Algorithmique et complexité

Laurent Bellon

2025-2026

Fondements et rappels :

Classes de complexité fondamentales

En théorie de la complexité, nous classifions les problèmes de décision selon les ressources (temps, mémoire) nécessaires pour les résoudre ou les vérifier. Cette classification aide à comprendre l’effort computationnel en fonction de la taille de l’entrée (n).

Classe P (Polynomial Time)

- **Définition** : Problèmes de décision résolubles en **temps polynomial** par une machine de Turing déterministe.
- **Interprétation** : Représente les algorithmes efficaces ou « faisables » en pratique.
- **Exemples** :
  - Trier une bibliothèque de livres par ordre alphabétique.
  - Vérifier si une liste de nombres est déjà triée.
  - Trouver le plus court chemin entre 2 villes sur une carte simple (comme Google Maps le fait vite).
  - Même si ces tâches peuvent prendre beaucoup de temps, le temps grandit raisonnablement.

Analogie : ranger un placard : c'est parfois long, mais ça reste faisable.

Classe NP (Nondeterministic Polynomial Time)

- **Définition** : Problèmes de décision dont une solution proposée peut être **vérifiée** en temps polynomial par une machine de Turing déterministe.
  - Alternativement : résolubles en temps polynomial par une machine de Turing non déterministe.

- **Remarque importante** : P est inclus dans NP.
- **Exemples** :
  - Deviner le mot de passe d'un compte (très long à trouver), mais une fois donné, on peut essayer → "ah oui, ça marche".
  - Retrouver le trajet optimal pour visiter plusieurs pays dans l'ordre le plus court.
  - Checker si une suite de clauses logiques (SAT) peut être vraie.

💡 Analogie : un sudoku très dur. Trouver la solution → difficile Vérifier si une grille remplie est correcte → très facile.

📌 État actuel : On ne sait pas si toutes ces tâches peuvent être rendues "rapides". C'est la question P vs NP.

Classe PSPACE (Polynomial Space)

- **Définition** : Problèmes résolubles par une machine de Turing déterministe en utilisant un **espace mémoire polynomial**.
  - Le temps peut être exponentiel, mais la mémoire reste polynomiale.
- **Relation** : P ⊆ NP ⊆ PSPACE.
- **Exemples** :
  - Certains jeux de réflexion à deux joueurs (comme Hex) : on explore les coups possibles.
  - Raisonnement logique avec état interne limité.

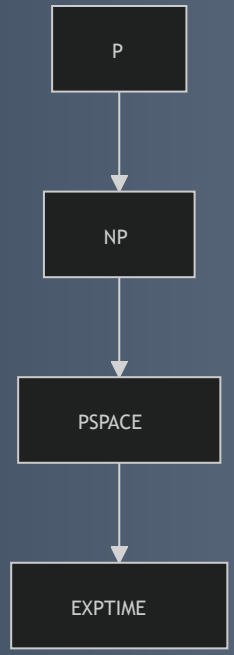
💡 Analogie : résoudre un labyrinthe en gardant en tête seulement le chemin actuel — on ne peut pas noter toutes les options, mais on peut revenir en arrière.

Classe EXPTIME (Exponential Time)

- **Définition** : Problèmes résolubles en **temps exponentiel** par une machine de Turing déterministe (temps de type (2^{p(n)})).
- **Relation** : Contient strictement PSPACE, NP, et P.
  - Hiérarchie : P ⊆ NP ⊆ PSPACE ⊆ EXPTIME

- **Exemples** :
  - Résolution exacte de certains jeux (échecs, Go) en taille arbitraire.
  - Analyse de modèles formels complexes.

Illustration de la hiérarchie des classes



Résumé des relations clés

- **P** : Problèmes décidables efficacement (polynomial déterministe).
- **NP** : Problèmes dont la solution peut être vérifiée efficacement; le statut P = NP est encore ouvert.
- **PSPACE** : Problèmes résolvables avec mémoire polynomiale, plus large que NP.
- **EXPTIME** : Problèmes nécessitant un temps exponentiel, incluant toutes les classes précédentes.

Pourquoi c'est important ?

👉 On réalise que :

- Toutes les solutions "code" ne se valent pas,
- Certains problèmes font exploser les ressources rapidement,
- Le métier de dev inclut raisonnement l'algorithmique, pas que du code.

