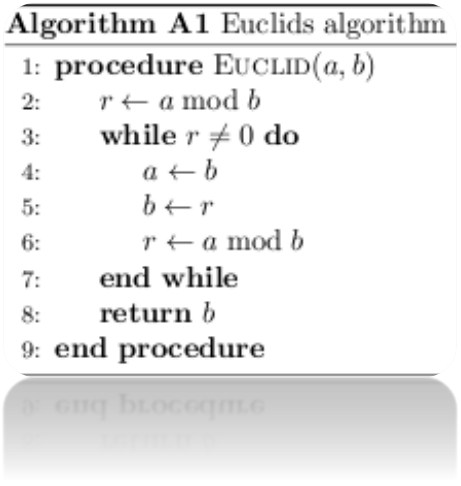
Le génie pour l'industrie

LOG100 / GTI100
Programmation en génie logiciel et des TI
Automne 2024

Cours 5 : Algorithmes & Structures de données

Chargé de cours: Anes Abdennebi

Crédits à: Ali Ouni, PhD



Algorithm A1 Euclids algorithm

```
1: procedure EUCLID(a, b)
2:     r ← a mod b
3:     while r ≠ 0 do
4:         a ← b
5:         b ← r
6:         r ← a mod b
7:     end while
8:     return b
9: end procedure
```

ALGORITHMES

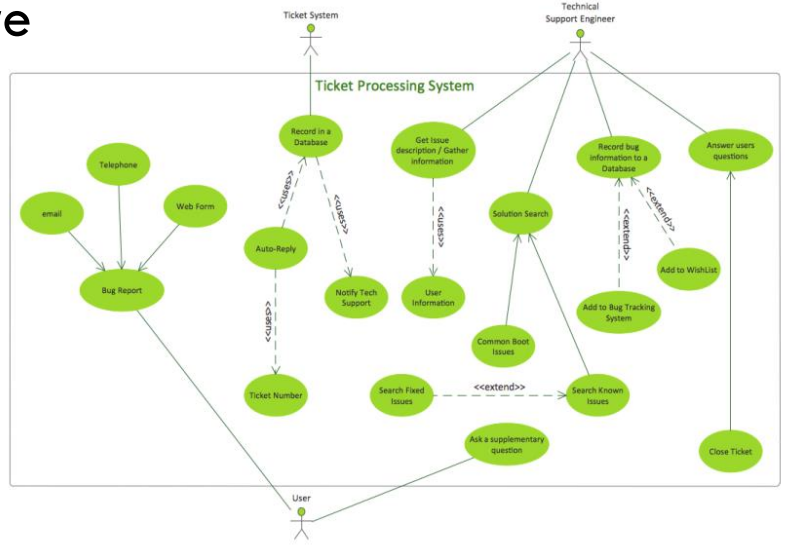
Analyse de complexité & récursivité

Algorithmes

- **ALGORITHME,** *n. m.* une séquence d'instructions (pour réaliser une « fonction »)
  - Remonte à l'Antiquité
    - Euclide : calcul du PGCD de deux nombres, etc.
  - Étymologie : al-Khwārizmī (c. 780 – c. 850)
    - *Algoritmi* en latin
    - Mathématicien perse
    - On lui doit aussi la désignation algèbre (al-jabr)

Algorithmes

- **Spécification** – ce que fait l'algorithme
  - Cahier de charges du problème à résoudre
- **Expression** – comment il le fait
  - Langage naturel
  - Graphiquement
  - Pseudo-code
  - …
- **Implémentation** – traduction de l'algorithme
  - Exprimé dans un langage de programmation réel



Algorithmes (Structure)

Entête ➔ **Algorithme** *NomAlgorithm*

Partie de déclaration ➔ **Déclarations** *Constantes, Variables*

Le Corps ➔

Début

Instruction 1;

Instruction 2;

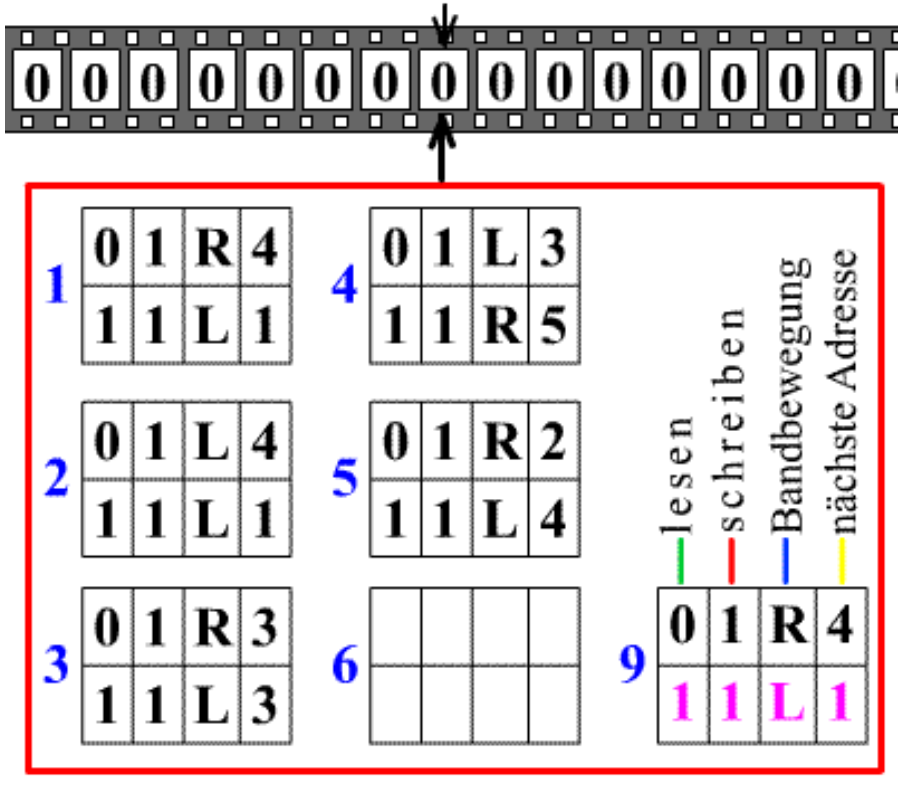
Instructions N;

