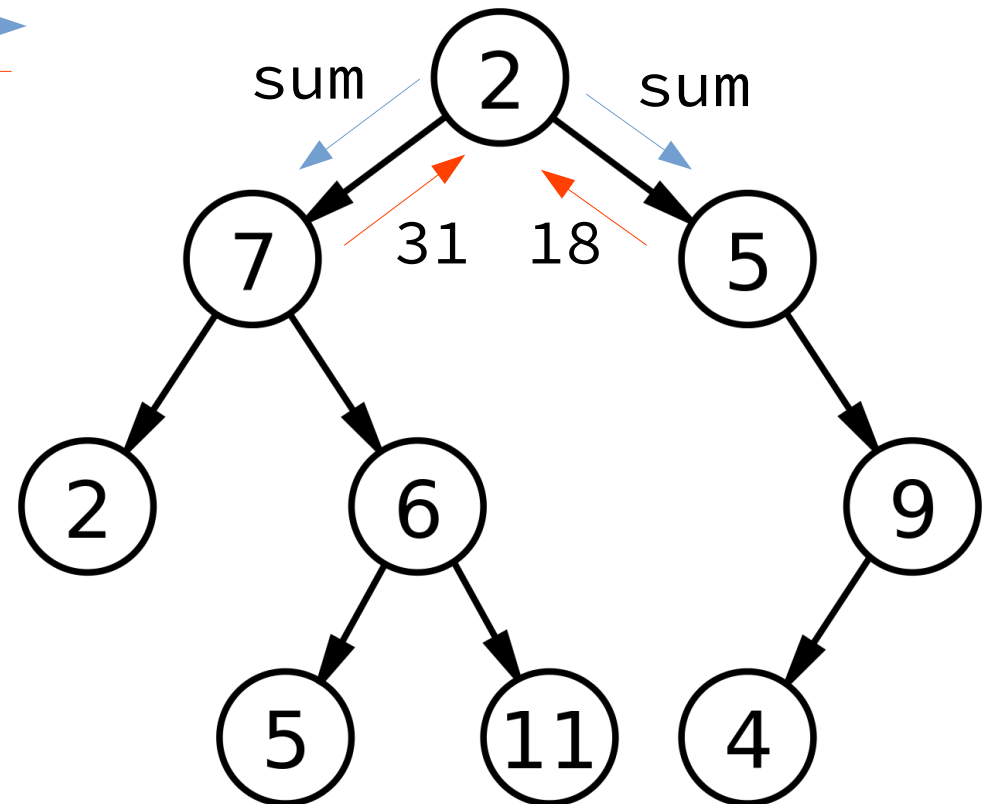
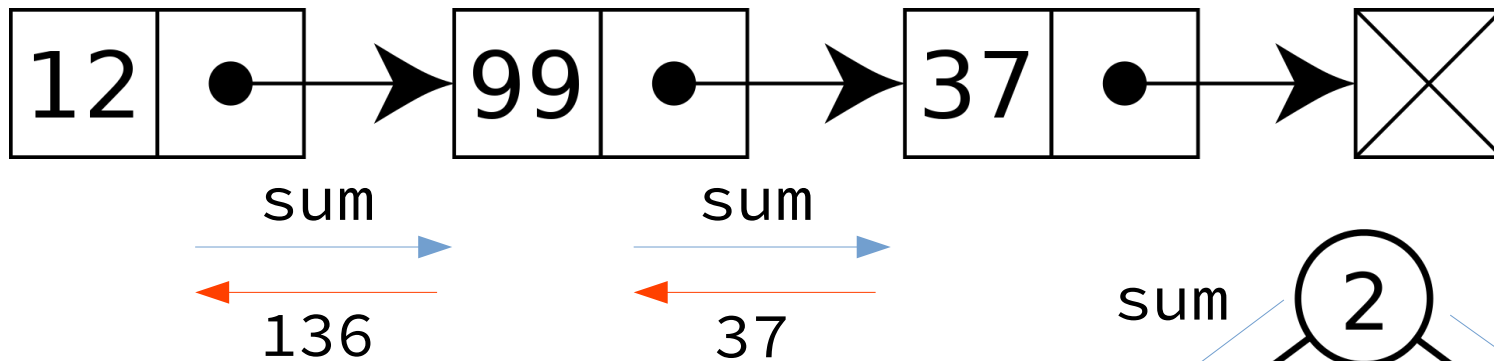
# Appels de fonction



```
int nbNewChars (int interval){  
    Character[] newchars;  
    newchars = findAllChars("time_since_creation", interval);  
    return newchars.length;  
}  
  
Character[] findAllChars (String criterion, int value){  
    ...  
    return chars;  
}
```

