Structures de données et récursivité

Cours 4 / GL / LP DA2I

Cédric Lhoussaine

2019-2020

Types "primitifs"

Les classes "wrapper"

À tout type primitif (`int`, `char`...) est associée une classe "wrapper" qui a pour seule fonction d'encapsuler une valeur:

- `Integer` pour `int`
- `Character` pour `char`
- `Boolean` pour `boolean` etc. . .

  Pas de conversion entre type primitifs et "wrapper"

  → passage par constructeur/méthode:

```
Integer i = new Integer(5);
Character c = new Character('A');
int j = i.intValue();
char d = c.charValue();
```

Autoboxing/unboxing

Depuis Java 1.5: transformation automatique des types primitifs en leurs classes "wrapper" (`Integer`, `Character`...) et *vice-versa*.

```
Integer a, b ;
int s ;
a = 5 ; // au lieu de a = Integer.valueOf(5) ; => boxing
b = 3 ; // boxing
s = a + b ; // unboxing
// au lieu de : s = a.intValue() + b.intValue() ;
```

Autoboxing/unboxing: dangers!

Le compilateur se contente de transformer certaines écritures :

- autorisés

```
Integer i = 3;
// -> Integer i = Integer.valueOf(3);

int j = i;
// -> int j = i.intValue();
```

Autoboxing/unboxing: dangers!

Le compilateur se contente de transformer certaines écritures :

- autorisés

```
Integer i = 3;
// -> Integer i = Integer.valueOf(3);

int j = i;
// -> int j = i.intValue();
```

- interdit

```
j.intValue();
"int cannot be dereferenced"
```

→ On ne peut pas appeler de méthode sur une expression ou une variable de type primitif!

Autoboxing/unboxing: dangers!

Le compilateur se contente de transformer certaines écritures :

- autorisés

```
Integer i = 3;
// -> Integer i = Integer.valueOf(3);

int j = i;
// -> int j = i.intValue();
```

- interdit

```
j.intValue();
"int cannot be dereferenced"
```

→ On ne peut pas appeler de méthode sur une expression ou une variable de type primitif!

- autorisé

```
((Integer) j).intValue();
// -> (Integer.valueOf(j)).intValue();
```

Égalité entre objets

Exercice

Qu'affiche le programme suivant ?

```
Date d1, d2, d3, d4;
d1 = new Date(8,4,2006);
d2 = d1 ;
d3 = new Date(8,4,2006) ;
d4 = new Date(9,4,2006) ;
System.out.println(d1 == d2) ;
System.out.println(d1 == d4) ;
System.out.println(d1 == d3) ;
```

Égalité physique

- Rappel: Les variables affectées à des objets contiennent en réalité des références sur ces objets.
- L'opérateur == teste l'égalité de deux références donc permet de dire si deux variables désignent une même instance.

  == *teste l'égalité physique*

Égalité physique

- Rappel: Les variables affectées à des objets contiennent en réalité des références sur ces objets.
- L'opérateur == teste l'égalité de deux références donc permet de dire si deux variables désignent une même instance.

  == *teste l'égalité physique*

Par conséquent:

```
Date d1, d2, d3, d4;
d1 = new Date(8,4,2006);
d2 = d1 ;
d3 = new Date(8,4,2006) ;
d4 = new Date(9,4,2006) ;
System.out.println(d1 == d2) ; // true
System.out.println(d1 == d4) ; // false
System.out.println(d1 == d3) ; // false !
```

La référence `null`

- si une variable d'un type objet ne contient aucune instance, sa valeur est `null`.

```
Date d ;
```

  d contient `null`

- pour savoir si une variable contient une instance ou non:

```
v == null
```

- si une méthode ne peut retourner aucun objet valide:

```
return null;
```

La référence `null`

Appel de méthode impossible sur une expression null

```
System.out.println(d);
```

affiche `null`

```
System.out.println(d.toString());
// Exception in thread "main" java.lang.NullPointerException
```