Fin

Algorithmes : indécidabilité

*On ne peut pas résoudre **tous** les problèmes avec des algorithmes*

C'est évident pour certains problèmes…

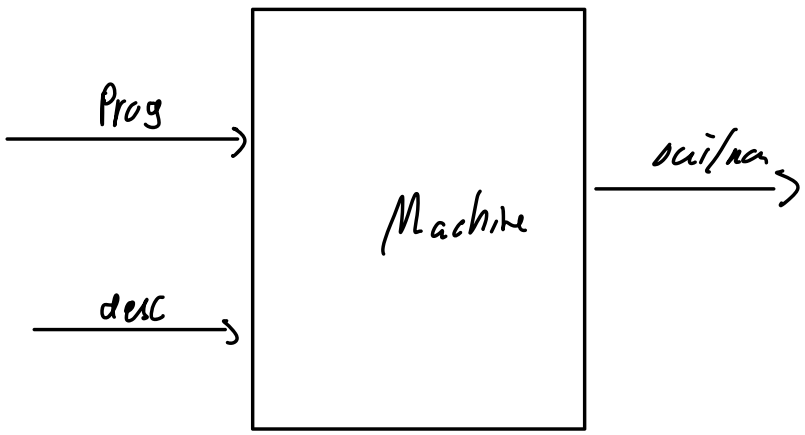


Ça l'est moins pour d'autres
(et demande des preuves formelles)

Indécidabilité – Un problème de décision pour lequel on sait qu'il est impossible de construire un algorithme qui mène à une réponse correcte *oui*/*non*

Le problème de l'arrêt *(halting problem)*

Étant donné un programme informatique quelconque (au sens de la machine de Turing), déterminer s'il finira par s'arrêter ou non.


Prog
desc
Machine
oui/non

Algorithmes : analyse de complexité

- Sert à évaluer le **temps de calcul** et/ou l'**espace mémoire** requis pour *dérouler* un algorithme

  Complexité (pire, moyen, meilleur)

  - On peut ainsi comparer les algorithmes
- S'exprime en fonction du **nombre de données** et de leur taille
  - S'intéresse seulement à l'ordre de grandeur
    - Pour *n* données :
      - Complexité logarithmique : O(log(n))
      - Complexité polynomiale : O(n) linéaire, $O(n^2)$ quadratique, etc.
      - Complexité exponentielle : $O(x^n)$
  - Toutes les opérations sont considérées **équivalentes**
  - En général, on ne considère que les **pires cas**
- Sert aussi à évaluer la **difficulté** d'un problème et à le classifier
  - Problèmes NP-difficiles (*NP-hard*), NP-complets (*NP-complete*), etc.

Algorithmes : analyse de complexité

Trouver la plus grande valeur dans un tableau d'entiers positifs

- Les deux algorithmes fonctionnent, **mais** ne sont pas également efficaces

Exemple:

```
int[] array = {3, 7, 4, 2, 9, 5};
```

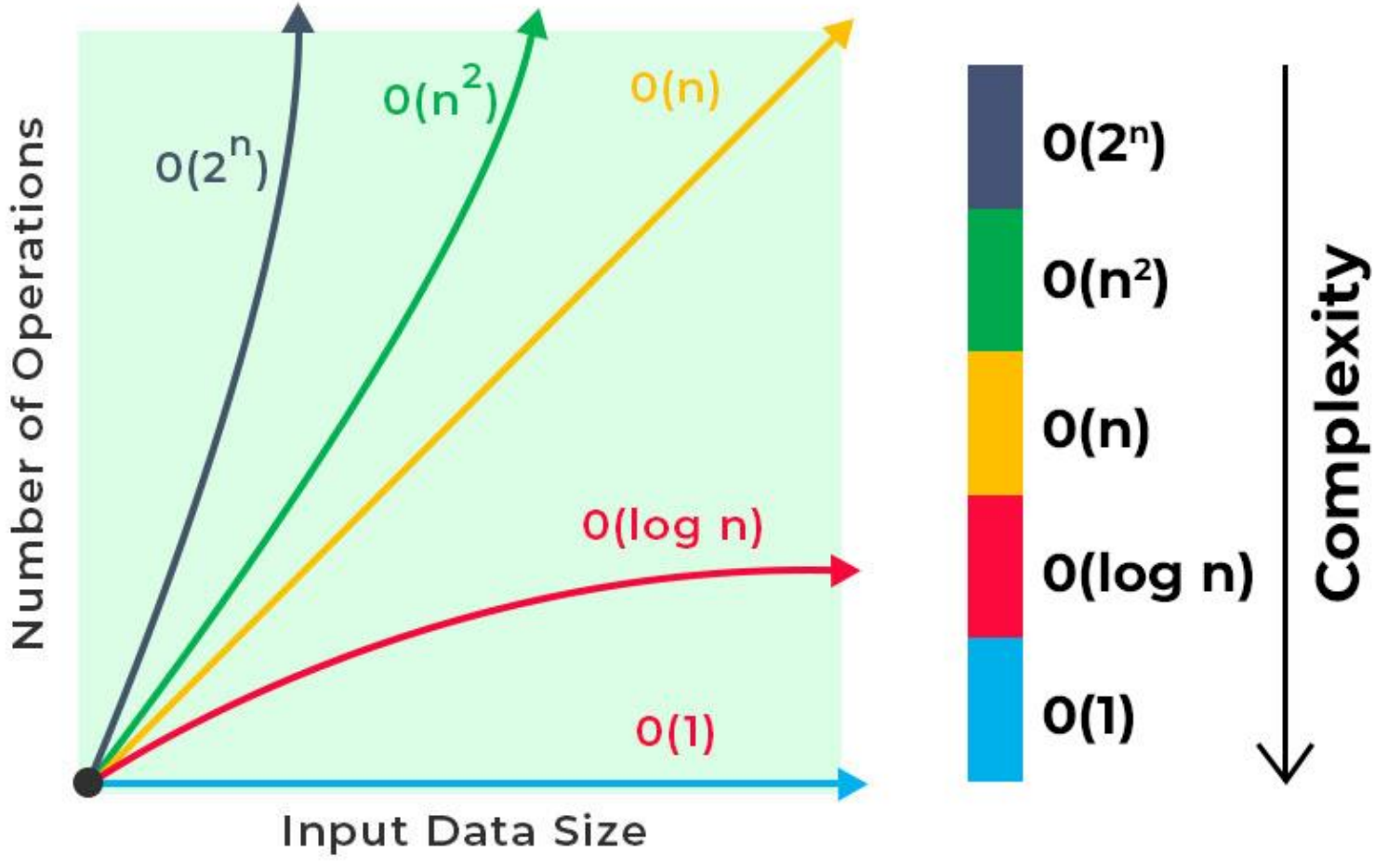
```
int getArrayMax(int[] array) {
  int n = array.length;
  if (n == 0)
    return -1;
  int currentMax = array[0];
  for (int i = 1; i < n; i++) {
    if (array[i] > currentMax)
      currentMax = array[i];
  }
  return currentMax;
}
```

```
int getArrayMax(int[] array) {
  int i, n = array.length;
  if (n == 0)
    return -1;
  boolean isMax;
  for (i = n - 1; i > 0; i--) {
    isMax = true;
    for (int j = 0; j < n; j++) {
      if (array[j] > array[i]) {
        isMax = false;
        break;
      }
    }
    if (isMax)
      break;
  }
  return array[i];
}
```

