

# Structure de données: avancé

Christophe Damas

José Vander Meulen

# Planning

- Collections & Maps (rappel) : 2 séances
- Graphes : 2 séances + projet (3 séances)
- Arbres : 2 séances
- Algorithme Minimax: 1 séance
- Code de Huffman : 2 séances

# Evaluation

- En juin
  - 10 % projet
  - 90 % examen sur machine
- En septembre
  - 100 % examen sur machine

C

en tant qu'informaticien, est-ce indispensable de bien connaître les structures de données comme les arbres binaires et les graphes? Est-il également indispensable de savoir calculer les complexités d'un algorithme en notation  $O(f(n))$ ?



Oui, une bonne compréhension des structures de données est importante pour un informaticien. Les structures de données sont une partie fondamentale de la programmation et sont utilisées pour organiser et stocker des données de manière efficace. Les structures de données telles que les arbres binaires et les graphes sont souvent utilisées pour résoudre des problèmes complexes dans divers domaines, tels que les réseaux, les bases de données et les algorithmes génétiques.

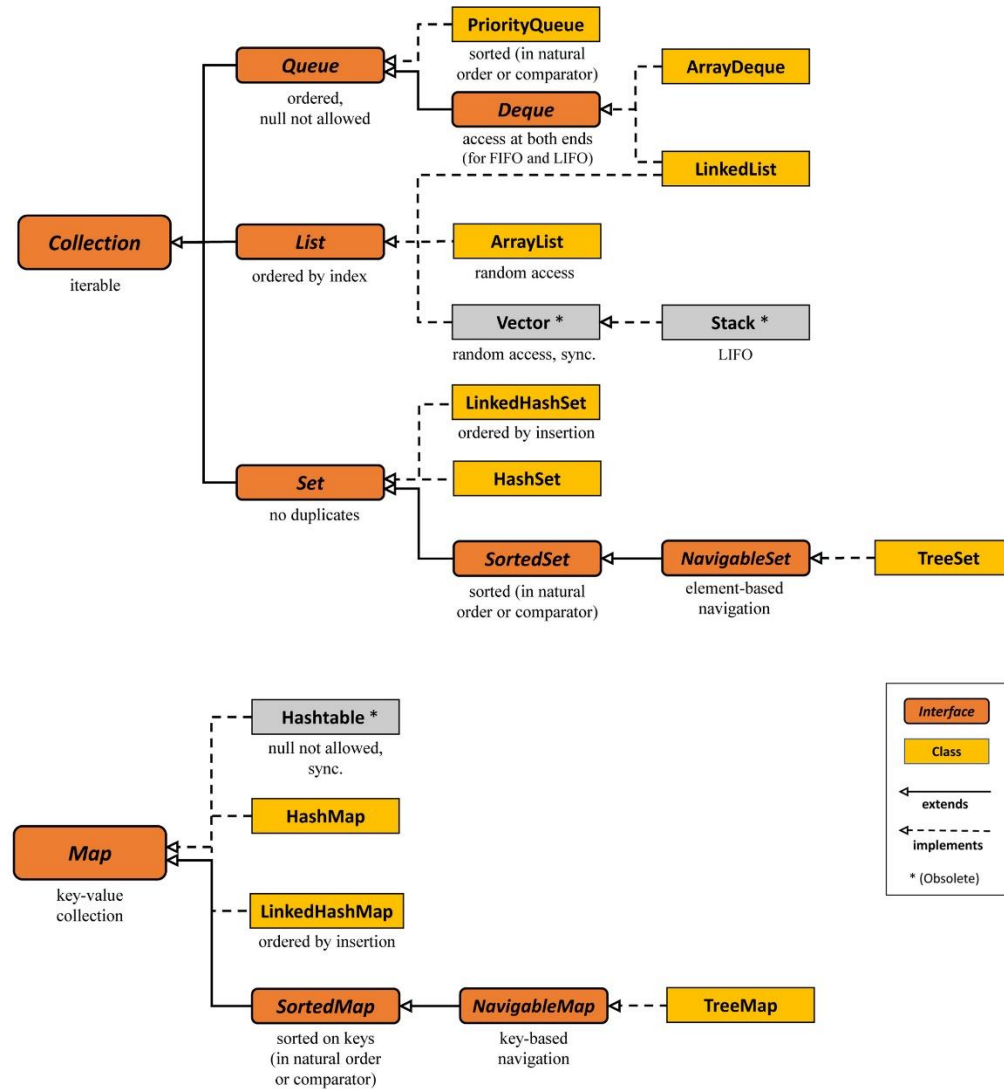


De même, savoir calculer la complexité d'un algorithme en notation  $O(f(n))$  est également très important pour un informaticien. La complexité algorithmique aide à évaluer la performance d'un algorithme et à comprendre ses limites. Cela permet aux développeurs de choisir les algorithmes les plus appropriés pour différents types de tâches et de garantir que leur code fonctionne efficacement, même pour les grands jeux de données. En conclusion, la compréhension des structures de données et la capacité à mesurer la complexité algorithmique sont des compétences importantes pour un informaticien.

# **COLLECTIONS, MAPS: RAPPEL**

Attention, cette présentation n'est qu'un bref rappel.  
Pour plus d'informations, référez-vous à la javadoc.

## Java 8 - Collections & Maps



# Structures de données JAVA utilisées dans ce cours

- List
  - ArrayList
- Queue
  - PriorityQueue
- Deque
  - ArrayDeque
  - LinkedList
- Set
  - HashSet
  - TreeSet
- Map
  - HashMap
  - SortedMap

# Interface List

- Séquence ordonnée

```
public interface List<E>{  
    boolean isEmpty();  
    int size();  
    E get(int index);  
    void add(int index, E element);  
    void add(E element);  
    E remove(int index);  
    boolean contains(Object o);  
    String toString();  
}
```

- Implémentation:
    - Basée sur un tableau: ArrayList
    - Basée sur une liste doublement chaînée: LinkedList
- Complexité ?