Sources utilisées

- Complexity Classes — Not just your regular Big-O, medium.com
- Complexity Classes: P, NP, co-NP, PSPACE, concretenonsense.wordpress.com
- Complexity class, Wikipedia
- A Short Guide to Hard Problems, quantamagazine.org

Conclusion

Une bonne compréhension de ces notions est indispensable pour analyser la faisabilité algorithmique des problèmes étudiés en informatique théorique.

Fondements et rappels : Relations entre classes de complexité fondamentales et implications théoriques

La théorie de la complexité classe les problèmes selon les ressources (temps, espace) nécessaires à leur résolution. Il est fondamental de comprendre non seulement chaque classe de complexité, mais aussi leurs relations et ce que ces relations impliquent en termes de potentiel et limitations des algorithmes.

Relations d’inclusion principales

Les classes de complexité les plus étudiées sont liées par des inclusions toujours vraies, même si certaines égalités restent des questions ouvertes.

Les inclusions suivantes sont généralement admises et démontrées :

$$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$$

Détail des inclusions

- **P ⊆ NP** : Tout problème résolvable en temps polynomial peut également être vérifié en temps polynomial.
- **NP ⊆ PSPACE** : Tous les problèmes vérifiables en temps polynomial utilisent au maximum un espace polynomial. Il est possible de simuler une machine non déterministe avec espace polynomial par une machine déterministe utilisant également espace polynomial.
- **PSPACE ⊆ EXPTIME** : Un problème résolvable avec un espace polynomial peut être résolu, en temps exponentiel dans le pire cas, par une machine déterministe.

Ces inclusions sont strictes dans la majorité des cas, mais leur démonstration rigoureuse est souvent encore ouverte, notamment la célèbre question P = NP.

Implications théoriques majeures

1. **Si P = NP, alors P = PSPACE = EXPTIME** Une égalité P=NP entraînerait une simplification majeure de toute la hiérarchie connue.

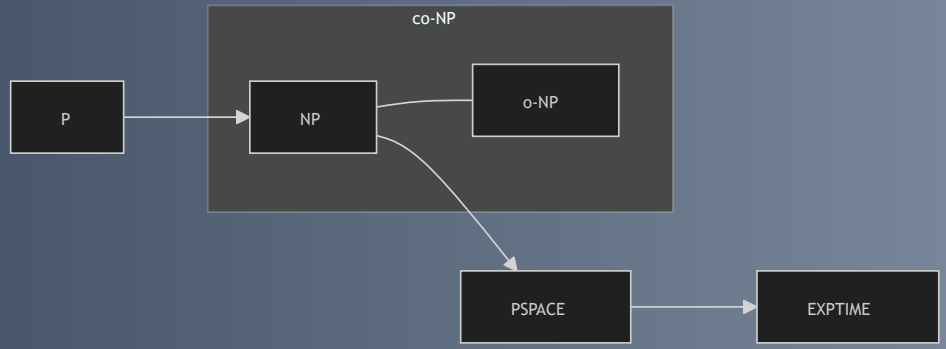
2. **Exemple de problème PSPACE-complet** La résolution de jeux à deux joueurs avec un nombre polynomial de coups (ex: Hex) est PSPACE-complet. Ces jeux sont en général encore plus difficiles que ceux de classe NP.

3. **Les classes co-NP et NP** co-NP est la classe des problèmes dont les NON solutions peuvent être vérifiées en temps polynomial. La question NP = co-NP impacterait la compréhension des problèmes "difficiles".

Exemples pour illustrer les classes

Classe	Exemple de problème	Ressource critique
P	Recherche dans liste triée (recherche binaire)	Temps polynomial
NP	Problème de satisfiabilité (SAT)	Vérification rapide
PSPACE	Jeu de Hex, TQBF (Théorie des quantificateurs booléens)	Espace polynomial
EXPTIME	Jeu d’échecs généralisé (plateau arbitraire)	Temps exponentiel

Diagramme des inclusions



Ce diagramme visualise la relation d’inclusion entre les classes avec la mention de co-NP chevauchant NP. Notez que l’égalité NP=co-NP est une question ouverte.

