



Plan

1. Structure == fonction
2. Structure == accélération
3. Exemple : tris qui dépendent de la structure
4. Pièges classiques

Structure == fonction (idée centrale)

Une structure de données ne sert pas uniquement à **stocker** des données.

Elle sert avant tout à **organiser** les données afin de rendre certaines opérations efficaces.

Choisir une structure de données, c'est choisir **comment on va travailler** avec les données.

Ainsi, une structure de données est un **choix algorithmique** : elle détermine les opérations qui seront rapide, celles qui seront lente, et les coûts liés à la maintenance.

Conteneur vs structure

Un **conteneur** naïf (par exemple un tableau non organisé) :

- stocke des données
- n'impose aucune règle particulière
- offre peu d'aide pour les opérations

Une **structure de données** :

- impose une organisation
- maintient des règles internes
- favorise certaines opérations au détriment d'autres

👉 La différence essentielle n'est pas où les données sont stockées, mais comment elles sont organisées.

La fonction dicte la structure

Avant de choisir une structure, il faut toujours se poser la question :

Quelles opérations doivent être rapides ?

Exemple, est-ce que mon application demande des :

- Recherches fréquentes ?
- Insertions fréquentes ?
- Suppressions fréquentes ?
- Accès au minimum ou au maximum ?
- Parcours séquentiel ?

La réponse à ces questions détermine la structure appropriée.

Exemple comparatif simple

Supposons que l'on doive gérer une collection de nombres.

Tableau non organisé

- Insertion : rapide $O(1)$
- Recherche : lente $O(n)$
- Accès au minimum : lent $O(n)$

👉 Bon choix si l'on insère beaucoup et cherche rarement.

Tableau triée

- Insertion : plus coûteuse $O(n)$
- Recherche : plus rapide $O(\log n)$
- Accès au minimum : immédiat $O(1)$

👉 Bon choix si l'on consulte souvent les données.

Tas (heap)

- Insertion : modérée $O(\log n)$
- Recherche : inefficace $O(n)$
- Accès au minimum : immédiat $O(1)$

👉 Bon choix si l'on veut accéder à l'élément prioritaire.

Il n'existe pas de structure universelle

Aucune structure de données n'est optimale pour toutes les opérations.

Une structure est toujours un **compromis**.

Accélérer une opération implique presque toujours une ou plusieurs de ces conséquences :

- ralentir une autre opération
- augmenter la complexité de l'implémentation
- utiliser plus de mémoire

Structure == fonction (formulation clé)

On peut résumer cette idée par la formule suivante :

La structure est déterminée par la fonction.

Autrement dit :

- on ne choisit pas une structure par habitude
- on la choisit en fonction de l'usage prévu (sa fonction)

Conséquence pour la programmation

Un mauvais choix de structure :

- rend le code plus lent
- rend les algorithmes plus complexes
- conduit souvent à des solutions artificielles ou inefficaces

Un bon choix de structure :

- simplifie les algorithmes
- rend le code plus clair
- améliore les performances sans optimisation complexe

Message à retenir

Une structure de données n'est pas un simple conteneur,
c'est une **décision algorithmique** qui encode une manière de résoudre
un problème.

Dans la section suivante...

... nous verrons pourquoi les structures permettent cette efficacité, en introduisant la notion d'**accélération** et le rôle fondamental des **invariants**.

Structure == accélération (idée centrale)

Les structures de données existent pour une raison essentielle : **accélérer certaines opérations.**

Cette accélération n'est pas obtenue par hasard ni par magie. Elle repose sur des **invariants**, c'est-à-dire des propriétés que la structure maintient en permanence.

Une structure de données est rapide parce qu'elle impose des règles strictes sur l'organisation des données.

Pourquoi l'organisation accélère ?

Si les données ne respectent aucune organisation particulière :

- il faut souvent tout parcourir
- presque toutes les opérations coûtent $O(n)$

En imposant une organisation :

- certaines questions deviennent plus simples
- certaines opérations deviennent plus rapides

👉 **L'organisation crée de l'information structurelle, qui est exploitée par les algorithmes.**

Les invariants comme source d'accélération

Un invariant est une propriété qui est toujours vraie pour une structure valide.

Exemples :

- **liste triée** → les éléments sont en ordre
- **tas** → le minimum (ou maximum) est à la racine
- **arbre de recherche** → relation clé gauche / clé droite
- **table de hachage** → clé → compartiment

Ces invariants permettent :

- d'éviter des comparaisons inutiles
- de limiter l'espace de recherche
- de prendre des décisions rapides

Accélérer = accepter un coût ailleurs

Maintenir un invariant a un coût :

- insertion plus lente
- suppression plus complexe
- implémentation plus sophistiquée

Ce coût est **volontaire**.

Une structure de données échange du travail lors des mises à jour contre de la rapidité lors des consultations.

Exemple fondamental : recherche linéaire vs dichotomique

Tableau non trié

- invariant : aucun
- recherche : $O(n)$

Tableau trié

- invariant : ordre total
- recherche dichotomique : $O(\log n)$

👉 L'algorithme de recherche est **plus complexe**, mais **beaucoup plus rapide** grâce à l'invariant (l'ordre total).