Égalité logique

- deux instances peuvent représenter des "choses" identiques ou de "même valeur"
- il faut donc pouvoir déterminer si deux objets peuvent être considérés comme égaux (ou interchangeables)
- attention, question de point de vue !
- en Java, on écrit une méthode dédiée :

```
public boolean equals(Object o)
```

`equals` teste l'égalité logique

Égalité logique (exemple)

```
Date d1, d2, d3, d4;
d1 = new Date(8,4,2006) ;
d2 = d1 ;
d3 = new Date(8,4,2006) ;
d4 = new Date(9,4,2006) ;
System.out.println(d1.equals(d2)) ; // true
System.out.println(d1.equals(d4)) ; // false
System.out.println(d1.equals(d3)) ; // true !
```

Les listes

Définition de liste

Une liste est un ensemble de données

- de même type
- en nombre quelconque
- avec accès séquentiel depuis la tête de la liste

Exemple

tête $\rightarrow \boxed{1} \rightarrow \boxed{1} \rightarrow \boxed{2} \rightarrow \boxed{3} \rightarrow \boxed{5} \rightarrow \boxed{8} \rightarrow \boxed{13}$

On ne peut accéder à $\boxed{5}$ qu'après avoir parcouru les élèments précédents depuis la tête.

Les listes dans le paradigme objet

- Quelle est l'entité permettant de construire une liste ?
- Quels sont ses attributs ?
- Comment savoir si on est à la fin ?
- Comment parcourir une liste ?
- Comment modifier une liste ?
- ...

Modélisation

ElementListe
`contenu: int`

`estDernier(): boolean`
`elementSuivant():`
`ElementListe suivant`
`contient(valeur v): boolean`
`placerALaFin(valeur: int)`
`placerEnOrdre(valeur: int)`

Implantation

```
public class ElementListe {
    private int contenu ;
    // la valeur stockée
    // référence sur l'élément suivant
    private ElementListe suivant = null ;

    public ElementListe(int valeur) { contenu = valeur ;}
    public boolean estDernier() { return (suivant == null);}
    public boolean contient(int v) { ... }
    public int contenu() { ... }
    public ElementListe elementSuivant() { ... }
    public void placerALaFin(int valeur) { ... }
    public String toString() { ... }
}
```

Récursivité

Définition de la récursivité

On parle de récursivité lorsque la définition d'un traitement ou d'une structure fait appel à ce traitement ou cette structure.

Exemple

```
factorielle(n) =
    si n <= 1 retourner 1
    sinon retourner n * factorielle(n-1)
```

Algorithme itératif (ou séquentiel)

```
public void placerALaFin(int v)
{
    // on place dans l'élément courant (tête)
    ElementListe l = this ;
    while (l.suivant != null)
        l = l.suivant ; // et on va jusqu'au dernier
    // enfin on rajoute un nouvel élément de liste
    l.suivant = new ElementListe(v) ;
}
```

Algorithme itératif (ou séquentiel)

```java
public void placerALaFin(int v)
{
    // on place dans l'élément courant (tête)
    ElementListe l = this ;
    while (l.suivant != null)
        l = l.suivant ; // et on va jusqu'au dernier
    // enfin on rajoute un nouvel élément de liste
    l.suivant = new ElementListe(v) ;
}
```

Algo séquentiel pour effectuer un traitement sur chaque élément :

1. élément courant ← tête de liste
2. FAIRE
   2.1 effectuer `t()` sur l'élément courant
   2.2 remplacer l'élément courant par le suivant
3. JUSQU'À élément courant `null`

→ Dans la version itérative de `placerALaFin` un élément particulier "examine" tous les autres!

Algorithme récursif

Algorithme récursif pour effectuer un traitement `t()` sur chaque élément. On réécrit le traitement t() sous la forme:

- d'une (ou +) condition d'arrêt, par exemple:
  `SI suivant null FAIRE...`
- d'un (ou +) appel récursif: demander au suivant d'effectuer t()!