- Passage d'infos par **arguments** et **return**
- On attend le résultat (newchars =)
- **Pile** d'exécution

# Visualisation



# Fonction qui renvoie une valeur

- Premier essai :

```
public static int sum(Tree<Integer> a){  
    int result = a.data() + sum(a.left()) +  
    sum(a.right());  
    return result;  
}
```

- Erreur si le fils gauche ou droit n'existe pas !
- → il faut traiter le cas particulier où l'arbre est null



# Créer des structures récursives

- Même idée : utiliser les appels récursifs, en pensant à l'ordre de ce qu'on fait
- Exemple : ajouter un élément à la fin d'une liste

Démo : `append.java`



# Résumé : algos arbres et listes

- Pour calculer ou modifier : approche **récursive** (rappel de la fonction sur les descendants)
- « Faire confiance » à la fonction appelée pour faire le boulot
- Récupérer le résultat si nécessaire
- Gérer le **cas de base** (« objet vide » ; mot-clé **null**)



# **Implémenter les structures récurives**

# Structures récursives

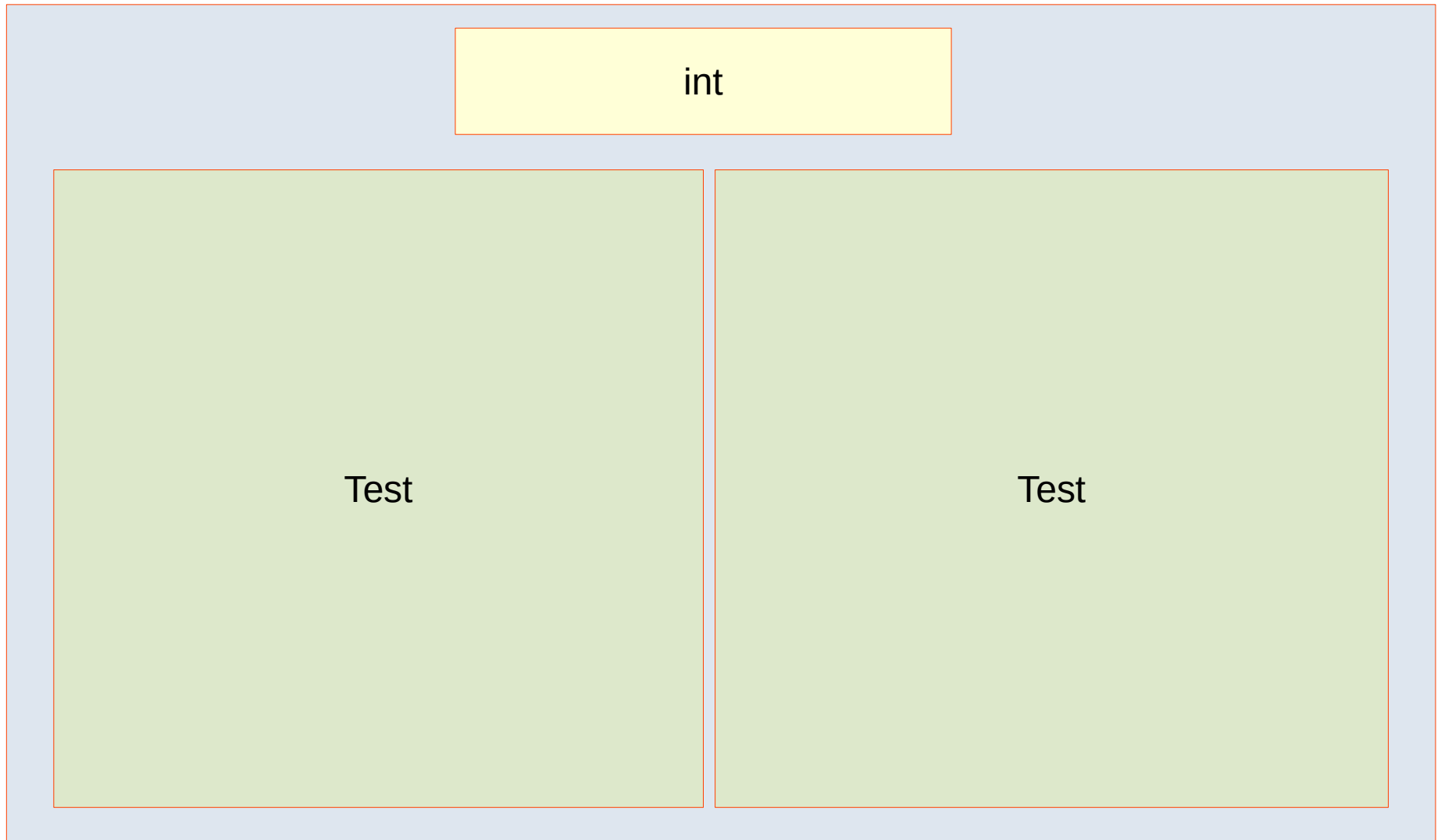
- Structure de données + récursion =

Un objet  
dont les membres  
sont du même type  
que lui-même !

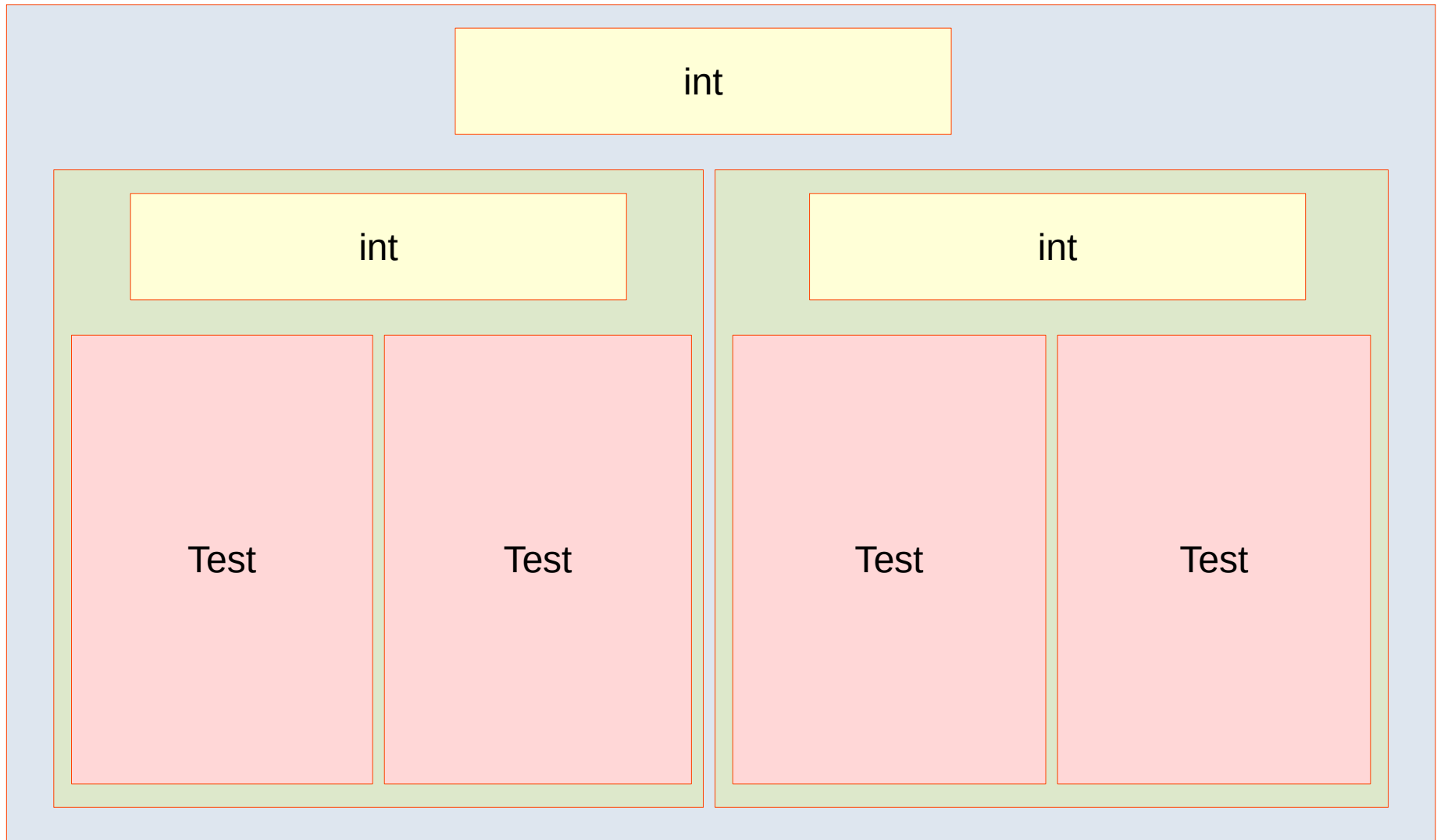
```
public class Test{  
    int etiquette;  
    Test gauche;  
    Test droite;  
}
```