Exemple : minimum dans une collection

Liste non organisée

- trouver le minimum : $O(n)$

Tas (heap)

- invariant : noeud parent toujours < noeuds enfants => min à la racine
- trouver le minimum : $O(1)$

👉 Le travail est fait à l'avance, lors des insertions/suppressions

Algorithme plus compliqué ≠ plus lent

Intuition fréquente mais fausse :

« Un algorithme compliqué est forcément lent »

Réalité :

- un algorithme simple sur une structure naïve peut être très lent
- un algorithme plus élaboré exploitant un invariant peut être très rapide

👉 La **complexité algorithmique** dépend du **contexte structurel**.

Où se cache la complexité ?

Dans une structure efficace :

- la complexité est souvent déplacée :
 - vers l'initialisation
 - vers les mises à jour
 - vers la maintenance de l'invariant

Exemples :

- déplacements dans un tableau trié
- réajustement dans un tas
- rotations dans un arbre pour l'équilibrer
- redimensionnement d'une table de hachage

Tris qui dépendent de la structure

Les méthodes de tri par **sélection**, **insertion** et **heapsort** poursuivent le même **objectif** et diffèrent essentiellement par la **structure de données utilisée** et les **invariants maintenus**.

Le problème commun

On veut trier une collection d'éléments.

À chaque étape, on cherche à :

- extraire un élément “intéressant” (minimum, ou position correcte)
- le placer au bon endroit

👉 La différence entre ces méthodes de tri ne vient pas du but, mais de la manière dont on organise les données intermédiaires.

Tri par sélection - aucune structure spéciale

Principe

- À chaque étape, on cherche le minimum dans la partie non triée
- On l'échange avec la position courante

Structure utilisée

- Tableau non organisé

Invariant

- La partie gauche est triée
- La partie droite n'a **aucune organisation**

Coût

- Recherche du minimum : $O(n)$
- Répété n fois → $O(n^2)$

👉 **Structure pauvre → travail répété**

7	4	8	2	5	3	9
2	4	8	7	5	3	9
2	3	8	7	5	4	9
2	3	4	7	5	8	9
2	3	4	5	7	8	9
2	3	4	5	7	8	9
2	3	4	5	7	8	9
2	3	4	5	7	8	9

Tri par insertion - structure partiellement organisée

Principe

- On insère chaque nouvel élément à la bonne place
- On maintient une partie triée

Structure utilisée

- Tableau avec préfixe trié

Invariant

- La partie gauche est triée

Coût

- Insertion : $O(n)$
- Répété n fois $\rightarrow O(n^2)$ [mais souvent meilleur en pratique]

7	4	8	2	5	3	9
7	4	8	2	5	3	9
4	7	8	2	5	3	9
4	7	8	2	5	3	9
2	4	7	8	5	3	9
2	4	5	7	8	3	9
2	3	4	5	7	8	9
2	3	4	5	7	8	9

👉 Invariant plus fort \rightarrow moins de travail inutile

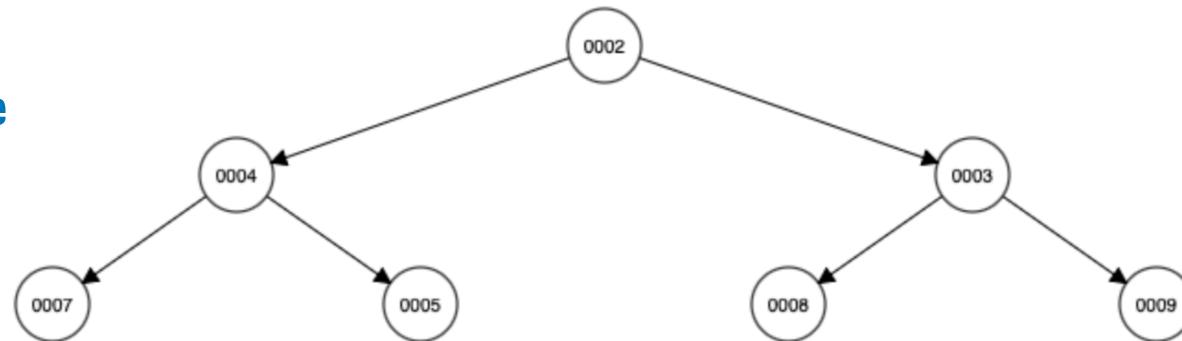
Heapsort - structure spécialisée (tas)

Principe

- On place tous les éléments dans un tas
- On extrait le minimum à chaque étape

Structure utilisée

- Tas (heap)