Algorithme récursif

Algorithme récursif pour effectuer un traitement `t()` sur chaque élément. On réécrit le traitement t() sous la forme:

- d'une (ou +) condition d'arrêt, par exemple:
  `SI suivant null FAIRE...`
- d'un (ou +) appel récursif: demander au suivant d'effectuer t()!

```
public void placerALaFin(int v)
{
    // si on n'est pas sur le dernier
    if (suivant != null)
        // alors on demande au suivant de faire le travail
        suivant.placerALaFin(v) ;
    else // sinon on fait l'ajout
        suivant = new ElementListe(v) ;
}
```

→ chaque élément travaille localement.

Remarque

Sur des structures de données récursives on effectue des traitements récursifs!

Types génériques

Depuis Java 1.5...

Augmenter la réutilisation des classes:

- les algorithmes de manipulation de liste ne dépendent pas du type de contenu
- mais il faut bien spécifier un type pour l'attribut contenu
- utiliser `Object` serait trop imprécis et imposerait des conversions

$\rightarrow$ il faut juste pouvoir spécifier que dans toutes les instances d'`ElementListe`, l'attribut `contenu` a un certain type X (a priori quelconque mais toujours le même)

Définition

- Type identifié par un symbole arbitraire ne se référant à aucun type existant, permettant d'imposer à une classe de travailler sur des objets d'un même type;
- On le note entre chevrons : `<T>` pour désigner le paramètre formel de la classe; et sans chevrons: `T` ailleurs...;
- On remplace `T` par le type réel lors de l'utilisation.

Exemple

```
public class ElementListe<T>{
    // T représente un type objet quelconque
    private T contenu; // la valeur stockée, du type T
    // référence sur l'élément suivant
    private ElementListe<T> suivant = null ;
    // constructeur : il a le nom de la classe
    public ElementListe(T valeur) { contenu = valeur ;}
    public boolean contient(T v) { ... }
    public T contenu() { ... }
    public ElementListe<T> elementSuivant() { ... }
    ...
}
```

Attention aux comparaisons!

```
public class ElementListe<T> {
    T contenu;
    ElementListe<T> suivant;
    // ...
    public boolean contient(T valeur) {
        if (contenu.equals(valeur)) // égalité logique
            return true ;
        if (suivant == null)
            return false ;
        return suivant.contient(valeur) ;
    }
}
public class MesListes {
    public static void main(String [] args) {
        ElementListe<String> l1 ;
        ElementListe<Integer> l2 ;
        ElementListe l3 ; // liste d'Object ;
        l1 = new ElementListe<String>("toto") ;
        l2 = new ElementListe<Integer>(3) ;
        l3 = new ElementListe("hello") ;
        String s = l1.contenu() ;
        int i = l2.contenu() ;
        Object o = l3.contenu() ; // et pas String
    }
}
```

Utilisation

- les objets retournés par les méthodes génériques sont automatiquement du "bon" type

Utilisation

- les objets retournés par les méthodes génériques sont automatiquement du "bon" type
- une classe utilisant un type générique constitue un type à part entière: `ElementListe<String>` et `ElementListe<Integer>` ne sont pas du même type!

Utilisation

- les objets retournés par les méthodes génériques sont automatiquement du "bon" type
- une classe utilisant un type générique constitue un type à part entière: `ElementListe<String>` et `ElementListe<Integer>` ne sont pas du même type!
- si l'on ne précise aucun type, `Object` est utilisé: `ElementListe` équivaut à `ElementListe<Object>`

Utilisation

- les objets retournés par les méthodes génériques sont automatiquement du "bon" type
- une classe utilisant un type générique constitue un type à part entière: `ElementListe<String>` et `ElementListe<Integer>` ne sont pas du même type!
- si l'on ne précise aucun type, `Object` est utilisé: `ElementListe` équivaut à `ElementListe<Object>`
- on peut utiliser autant de types génériques que nécessaire, par exemple : `Couple<T,V>`

Les arbres

Définition

Un arbre est une structure de données constituée de noeuds qui sont:

- une racine : nœud par lequel on accède à tous les autres
- des feuilles : nœuds qui ne permettent d'accéder à aucun autre
- des nœuds intermédiaires $\rightarrow$ eux-mêmes racines de sous-arbres

Caractéristiques

- accès séquentiel depuis la racine;
- type de données uniforme;
- nombre de nœuds quelconque. *Cas particuliers* : arbres n-aires (binaires, ternaires. . .);
- hauteur : nombre maximum de branches à parcourir de la racine aux feuilles.