Notes complémentaires

- **PSPACE = NPSPACE** est démontré, une rare égalité entre classes déterministes et non déterministes en espace.
- **EXPTIME contient strictement les classes de temps polynomial**.
- On ignore si des inclusions strictes existent entre P, NP, et PSPACE (sauf PSPACE (\subseteq) EXPTIME connu).

Synthèse et conclusion

Cet article permet de comprendre les relations fondamentales entre classes classiques de la complexité computationnelle, soulignant leurs inclusions, les questions ouvertes majeures et les implications profondes sur ce qui est calculable efficacement ou non. Connaître ces frontières est déterminant pour analyser la difficulté intrinsèque des problèmes d'algorithmique.

Sources utilisées

- Classe de complexité - Wikipédia
- Co-NP, PSPACE, and EXPTIME Explained Simply - Medium
- P (complexité) - Wikipédia
- Leçon 915 : Classes de complexité. Exemples - ENS Rennesc

Fondements et rappels :

Notations asymptotiques - Big O, Omega, Theta

Pourquoi les notations asymptotiques ?

- **Objectif** : Analyser la complexité algorithmique.
- **Description** : Elles décrivent le comportement des fonctions (temps ou espace) nécessaires pour résoudre un problème, en fonction de la taille de l'entrée.
- **Avantage** : Simplifient la comparaison des algorithmes en négligeant les constantes et les termes de moindre ordre.

Big O : majoration asymptotique (limite supérieure)

- **Notation** : (f(n) = O(g(n)))
- **Définition** : Il existe des constantes positives (c) et n0 telles que, pour n > 0 0≤f(n)≤c·g(n)
- **Interprétation** : (f) croît au plus comme (g), jusqu'à une constante multiplicative à partir d'un certain rang.

Omega : minoration asymptotique (limite inférieure)

- **Notation** : (f(n) = \Omega(g(n)))
- **Définition** : Il existe (c > 0), n0 tel que, pour tout pour n > 0, 0≤cg(n)≤f(n).
- **Interprétation** : (f) croît au moins comme (g) à partir d’un certain point.

Theta : croissance asymptotiquement équivalente (majoration et minoration)

- **Notation** : (f(n) = \Theta(g(n)))
- **Définition** : Il existe deux constantes (c_1, c_2 > 0), (n_0) telles que, pour tout n > 0, c1g(n)≤f(n)≤c2g(n).
- **Interprétation** : (f) et (g) ont le même ordre de grandeur asymptotique.

Exemples concrets : Fonctions polynomiales

1. **Fonction linéaire :** $f(n) = 5n + 3$

    1. $f(n) = O(n)$ (le terme linéaire domine)
    2. $f(n) = \Omega(n)$
    3. Donc, $f(n) = \Theta(n)$

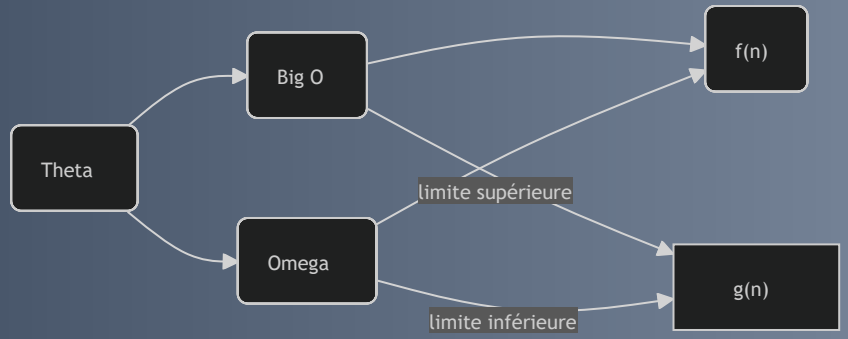
2. **Fonction quadratique :** $h(n) = 3n^2 + 2n + 1$

    1. $h(n) = O(n^2)$
    2. $h(n) = \Omega(n^2)$
    3. Donc, $h(n) = \Theta(n^2)$

Exemples concrets : Comparaison différenciée

- **Soient** :
  - $f(n) = n$
  - $g(n) = n \log n$
- **Observation** :
  - Nous avons $f(n) = O(g(n))$
  - Mais $f(n) \neq \Omega(g(n))$ car $n \log n$ croît plus vite que n.