Invariant

- Le minimum est toujours accessible à la racine

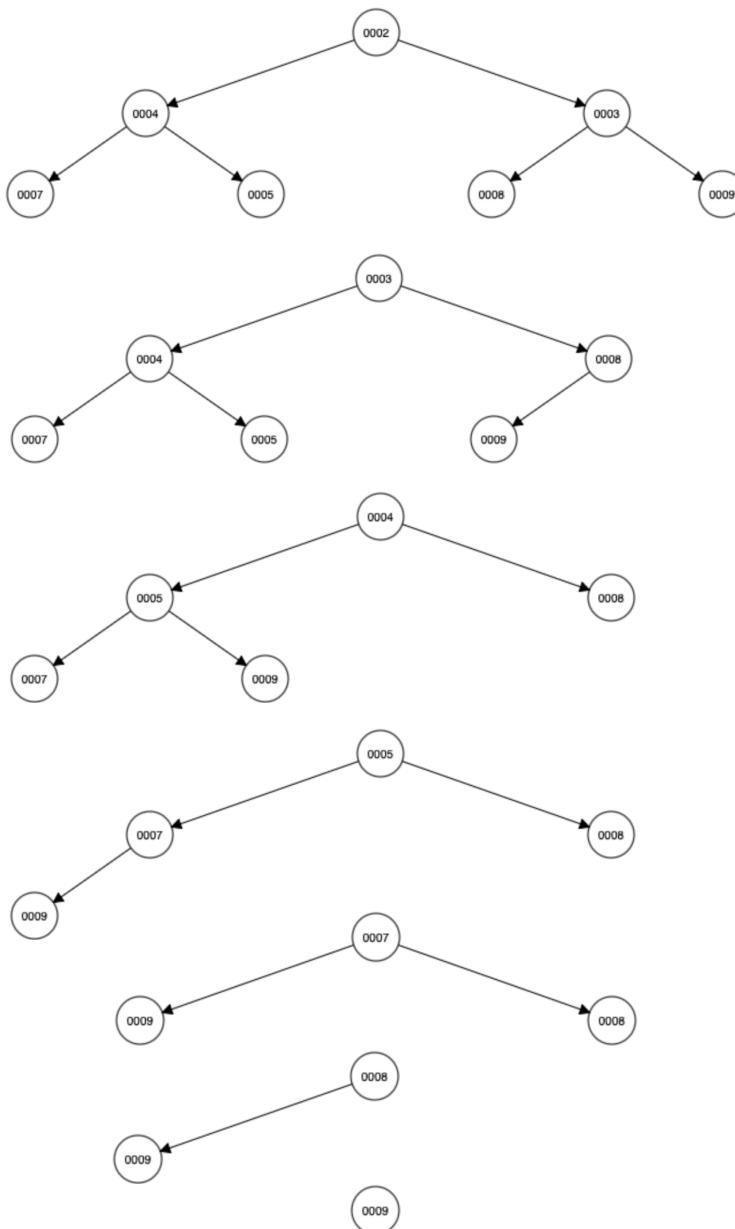
Coût

- Construction du tas : $O(n)$
- Extraction répétée : n fois $O(\log n)$
- Total : **$O(n \log n)$**

👉 Invariant fort → accélération majeure

7	4	8	2	5	3	9
7						
4	7					
4	7	8				
2	4	8	7			
2	4	8	7	5		
2	4	3	7	5	8	
2	4	3	7	5	8	9
2	4	3	7	5	8	9
3	4	8	7	5	9	2
4	7	8	9	5	3	2
5	7	8	9	4	3	2
7	9	8	5	4	3	2
8	9	7	5	4	3	2
9	8	7	5	4	3	2
9	8	7	5	4	3	2

Heapsort - structure spécialisée (tas)



7	4	8	2	5	3	9
7						
4	7					
4	7	8				
2	4	8	7			
2	4	8	7	5		
2	4	3	7	5	8	
2	4	3	7	5	8	9
2	4	3	7	5	8	9
3	4	8	7	5	9	2
4	7	8	9	5	3	2
5	7	8	9	4	3	2
7	9	8	5	4	3	2
8	9	7	5	4	3	2
9	8	7	5	4	3	2
9	8	7	5	4	3	2

Ces 3 méthodes de tri ne diffèrent pas tant par leurs étapes que...

... par la structure de données qu'ils utilisent.

Le tri devient rapide non pas parce que l'algorithme est plus malin, mais parce que la structure maintient plus d'information.

Structure == accélération

- Sélection : aucun invariant → tout est cher
- Insertion : invariant local → gain partiel
- Heapsort : invariant global → gain asymptotique

👉 Plus l'invariant est fort, plus l'accélération est grande, mais plus le coût de maintenance est élevé.

À retenir

- 1. Les invariants rendent certaines opérations rapides.**
- 2. Maintenir un invariant a un coût.**
- 3. Un algorithme plus sophistiqué peut être plus rapide grâce à la structure.**

Message clé

La rapidité d'un algorithme ne vient pas de sa simplicité, mais de la qualité des invariants sur lesquels il s'appuie.

Pièges classiques

- Confondre structure et algorithme
- Penser que “plus de code = plus lent”
- Croire qu’une structure est “meilleure en général”
- Oublier le coût de maintenance des invariants

Pour la suite du cours

Tout le cours consiste à apprendre **quelles structures permettent quels invariants, et comment ces invariants transforment le coût des algorithmes.**

Dans les prochains blocs, nous verrons :

- quels invariants sont utilisés par les structures classiques,
- comment ils sont maintenus,
- et comment analyser les compromis qu'ils imposent.