- Ici : un arbre binaire sur des entiers  
(comment faire pour définir le premier arbre  
binaire?)

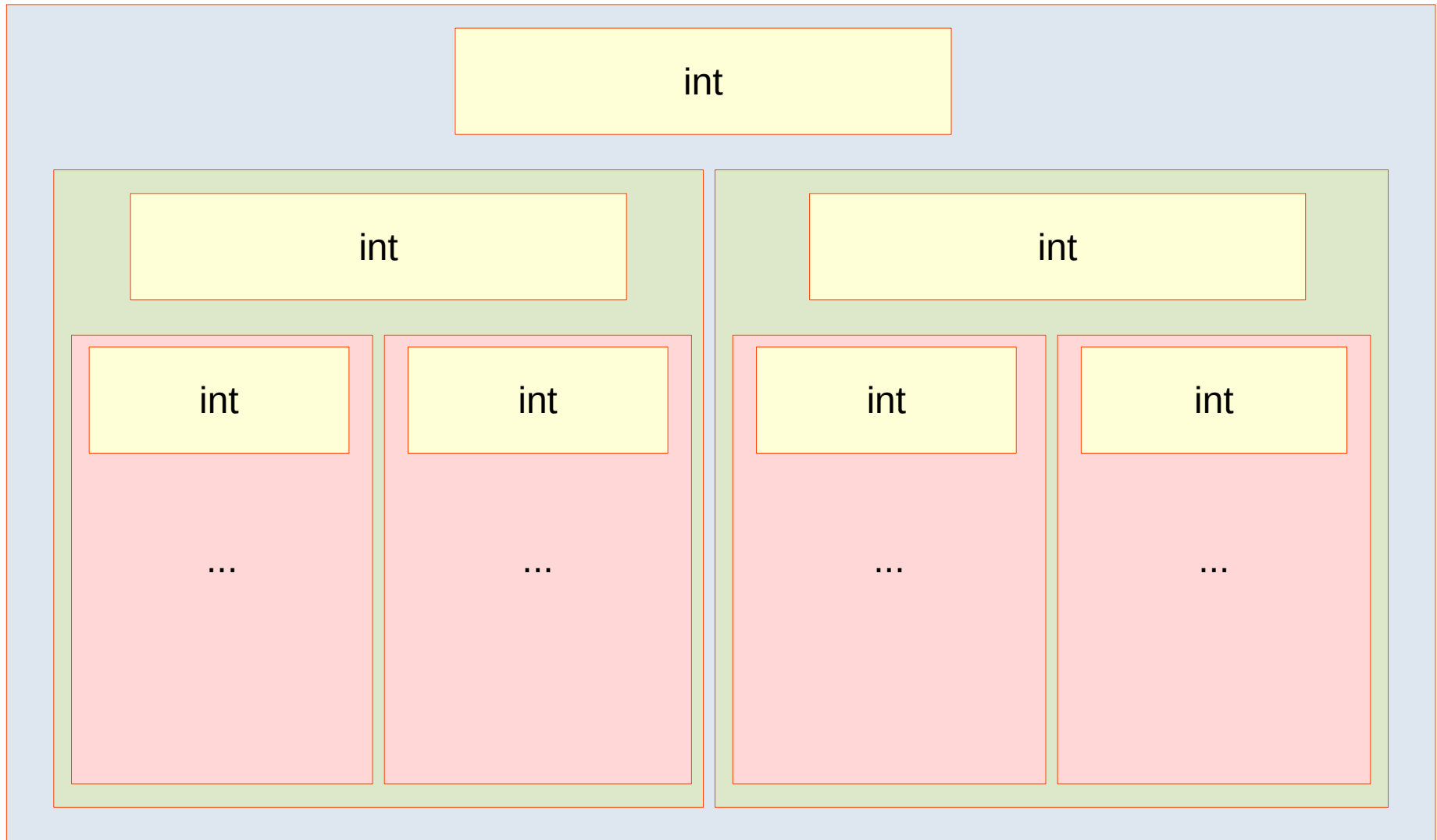
# Structures récursives



# Structures récursives



# Structures récursives





# Rappels d'orienté objet

## Getters / Setters

- On fournit une interface (`public`) aux utilisateurs
- L'implémentation est cachée (`private`) et peut changer ; l'utilisateur ne s'en soucie pas
- Pour un arbre binaire :  
`left()`, `right()`, `setLeft()`, `setRight()`

# Détails de l'implémentation

- Les arbres ont souvent plus de deux descendants.
- Implémentation par liste ou tableau de descendants, ou comme vous voulez (private)

`Liste<Arbre<Type>>` ou `Arbre<Type> []`

- `interface` : sert à voir les signatures des fonctions sans lire tout le code
- `extends` : `BTree` est un type particulier de `Tree`



# Types génériques

- On peut définir la structure « tableau » avec différents types : `int[]`, `String[]`, `bool[]`...
- Faisons la même chose avec les **types génériques**

```
public class Tree<T> implements TreeI<T>{  
    private T data;
```

- T sera une classe (Integer, String, Boolean...)

# Types génériques

- Pour faire court, on peut écrire :

```
BTree<Integer> ABR = new BTree<>(data:5,  
                                new BTree<>(data:3,
```

et l'exécuteur devine quel type d'arbre est construit.

# Exceptions

- Une **exception** est un signal envoyé par le programme disant qu'il est dans une situation ou il faut arrêter l'exécution
- Exemples
  - `IllegalArgumentException`
  - `NullPointerException`
  - `ArrayIndexOutOfBoundsException`
- Le programme entier crashe et donne des infos.

# Lancer une exception

- Pour lancer une exception :

```
public T get(int i) throws IllegalArgumentException{  
    if (i < 0){  
        throw new IllegalArgumentException();  
    }  
    if (i == 0){
```

- throws : pour dire au programmeur quelle exception attendre (sans lire le code)
- On peut même définir ses propres exceptions !