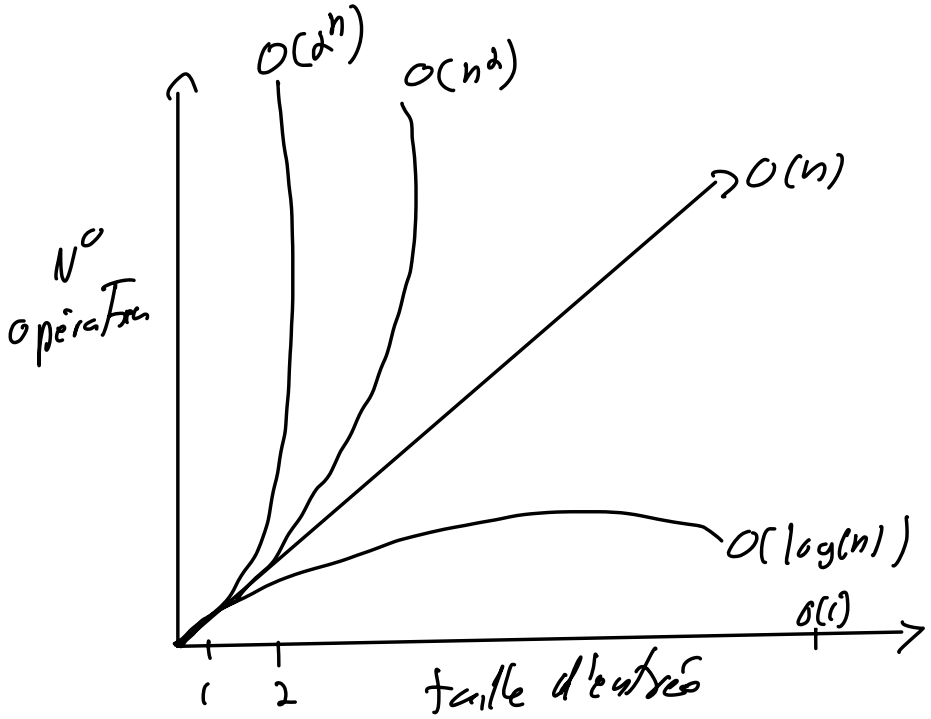
L'**analyse de complexité** peut être vue comme une forme de **quantification formelle** de cette efficacité

On peut également procéder à des mesures ***empiriques***

Algorithmes : analyse de complexité



Source: https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/



ex

$n = 16$
$op = 4$

$f(16) = 4$?
$\log_2(16) = 4$

$n = 100$
$op = 10000$

$f(100) = 10000$?
(n^2)

Algorithmes : analyse de complexité


```
class GFG {
    public static void main(String[] args)
    {
        int n = 3;
        int m = 3;
        int arr[][]
            = { { 3, 2, 7 }, { 2, 6, 8 }, { 5, 1, 9 } };
        int sum = 0;


        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {

                sum += arr[i][j];
            }
        }
        System.out.println(sum);
    }
}
```

Algorithmes : analyse de complexité

Procédure générale

1. Retrouver les données d'entrée et leur "nombre" *n*
2. Exprimer le nombre d'opérations en fonction de *n*
3. Ne garder que les termes de plus haut ordre
4. Enlever tous les facteurs constants

```
int getArrayMax(int[] array) {
  int n = array.length;
  if (n == 0)
    return -1;
  int currentMax = array[0];
  for (int i = 1; i < n; i++) {
    if (array[i] > currentMax)
      currentMax = array[i];
  }
  return currentMax;
}
```

Exercices

```
int getArrayMax(int[] array) {
  int i, n = array.length;
  if (n == 0)
    return -1;
  boolean isMax;
  for (i = n - 1; i > 0; i--) {
    isMax = true;
    for (int j = 0; j < n; j++) {
      if (array[j] > array[i]) {
        isMax = false;
        break;
      }
    }
    if (isMax)
      break;
  }
  return array[i];
}
```

