Pour toutes les structures de données, n le nombre d'éléments (de clés, pour les dictionnaires).

Séquences (List) et tableaux

`T[]` (tableau)

- Utilisation : lorsque le nombre d'élements à stocker est connu à l'avance et fixe.
- Accès par index `a[i]` : $\Theta(1)$
- Accès en écriture `a[i]=...` : $\Theta(1)$
- Recherche d'un élément donné : $\Theta(n)$, il faut parcourir le tableau
- Insertion/suppression d'un élément : $\Theta(n)$, il faut trouver l'élément et éventuellement faire des décalages
- Mémoire : très compacte, taille fixe.

`ArrayList<T>`

- Accès / `get(i)`, `set(i)` : $\Theta(1)$
- Recherche d'un élément donné : `contains(x)` / `indexOf(x)` : $\Theta(n)$
- Ajout en fin `add(x)` : $\Theta(1)$, amorti (lorsque le tableau est plein, il est agrandi est recopié)
- Suppression en fin : $\Theta(1)$
- Insertion/suppression à l'index i : $\Theta(n)$ (décalages)
- Mémoire : compacte, taille variable.

`LinkedList<T>` (liste doublement chaînée)

- Accès par index `get(i)`, `set(i)` : $\Theta(n)$
- Recherche d'un élément donné : `contains(x)` / `indexOf(x)` : $\Theta(n)$
- Ajout/suppression en tête/fin : $\Theta(1)$
- Insertion/suppression via itérateur positionné : $\Theta(1)$
- Recherche/ `contains` : $\Theta(n)$

Piles et files (Deque / Queue)

- Comportement d'une file d'attente (FIFO) : n'importe quelle implémentation de l'interface `Queue` avec
  - `peek` (equiv. `peekFirst`)
  - `add` (equiv. `addLast`)
  - `remove` (equiv. `removeFirst`)
- Comportement d'une pile (LIFO): n'importe quelle implémentation de l'interface `DeQueue` avec
  - `peek` (equiv. `peekFirst`)
  - `push` (equiv. `addFirst`)
  - `remove` (equiv. `removeFirst`)

`ArrayDeque<T>`

- `addFirst/addLast`, `removeFirst/removeLast`, `peek` : $\Theta(1)$ amorti
- Pas d'accès indexé standard

Éléments non ordonnés et sans doublons : ensembles (Set)

`HashSet<T>` (table de hachage)

- `contains`, `remove` : $\Theta(1)$ en moyenne, pire $\Theta(n)$
- `add` : $\Theta(1)$ en moyenne, pire $\Theta(n)$, et amorti à cause du re-hash.
- Pas d'ordre sur les éléments, pas de doublons
- Sensible à `hashCode()` / `equals()`

`LinkedHashSet<T>`

- Même complexités que `HashSet`
- Préserve l'ordre d'insertion (coût mémoire un peu plus élevé)

`TreeSet<T>` (arbre équilibré, ordre naturel ou comparateur)

- `add`, `remove`, `contains` : $\Theta(\log n)$
- Itération triée : $\Theta(n)$
- Min et max (`first/last`) : $\Theta(\log n)$
- Opérations de voisinage (`lower/higher/floor/ceiling`) : $\Theta(\log n)$
- Les éléments doivent implémenter l'interface `Comparable<T>`

Association clé-valeur : dictionnaires (Map)

Fonctions d'ordre supérieur pour les mises à jour:

- valeurs non mutables : `compute` (utilisation de la clé) ou `merge`
- valeurs mutables (ex. une collection) : `computeIfAbsent`

`HashMap<K,V>`

- `get`, `containsKey`, `remove` : $\Theta(1)$ en moyenne, pire cas en $\Theta(n)$
- `put` : $\Theta(1)$ en moyenne, pire $\Theta(n)$, et amorti à cause du re-hash.
- Pas d'ordre, pas de doublons dans les clés.
- Notes : redimensionnement $\Theta(n)$ mais amorti sur la séquence d'opérations

`LinkedHashMap<K,V>`

- Même complexités que `HashMap`
- Préserve l'ordre d'insertion (coût mémoire un peu plus élevé)

`TreeMap<K,V>`

- `put`, `get`, `remove`, `containsKey` : $\Theta(\log n)$
- Parcours des clés triées : $\Theta(n)$
- Min et max (`firstEntry/lastEntry`) : $\Theta(\log n)$
- Requêtes de plage (`subMap/headMap/tailMap`) : $\Theta(\log n)$ pour positionner + itération
- Les clés doivent implémenter l'interface `Comparable<K>`

Choix rapide

- Accès indexé fréquent, taille variable : `ArrayList`
- Insertions/suppressions aux extrémités : `ArrayDeque`
- Test d'appartenance sans ordre, sans doublons: `HashSet`
- Test d'appartenance avec ordre, sans doublons: `LinkedHashSet`
- Clés → valeurs sans ordre : `HashMap`
- Clés → valeurs avec ordre d'insertion: `LinkedHashMap`
- Besoin d'ordre trié / requêtes de plage : `TreeSet` / `TreeMap`
- Extraire souvent min/max : `PriorityQueue`