# Attraper une exception

```
public Tree<T> child(int n) {  
    try{  
        return this.children().get(n);  
    }  
    catch (IndexOutOfBoundsException e){  
        return null;  
    }  
}
```

- On essaie de faire ce qu'on veut (try) et on attrape l'exception (catch) si on ne veut pas que le programme s'arrête
- L'objet e contient des infos sur l'erreur (e.getMessage(...))



# Méthodes algorithmiques



# Problèmes d'optimisation

Le type de problèmes qu'on va regarder :

- On cherche une **solution** à un **problème**
- qui doit respecter certaines règles, et
- qui est la meilleure possible sur un certain critère (critère d'optimisation)

# Exemple 1 : rendu de monnaie

- Vous devez donner 226€ et vous avez à disposition des pièces : 5€, 2€, 1€ (à volonté). Quelles pièces choisir pour rendre le moins de pièces possibles ?
- **Règle** : la somme des pièces fait la somme demandée
- **Critère d'optimisation** : le minimum de pièces



# Algorithmes gloutons

- Approche **gloutonne** (*greedy* en anglais) :
  - Je me fixe un critère de choix simple (**critère glouton**)
  - Dès que j'ai un choix à faire, je suis mon critère
- « glouton » parce qu'on fait le choix qui a l'air bien tout de suite, sans penser au long terme

# Exemple 1 : rendu de monnaie

- Vous devez donner 226€ et vous avez à disposition des pièces : 5€, 2€, 1€ (à volonté). Quelles pièces choisir ?
- **Critère glouton** (par exemple) : toujours rendre la plus grande pièce possible

Démo : Rendu.java

# Réfléchir / améliorer l'algo

- Peut-on faire plus efficace ?
- L'approche gloutonne est-elle **optimale** ?  
Ex : 10€ avec des pièces de 4€, 3€, 1€
- Que faire si le rendu est **impossible** ?  
Ex : 7€ avec des pièces de 5€ et de 3€
- En général : un algorithme glouton est **rapide** mais **pas optimal**. Ses performances dépendent beaucoup du critère. Il faut tester !



# Force brute



# Force brute

- Idée générale: il faut essayer **tous** les choix et garder le meilleur
- On a toujours la réponse **optimale** mais le programme est plus **lent**.

# Rendu de monnaie : rappel

- On a des pièces (disons 4€, 3€, 1€) et on doit rendre une certaine somme.
- **Approche force brute** : essayer toutes les possibilités

Sur 227€ : comparer le nombre de pièces sur 223€, 224€ et 226€.

# Rendu de monnaie : rappel

- On a des pièces (disons 4€, 3€, 1€) et on doit rendre une certaine somme.

$N(k)$  : pièces nécessaires pour  $k$ €.

- **Glouton (pièce la plus grande)**

$N(10) \rightarrow$  renvoyer  $[5] + N(6)$

- **Force brute :**

$N(10) \rightarrow$  essayer  $N(6)$ ,  $N(7)$  et  $N(9)$ .

renvoyer la liste la plus courte entre  
 $[1] + N(9)$ ,  $[3] + N(7)$ ,  $[4] + N(6)$

## Exemple 2 : voyageur (projet S3)

- On est en  $(0,0)$  et on a une liste de villes à visiter  $[(3,5), (10,-1), (4,6), (3,7), (-4,1)]$
- Trouver un ordre pour **faire le moins de distance possible**



# Exemple 2 : voyageur (projet S3)

- Problème connu pour ne **pas** avoir d'algorithme rapide et optimale
- Un algorithme glouton n'est pas optimal, mais peut être suffisant
- Idées : commencer par la ville la plus proche, d'est en ouest...
- Pour une vraie application : essayer plusieurs critères et **tester**

# Résumé sur les méthodes

- Problèmes où vous avez des **choix à faire** et l'utilisateur peut attendre des réponses **plus ou moins bonnes**
- Approche **gloutonne** : fixe un critère de choix. Rapide mais la qualité des réponses varie.
- Approche **force brute** : essayer toutes les possibilités. Lent mais toujours optimal.
- Pas forcément récursif !