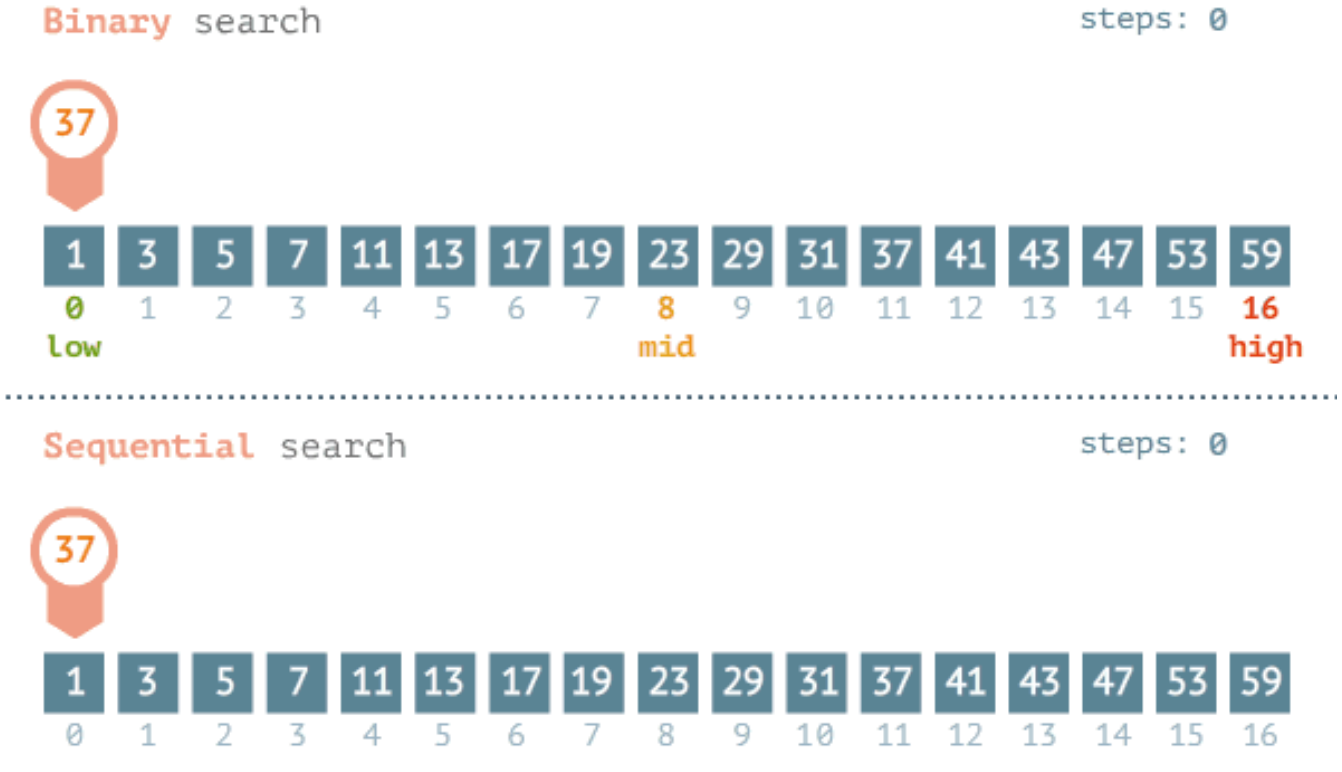
1. Trouvez la **complexité** des implémentations de `getArrayMax()`
2. Proposez un algorithme qui retourne, pour un tableau de **réels**, la paire de valeurs les plus **éloignées** l'une de l'autre
3. Cherchez un exemple d'algorithme à complexité **logarithmique**

Algorithmes : classification

- Par implémentation
  - **Récursivité** ou itération
  - Séquentiel, parallèle, ou distribué
  - Déterministe ou non déterministe
  - Exacte ou approchée
  - Quantique
- Par paradigme de conception
  - Force brute
  - **Diviser pour régner**
  - Recherche et énumération
- Problèmes d'optimisation
  - Programmation linéaire
  - Programmation dynamique
  - Algorithmes gloutons
  - Méta-heuristiques
    - *Hill Climbing*
    - Recherche taboue
    - Recuit simulé
    - Algorithmes génétiques
    - …

Algorithmes : Exemples

- Algorithmes de recherche (Séquentiel vs Binaire)