Parcours d'un arbre

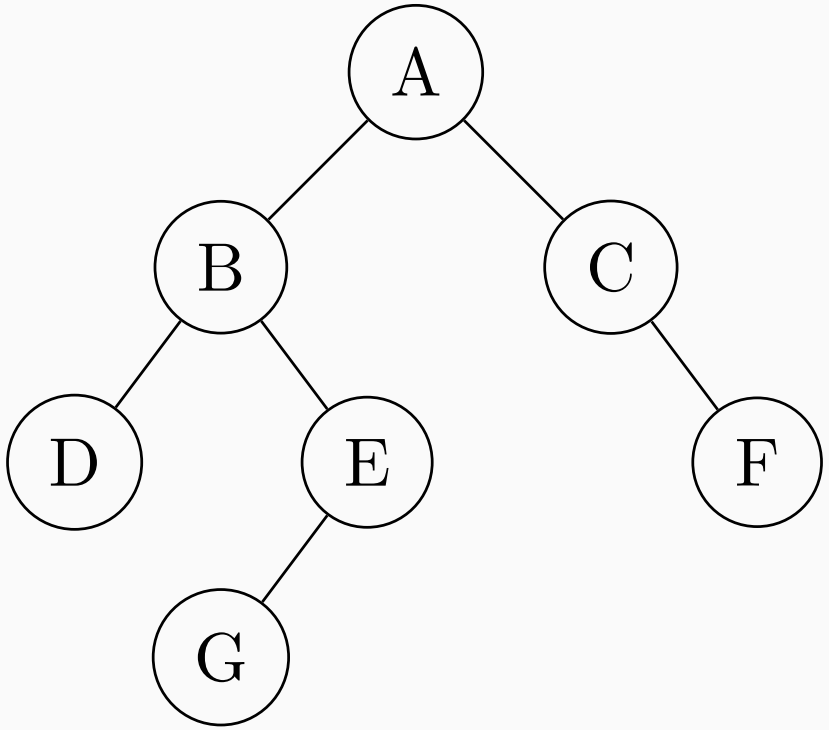
Deux classifications possibles des traitements :

- selon le niveau dans l'arborescence :
  - en largeur d'abord
  - en profondeur d'abord
- selon l'ordre des traitements
  - préfixe : nœud courant puis sous-arbres
  - postfixe : sous-arbres puis nœud courant
  - infixe : alternance sous-arbres / nœud courant

Les arbres binaires

Chaque noeud peut avoir entre 0 et 2 sous-arbres $\rightarrow$ sous-arbre gauche, sous-arbre droit.

Exemple

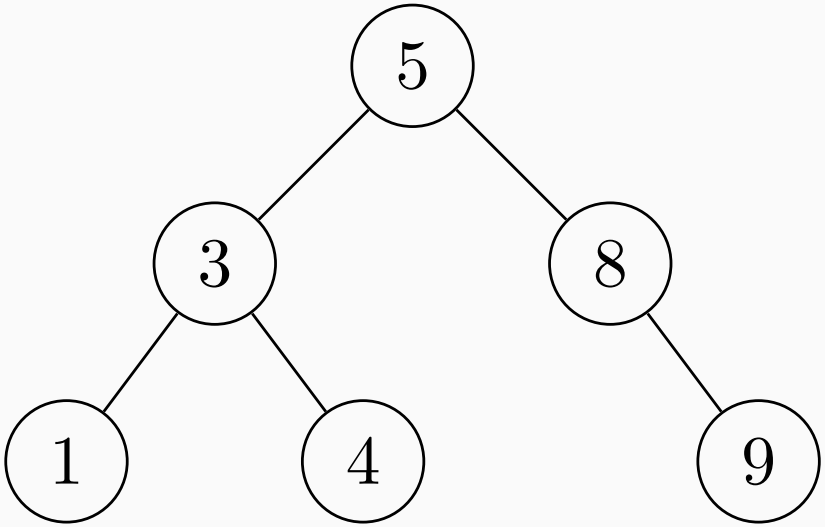


Les arbres binaires de recherche

Arbre binaire dans lequel:

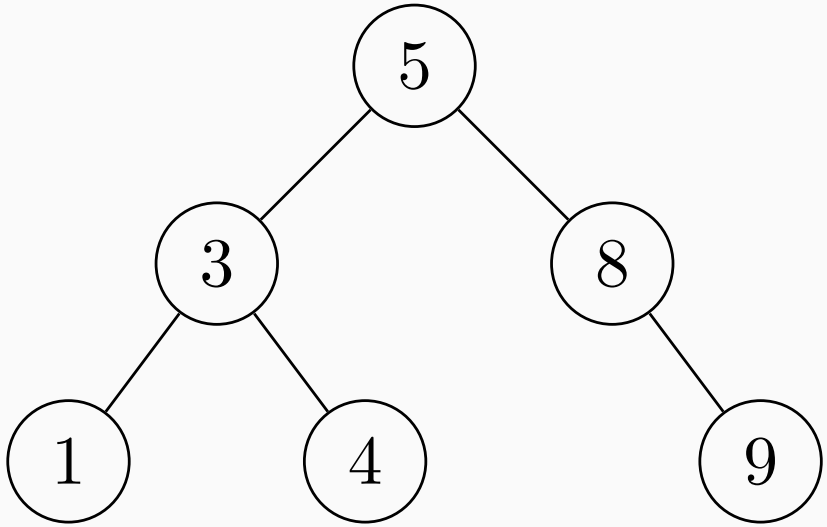
- la racine est quelconque
- les deux sous-arbres de la racine, s'ils existent, sont des arbres binaires de recherche
- tous les éléments du sous-arbre de recherche gauche sont inférieurs à la racine
- tous les éléments du sous-arbre de recherche droit sont supérieurs à la racine

Les arbres binaires de recherche (exemple)



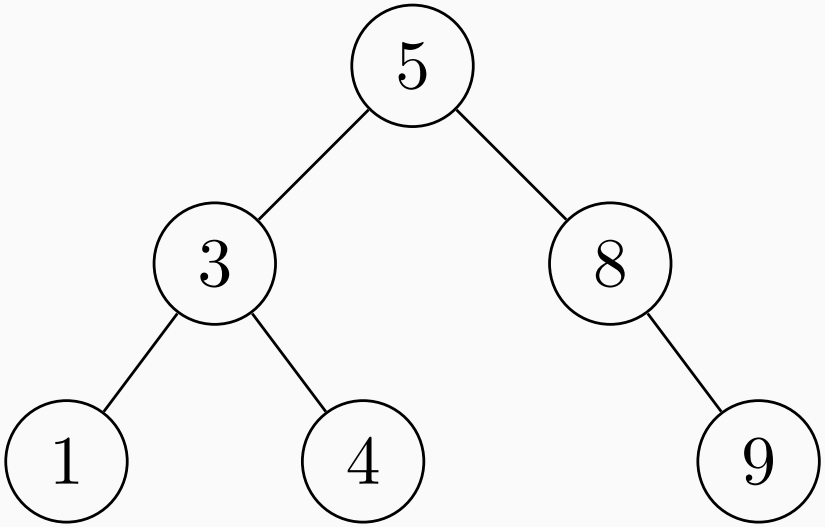
- affichage préfixe: 5 3 1 4 8 9
- affichage postfixe:

Les arbres binaires de recherche (exemple)



- affichage préfixe: 5 3 1 4 8 9
- affichage postfixe: 1 4 3 9 8 5
- affichage infixe:

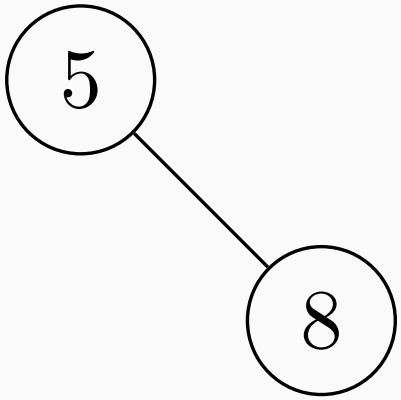
Les arbres binaires de recherche (exemple)



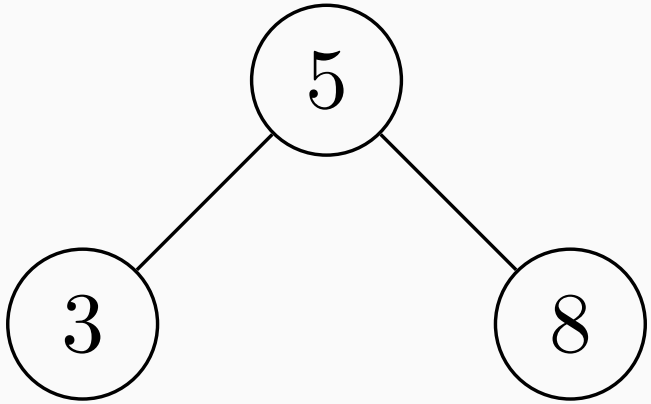
- affichage préfixe: 5 3 1 4 8 9
- affichage postfixe: 1 4 3 9 8 5
- affichage infixe: 1 3 4 5 8 9

Les arbres binaires de recherche (construction incrémentale)

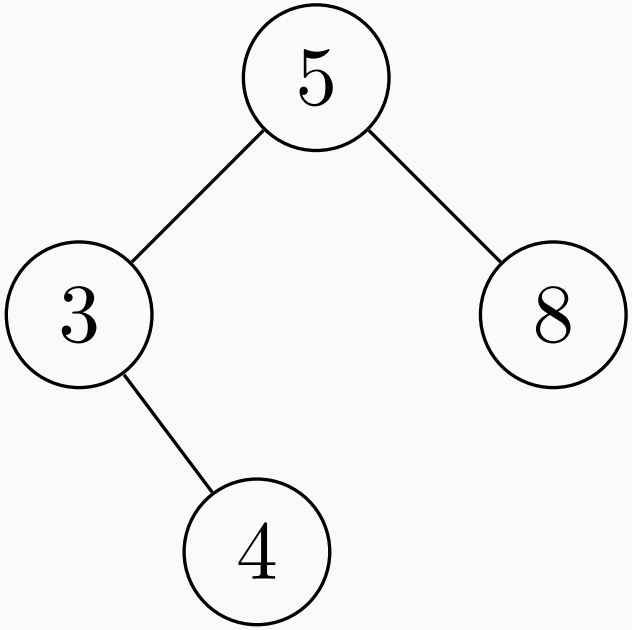
Les arbres binaires de recherche (construction incrémentale)



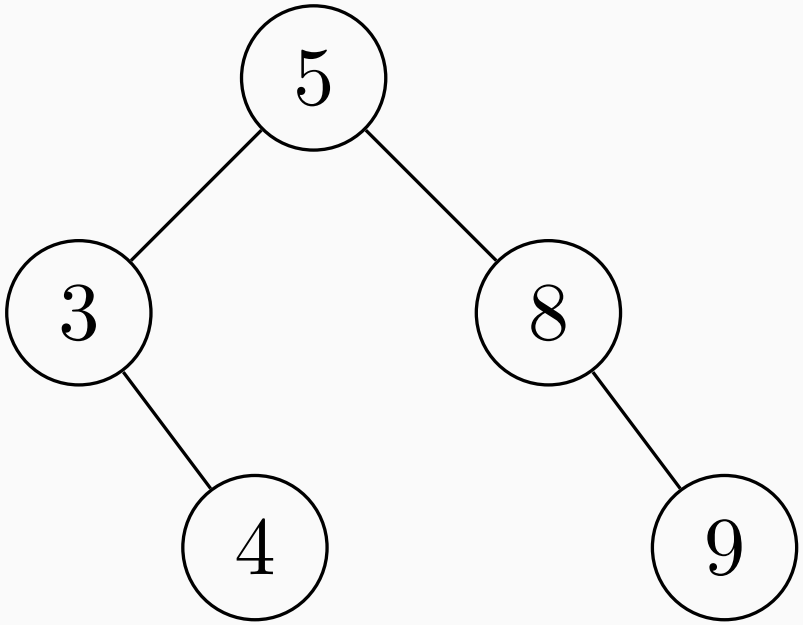
Les arbres binaires de recherche (construction incrémentale)