Illustration intuitive



- **Big O** est une borne supérieure pour (f(n)) par rapport à (g(n)).
- **Omega** est une borne inférieure pour (f(n)) par rapport à (g(n)).
- **Theta** combine les deux, indiquant que (f(n)) et (g(n)) ont la même croissance asymptotique.

Utilisation pratique

- **Big O** :
  - Affirme une **complexité maximale**.
  - Exemple : « l'algorithme est (O(n^2)) » implique que le temps ne dépasse pas une fonction quadratique.
- **Omega** :
  - Utile pour démontrer des **bornes minimales**.
  - Exemple : Certains algorithmes de tri ont une borne inférieure (\Omega(n \log n)).
- **Theta** :
  - La notation la plus **précise**, indiquant la complexité exacte à l'ordre près.

Sources et références pour approfondir

- Big O notation - Wikipedia
- Asymptotic notations — GeeksforGeeks
- Notations Asymptotiques – Le Petit Aprentissage
- Introduction to Algorithms, Cormen et al., Chapitre 3 (CLRS)

Fondements et rappels : Analyse du pire cas, du cas moyen et complexité amortie

Pourquoi analyser la complexité différemment ?

L’analyse de la complexité d’un algorithme ne se limite pas à une seule mesure absolue. Pour décrire finement ses performances, trois approches principales sont utilisées :

- **Pire cas**
- **Cas moyen**
- **Complexité amortie**

Chaque méthode offre une perspective différente sur le comportement de l’algorithme selon les entrées et la fréquence des opérations.

Analyse du Pire Cas (Worst-case)

- **Définition** : Mesure du temps d'exécution maximal pris par un algorithme pour n'importe quelle entrée de taille (n).
- **Utilité** : Fournit une garantie absolue que l'algorithme ne dépassera jamais ce temps, utile pour des applications temps réel ou critiques.
- **Exemple** :
  - Recherche linéaire dans une liste non triée : pire cas = (O(n)), lorsque l'élément recherché est absent ou en fin de liste.

Analyse du Cas Moyen (Average-case)

- **Définition** : Durée moyenne d'exécution d'un algorithme en supposant une distribution probabiliste donnée sur l'ensemble des entrées de taille (n).
- **Précision** : Plus proche de la performance réelle mais dépend fortement de la modélisation statistique des entrées (souvent hypothèse d'entrées uniformément distribuées).
- **Exemple** :
  - Tri rapide (Quicksort) a un temps moyen en (O(n \log n)), même si son pire cas est (O(n^2)).
  - Recherche dans une table de hachage : temps moyen constant (O(1)), pire cas linéaire (O(n)).

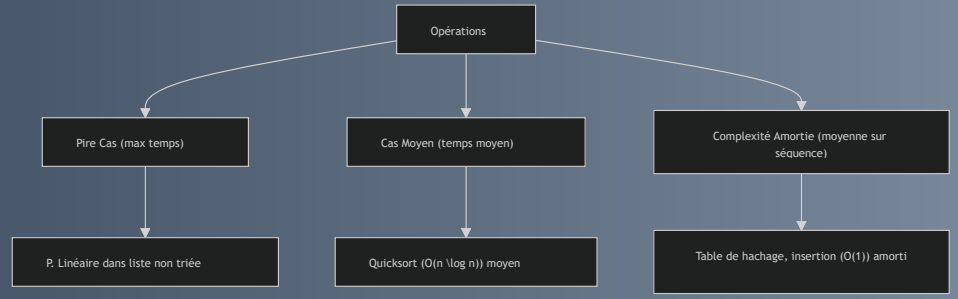
Complexité Amortie

- **Définition** : Temps moyen d'une opération pris sur une séquence d'opérations, lissant les coûts élevés ponctuels sur la totalité des opérations.
- **Objectif** : Éviter de se focaliser sur un coût ponctuel élevé mais rare.
- **Exemple** :
  - Table de hachage dynamique où la réallocation coûteuse est compensée par de nombreuses insertions rapides.
  - Insertion dans une pile avec opération `push` en (O(1)) et `resize` occasionnel en (O(n)), amorti à (O(1)) par insertion.