Source: www.mathwarehouse.com

www.mathwarehouse.com

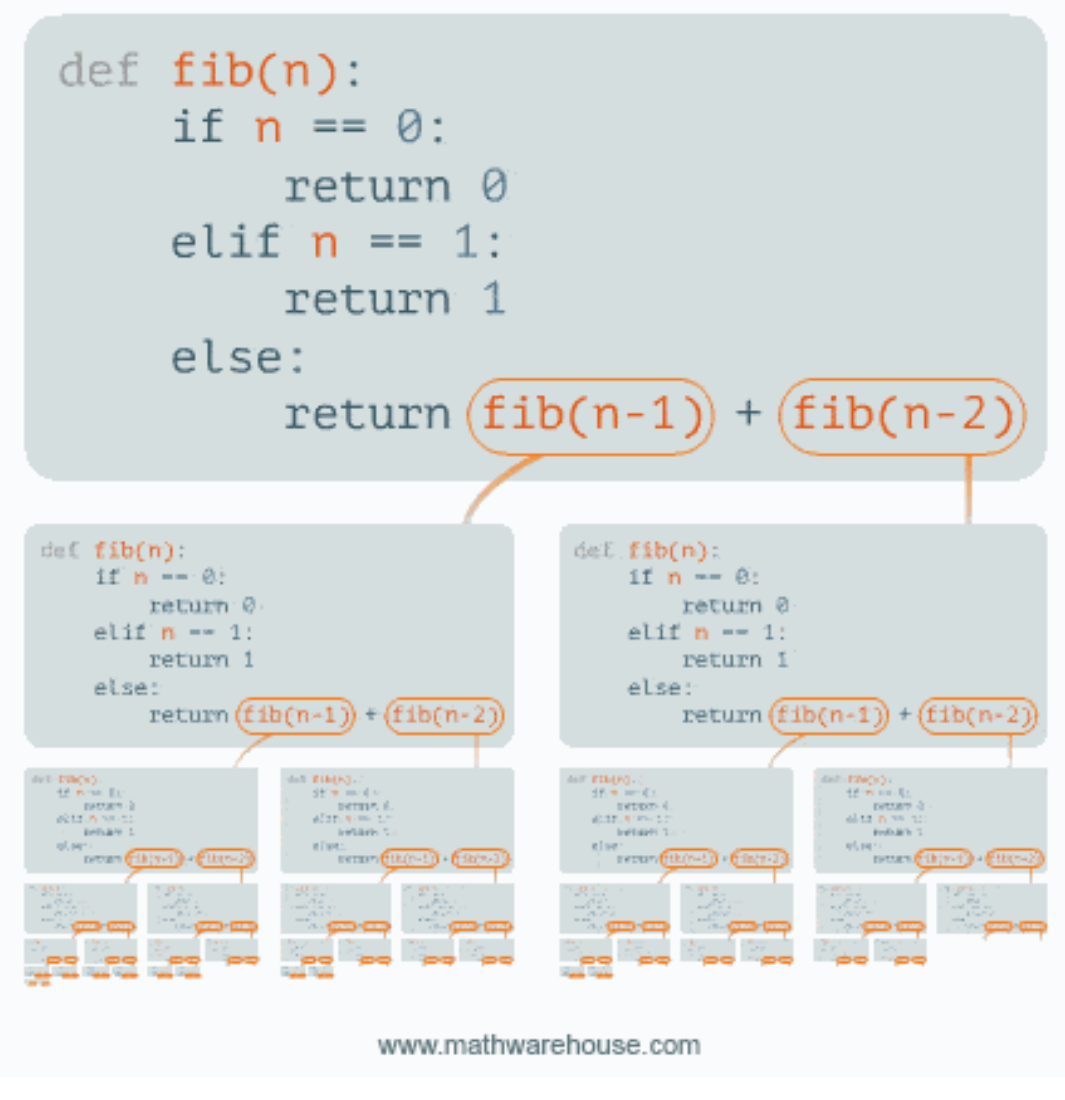
Récursivité d'algorithme

- Contexte général
  - La tactique « Diviser pour régner »
    - Principe : **diviser** un problème en sous-problèmes du même type que le problème original, **résoudre** ces sous-problèmes, et **combiner** les résultats
    - On peut rajouter une table qui stocke les résultats de ces sous-problèmes
- Définition d'une fonction récursive
  - Un ou plusieurs cas de **base**
    - Entrées générant des sorties triviales sans récursion
      - Essentielles pour éviter des boucles infinies
  - Un ou plusieurs cas de **récursion**
    - Entrées pour lesquels la fonction s'appelle elle-même

```
public static int factorial(int n) {
  if (n == 1) return 1;
  return n * factorial(n - 1);
}
```

Algorithmes : Exemples

- Algorithmes récurives



Source: www.mathwarehouse.com

Récursivité d'algorithme

Exercices

Algorithm A1 Euclids algorithm

```
1: procedure EUCLID(a, b)
2:     r ← a mod b
3:     while r ≠ 0 do
4:         a ← b
5:         b ← r
6:         r ← a mod b
7:     end while
8:     return b
9: end procedure
```

1. Proposez une version **récursive** de l'algorithme d'Euclide
2. Proposez un algorithme **récursif** pour la recherche d'un entier dans un tableau d'**entiers trié**
3. Commenter la fonction récursive suivante :

```
public static double h(int n) {
  return 1.0 / n + h(n - 1);
}
```

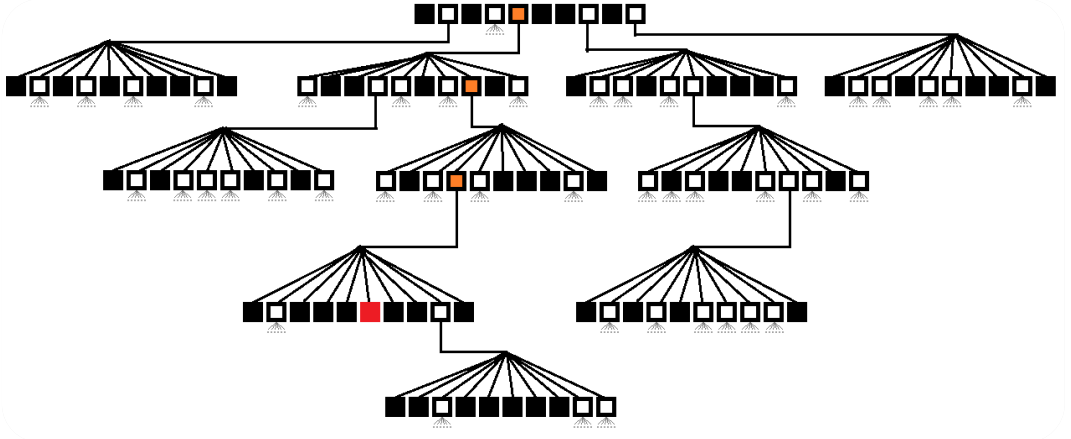
Récursivité d'algorithme

1. Évaluez `mystery1(1, 4)`
2. Évaluez `mystery1(2, 3)`
3. Que fait `mystery1` ?
4. Évaluez `mystery2(1, 4)`
5. Évaluez `mystery2(2, 3)`
6. Que fait `mystery2` ?
7. Que fait `mystery3` ?
8. Proposez une version de `mystery3` avec **un seul** appel récursif

```
public static int mystery1(int a, int b) {
  if (b == 0)
    return 0;
  else
    return a + mystery1(a, b - 1);
}
```

```
public static int mystery2(int a, int b) {
  if (b == 0)
    return 1;
  else
    return a * mystery2(a, b - 1);
}
```

```
public static String mystery3(String s) {
  int n = s.length();
  if (n <= 1)
    return s;
  String a = s.substring(0, n / 2);
  String b = s.substring(n / 2, n);
  return mystery3(b) + mystery3(a);
}
```