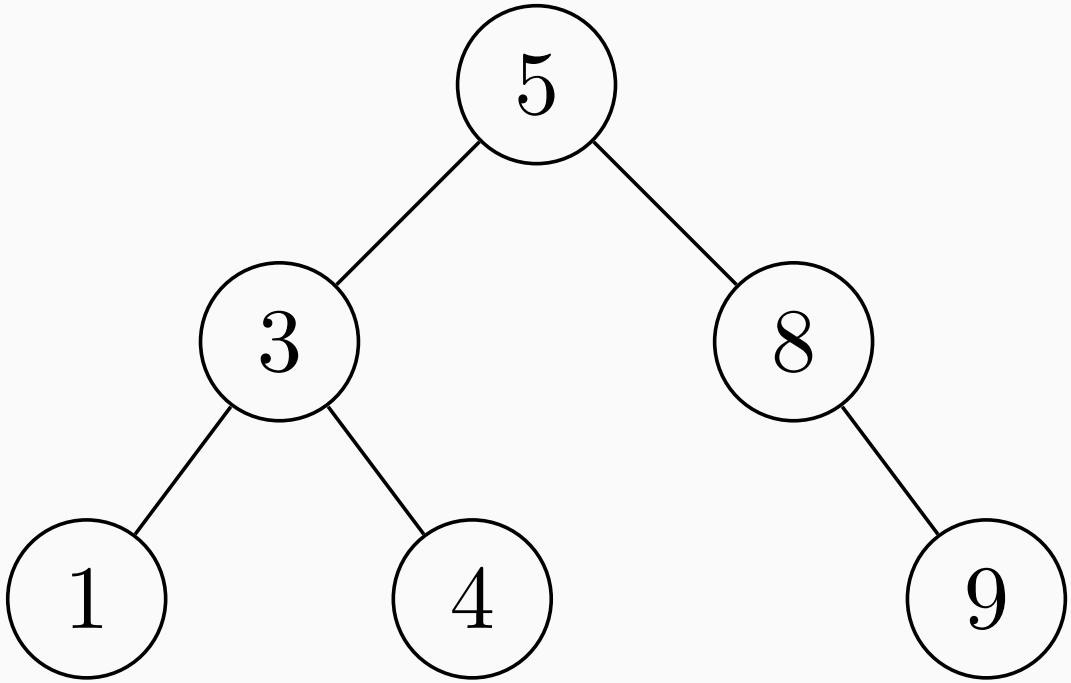
Les arbres binaires de recherche (construction incrémentale)



Les arbres binaires de recherche (construction incrémentale)



Les arbres binaires de recherche (construction incrémentale)



Intérêt des arbres

Avantages :

- rapidité d'accès $\mathcal{O}(ln(n))$ vs. $\mathcal{O}(n)$
- classification / discrimination
- nombreux algorithmes disponibles

$\rightarrow$ utilisé dans de nombreux objets Java (`TreeSet`, `TreeMap`)

Implantation en Java

```
public class ArbreBinaire {
    // le contenu du noeud
    private int contenu ;
    // les sous-arbres gauche et droit
    private ArbreBinaire gauche, droit ;
    ...
}
```

Les arbres sont des structures de données récursives, donc leurs traitements sont également récursifs!

Emacs 26.2 (Org mode 9.1.9)