# Interface Deque

- Collection linéaire qui permet l'ajout et la suppression des deux cotés
- Permet d'implémenter une pile (LIFO), une file (FIFO),...

```
public interface Deque<E> implements Queue<E>{
    boolean isEmpty(); int size();
    boolean contains(Object o); String toString();

    //operation pour implémenter une pile
    void push(E item);
    Object pop();
    Object peek();

    //operation pour implémenter une file
    void add(E e);
    E poll();

    ... (addFirst, addLast, removeFirst, removeLast, ...)
}
```

- 2 implémentations
  - Basée sur un tableau: ArrayDeque
  - Basée sur une liste doublement chaînée: LinkedList

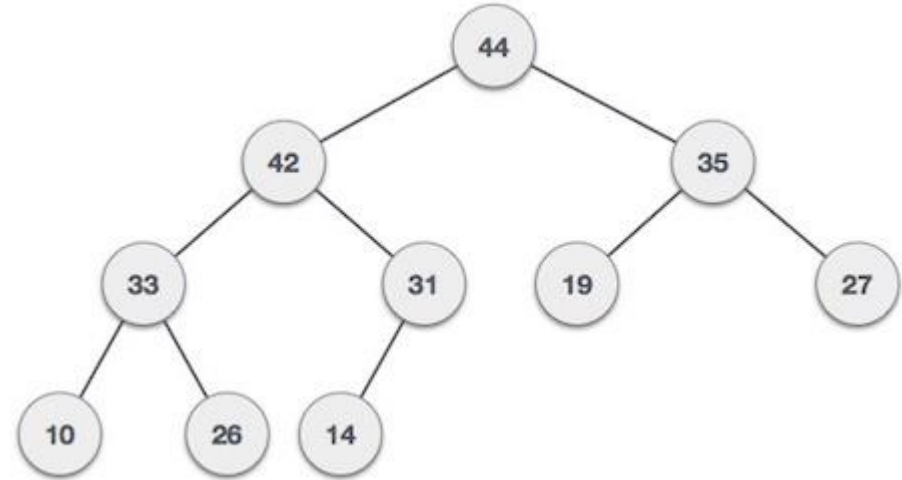
# Priority Queue

```
public class PriorityQueue<E> implements Queue<E>{  
    void add(E e);  
    E peek();  
    E poll();  
    boolean isEmpty();  
    int size();  
}
```

- Les éléments sont triés par leur ordre naturel (Comparable) ou par un Comparator donné à la construction de la file
  - Comparable vs Comparator
  - <https://www.youtube.com/watch?v=oAp4GYprVHM>

# Priority Queue: implémentation

- Basé sur les tas (heap)



- Complexité:
  - add:  $O(\log(n))$
  - peek:  $O(1)$
  - poll:  $O(\log(n))$

-	44	42	35	33	31	19	27	10	26	14	
0	1	2	3	4	5	6	7	8	9	10	

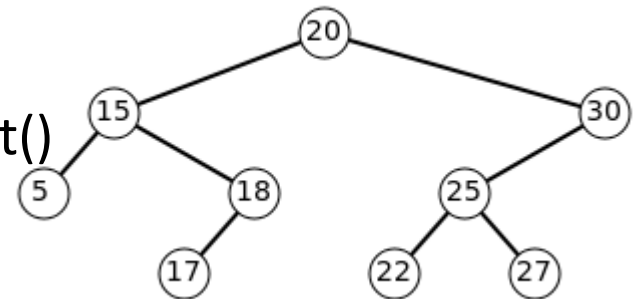
# Interface Set

- Collection qui ne contient pas de doublons

```
public interface Set<E>{  
    boolean isEmpty() ;  
    int size() ;  
    boolean add(E e) ;  
    boolean contains(Object o) ;  
    boolean remove(Object o) ;  
    String toString() ;  
}
```

# Ensemble: implémentation

- Ensemble non trié
  - table de hashage: HashSet
    - Operations en  $O(1)$
- Ensemble trié
  - implémente l'interface SortedSet
    - Méthodes supplémentaires first(), last()
  - arbre binaire: TreeSet
    - Operations en  $O(\log(N))$
  - éléments doivent pouvoir être comparé
    - Comparable vs Comparator



# PriorityQueue vs TreeSet

- Points commun
  - Insertion / suppression du maximum en  $O(\log(n))$
- Différences
  - Avantages PriorityQueue
    - Doubleton possible dans PriorityQueue
    - $O(1)$  pour `peek()` dans PriorityQueue ( $O(\log(n))$  dans TreeSet)
  - Avantages TreeSet
    - $O(\log(n))$  pour `contains()` et `remove()` dans TreeSet ( $O(n)$  dans PriorityQueue)
    - `iterator()`: l'ordre d'iteration est respecté dans TreeSet (pas respecté pour PriorityQueue)

# Dictionnaire

- Collection qui associe des clés et des valeurs

```
public interface Map <K,V>{  
    boolean isEmpty() ;  
    int size() ;  
    boolean put(K key,V value) ;  
    boolean containsKey(K key) ;  
    Object remove(Object key) ;  
    Object get(Object key) ;  
}
```

# Dictionnaire: Implémentation

- Dictionnaire non trié: HashMap
  - Table de hashage
    - Opérations:  $O(1)$
- Dictionnaire trié: TreeMap
  - Utilisation arbre binaire
    - Opérations:  $O(\log(N))$
  - Méthodes supplémentaires pour obtenir les clés les plus basses/hautes
  - Comparator/Comparable