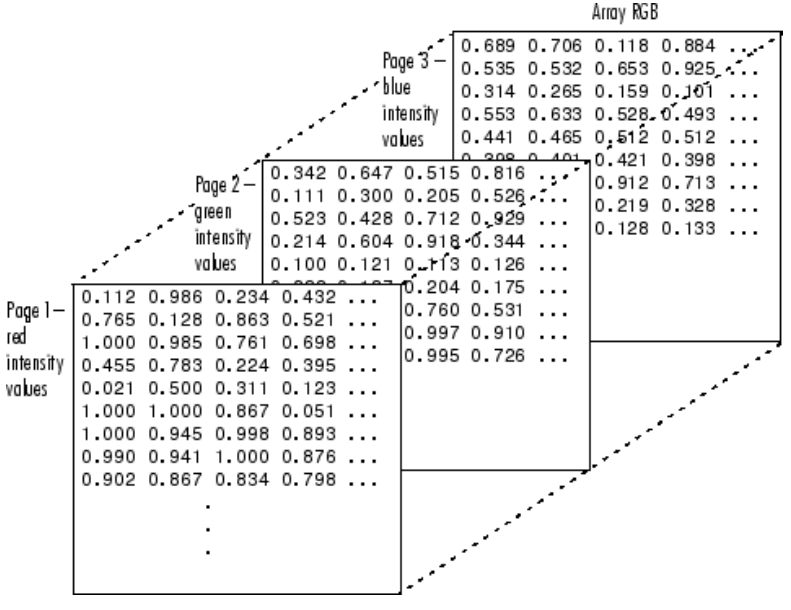


STRUCTURES DE DONNÉES

Listes chaînées, arbres, structures Java

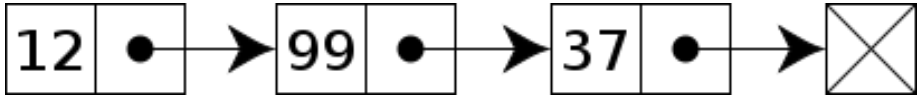
Structures de données

- **Structure de données –** manière particulière d'organiser les données pour une utilisation efficace
  - Bâties sur les structures de données primitives
  - S'inspirent de structures mathématiques connues
  - S'accompagnent, en POO, d'opérations essentielles
- Quelques structures de données :
  - Tableaux et listes
  - Tableaux associatifs, tables de hachage
  - Ensembles
  - Arbres et graphes
  - Classes et objets



Listes chaînées

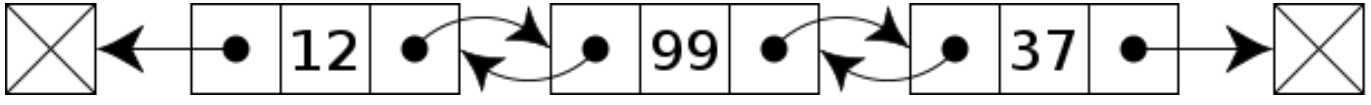
- Groupe de nœuds qui, pris ensemble, forment une **séquence**
  - Sous la forme la plus simple, chaque nœud est composée d'une donnée et d'une **référence** (un lien) vers le **nœud suivant** dans la séquence



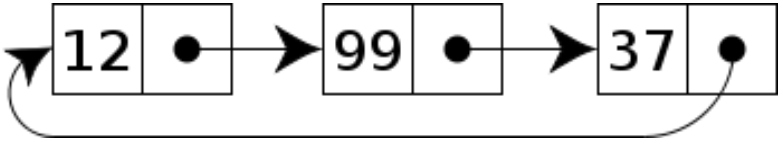
- Structure efficace pour l'**ajout** et la **suppression** d'éléments à **n'importe quelle** position de la séquence
- Permettent d'implémenter des types abstraits tels que :
  - Les piles (*stack*)
  - Les files (*queue*)

Listes chaînées : variantes

- Listes doublement chaînées :



- Listes circulaires :



Étant donnée une liste chainée simple non circulaire, quelles méthodes faut-il pour implémenter :

- Une pile ?
- Une file ?

Listes chaînées : exercice


```
public class LinkedList {
  private Node head;

  private class Node {
    Node next;
    Object data;

    public Node(Object _data) {
      next = null;
      data = _data;
    }

    public Node(Object _data, Node _next) {
      next = _next;
      data = _data;
    }

    // Accessors/Modifiers for next and data
  }
}
```

Implémentez les méthodes suivantes de `LinkedList` :

1. **`public`** `LinkedList()` – constructeur
2. **`public void`** `add(Object data)` – ajoute data à la fin de la liste
3. **`public void`** `add(Object data, int index)` – ajoute data à une position donnée (`index`)
4. **`public`** `Object get(int index)` – retourne l'objet à la position `index`
5. **`public boolean`** `remove(int index)` – supprime l'objet à la position `index`, renvoie `true` si l'opération s'est effectuée correctement
6. **`public int`** `size()` – retourne le nombre de nœuds
7. **`public`** `Object getMthToLast(int m)` – retourne l'élément qui est à *m* indices du dernier élément

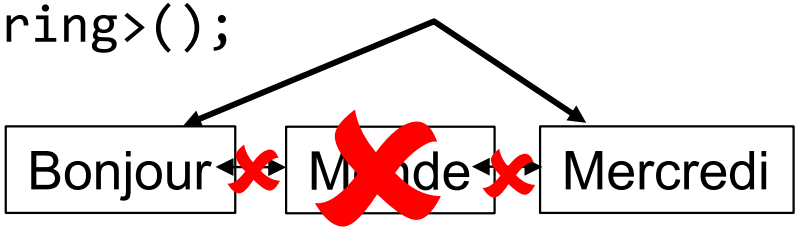
LinkedList<E> – constructeurs et méthodes importantes

- Constructeurs :
  - LinkedList() – celui-ci construit une liste vide
  - LinkedList(Collection<? extends E> c) – celui-ci construit une liste basée sur une autre collection

- Quelques méthodes importantes :
  - boolean add(E e) – ajoute un élément à la fin de la liste
  - E get(int index) – retourne un élément à la position index, le premier élément se retrouve à la position 0
  - E remove(int index) – efface l'élément à la position index

LinkedList<E> – exemple

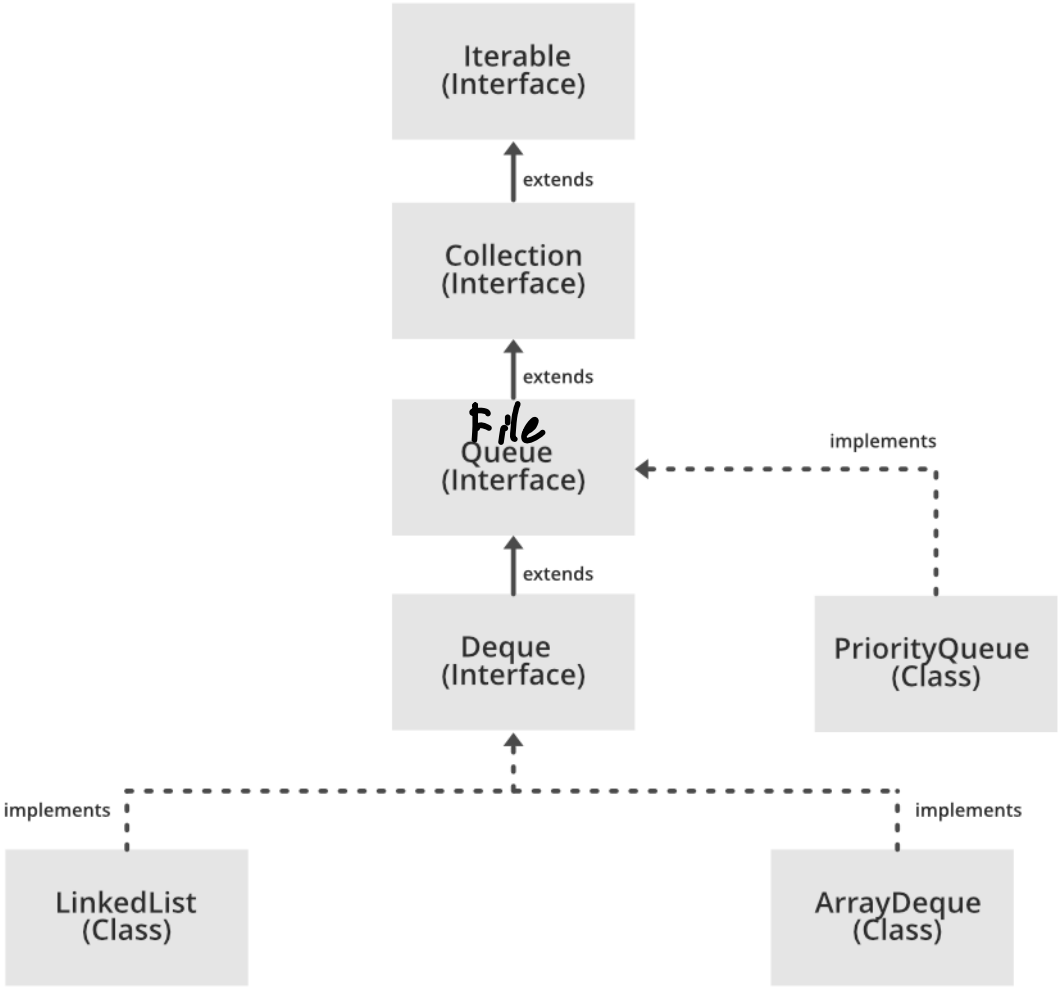

```
List<String> liste = new LinkedList<String>();
liste.add("Bonjour");
liste.add("Monde");
liste.add("Mercredi");
liste.remove(1);
System.out.println("Taille " + liste.size( ));
String deuxieme = liste.get(1);
System.out.println(" Deuxieme " + deuxieme);
```



Les Files

- La File est une interface dans java qui étend autres classes (***Collections*** et ***Iterable***), mais aussi est également étendu par d'autres interfaces et classes (Ex: ***Deque, LinkedList, PriorityQueue***)

Les Files



Source: https://www.geeksforgeeks.org/queue-interface-java/

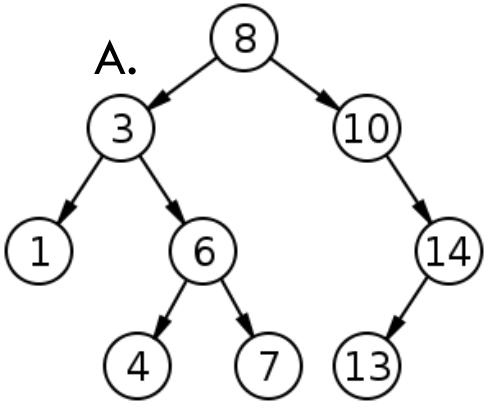
Les Piles

- La Pile est une classe qui étend la classe (***Vector***). C'est possible de l'utiliser en écrivant:
- Stack<Integer> pile = new Stack<Integer>();

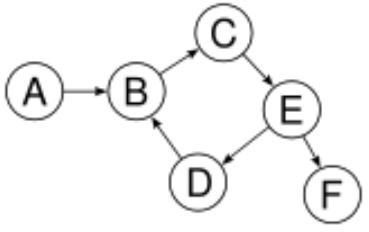
Arbres

- Structure hiérarchique d'arbre
  - Une valeur **racine** et des sous-arbres
  - Des nœuds, des arcs, un graphe **connecté, pas de cycle** (même dans une perspective non-orientée)

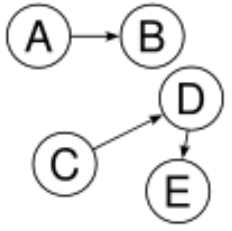
A.



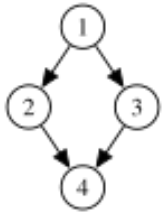
C.



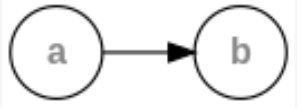
E.



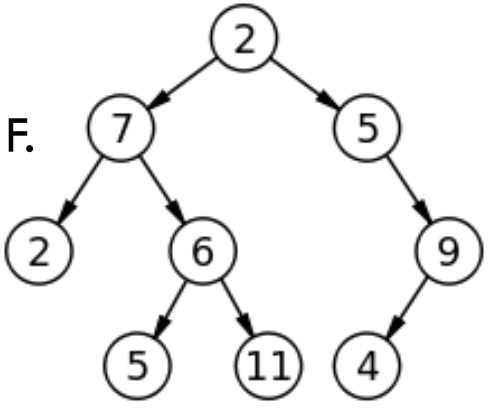
B.



D.



F.

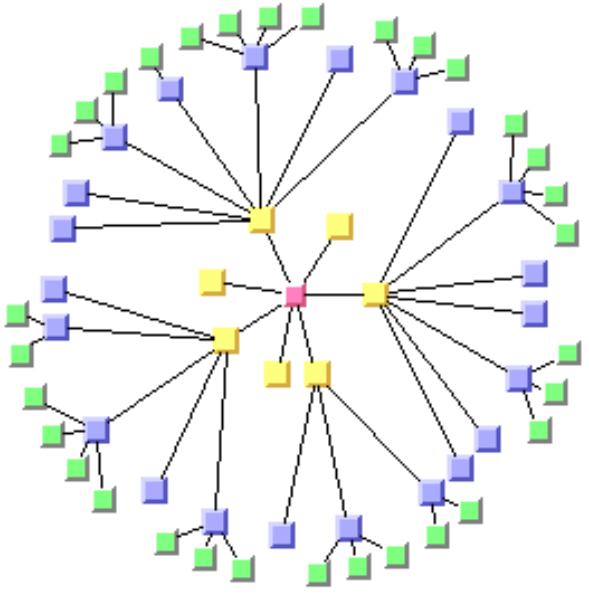


Arbres : terminologie et implémentation

- Le nœud **racine** : au sommet de l'arbre, sans parent
- Nœud **parent** → Nœud **enfant** (lien direct)
- **Descendant** de *x* : un nœud pour lequel il existe un **chemin** en partant de *x* ; *x* est son **ancêtre**
- **Feuille** : nœud sans enfant
- **Profondeur** : entre feuille et racine
- **Hauteur** de l'arbre : profondeur max.

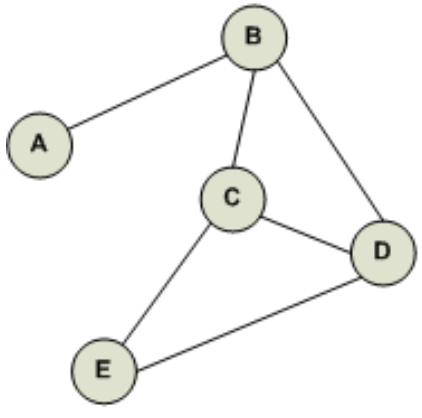
Possible implémentation :

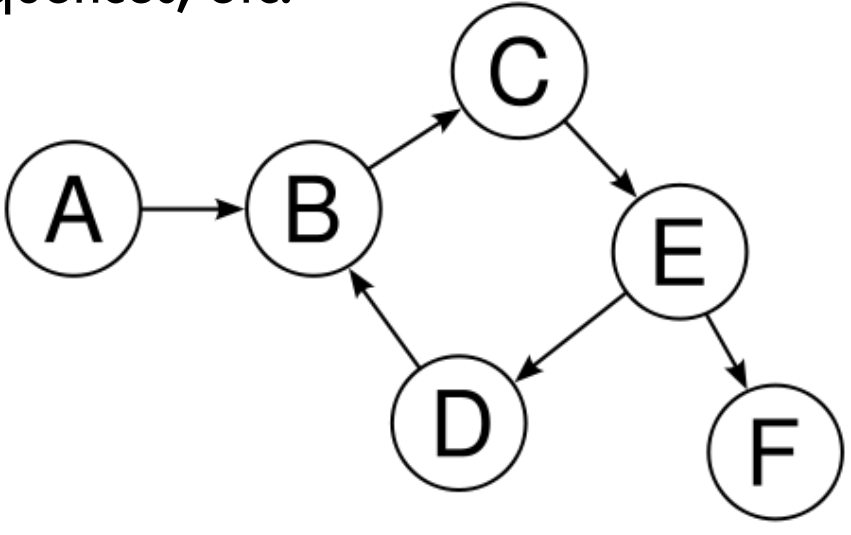
```
public class Node {
   public Node[] children;
}
```



Graphes

- Un nombre fini de nœuds et un ensemble d'arcs reliant ces nœuds
- Une des structures mathématiques les plus utilisées
  - Comprend les arbres, les séquences, etc.





Proposez une structure pour représenter un graphe

Structures de données Java

- `Collection` : interface très générale
- Structures de données **abstraites** :
  - `Set` : chaque élément de la collection est unique
  - `SortedSet` : les éléments sont triés selon un ordre donné
  - `List` : une collection ordonnée (indexée)
- Structures de données **concrètes** :
  - `HashSet` : une implémentation de `Set` qui u… localiser les éléments du `Set`
  - `TreeSet` : une implémentation de `SortedSet` qui stocke ses éléments dans un arbre binaire équilibré
- `LinkedList`, `ArrayList`, `Vector` : quelques implémentations de l'interface `List`

Un arbre binaire équilibré est défini comme un arbre binaire dans lequel la hauteur des sous-arbres gauche et droit de n'importe quel nœud ne diffère pas de plus de 1
Source: https://www.programiz.com/dsa/balanced-binary-tree

HashSet<T> – constructeurs et méthodes importants


```
HashSet<String> voitures = new HashSet<String>();

voiture.add(“Tesla");
voiture.add(“BMW");
voiture.add(“Audi");
voiture.add(“Toyota");
voiture.add(“Maclaren");
voiture.add(“Audi");
System.out.println(voitures + "");
```

Résultat:

```
[Maclaren, Toyota, Audi, Tesla, BMW]
```

TreeSet<S> – constructeurs et méthodes importants

- Constructeurs :
  - TreeSet() – celui-ci construit un ensemble vide
  - TreeSet(SortedSet<E> s) – celui-ci construit un ensemble basé sur un autre ensemble
  - TreeSet(Collection<? extends E> c) – celui-ci construit un ensemble basé sur un autre collection
  - TreeSet(Comparator<? super E> comparator) – celui-ci construit un ensemble vide qui va être ordonné avec un comparateur spécial

éléments du type E ou des sous-classes de E

objet qui implémente l'interface Comparator pour un type E ou pour une des super-classes de E

- Quelques méthodes importantes :
  - E first() – retourne le premier élément (le plus petit) dans l'ensemble
  - E last() – retourne le dernier élément (le plus grand) dans l'ensemble
  - SortedSet<E> subSet(E fromElement, E toElement) – retourne un sous-ensemble avec les éléments entre fromElement (inclusif) et toElement (exclusif), i.e. [fromElement. toElement)

TreeSet<S> – constructeurs et méthodes importants


```
TreeSet<String> tree_set = new TreeSet<String>(new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return o2.compareTo(o1);
    }
});

tree_set.add(“A");
tree_set.add(“Z");
tree_set.add(“F");
tree_set.add(“30");
tree_set.add(“E");
tree_set.add("S");
```

TreeSet<S> – exemple


```
String couleurs[ ] = {"jaune", "bleu", "blanc", "noir", "vert",
"rouge"};
SortedSet<String> arbre =
        new TreeSet<String>( Arrays.asList(couleurs) );
String premier = (String) arbre.first( );
String dernier = (String) arbre.last( );
System.out.println("Premier " + premier + " dernier " + dernier);
```

Arrays est une classe dans framework Collections

asList transforme un tableau en une liste (Collection)

blanc

vert

Structures de données Java