Illustrations par exemples

Type d’analyse	Exemple	Complexité typique
Pire cas	Tri fusion	$O(n \log n)$
Cas moyen	Quicksort (supposé aléatoire)	$O(n \log n)$
Complexité amortie	Table de hachage (réallocation)	$O(1)$ amorti par insertion

Comparaison des analyses



Synthèse

- Le **pire cas** fournit une borne supérieure sûre mais parfois pessimiste.
- Le **cas moyen** représente la performance espérée sous hypothèses statistiques.
- La **complexité amortie** évalue l'effort moyen par opération dans une séquence, souvent plus proche de la réalité dans des structures avec opérations coûteuses occasionnelles.

Sources et références

- Introduction to Algorithms, Cormen et al., Sections 2.1-2.3 (MIT Press)
- Worst, Average and Amortized Analysis - GeeksforGeeks
- Analysis of algorithms - Wikipedia
- Amortized analysis - Wikipedia

Complexité moyenne et adversariale : Une distinction fondamentale

Complexité moyenne et adversariale

Lors de l'analyse d'algorithmes, il est fondamental de comprendre deux approches principales de mesure de la complexité :

- La complexité moyenne (**average-case complexity**)
- La complexité adversariale (**worst-case complexity**)

Ces notions offrent des perspectives différentes sur la difficulté des problèmes et la performance des algorithmes.

Complexité adversariale (Worst-case complexity)

- **Définition** : Mesure la performance maximale d'un algorithme sur toutes les entrées possibles de taille (n).
- **Caractéristique** : Donne une borne supérieure absolue, garantissant que jamais l'algorithme ne dépassera ce temps.
- **Contextes d'utilisation** : Très utilisée en théorie, en particulier dans des contextes où garantir un temps maximum d'exécution est nécessaire (systèmes embarqués, sécurité).

Exemple : Le tri rapide (Quicksort) a un pire cas (\mathcal{O}(n^2)), lorsqu'on trie une liste déjà triée selon un mauvais pivot.

Complexité moyenne (Average-case complexity)

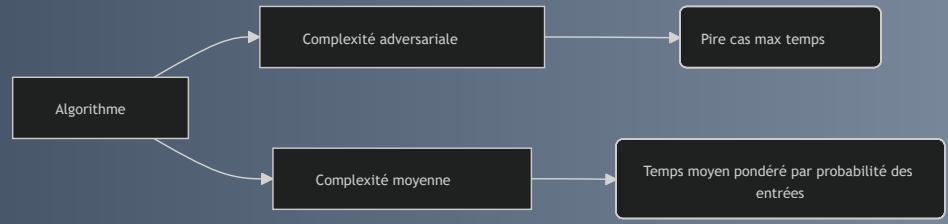
- **Définition** : Mesure attendue sur la distribution des entrées d'une taille fixe, considérant chaque entrée avec une probabilité donnée.
- **Caractéristique** : Reflète typiquement la performance observée en pratique si la distribution des données est bien modélisée.
- **Difficulté** : Nécessite un modèle probabiliste réaliste des entrées — souvent supposé uniforme, ce qui peut ne pas correspondre aux données réelles.

Exemple : Quicksort : complexité moyenne (\mathcal{O}(n \log n)) si les pivots sont choisis aléatoirement ou si la permutabilité des données est supposée.

Points clés de distinction

Aspect	Complexité adversariale	Complexité moyenne
Assurance	Garantie sur toutes les entrées	Moyenne selon une distribution spécifique
Modélisation des entrées	Aucune	Cruciale, hypothèses sur la distribution
Robustesse	Très robuste, pessimiste	Sensible aux hypothèses
Usage	Contextes critiques, théorique	Analyse statistique, performances pratiques

Illustration synthétique



Exemples concrets

Recherche dans une liste non triée

- **Adversariale** : $\mathcal{O}(n)$ car l'élément peut être absent ou en fin de liste.
- **Moyenne** : $\mathcal{O}(n/2)$, soit toujours linéaire mais deux fois plus petit en moyenne si éléments uniformément distribués.

Table de hachage

- **Adversariale** : Dans le pire cas, insertion et recherche peuvent coûter $\mathcal{O}(n)$ (si collisions massives).
- **Moyenne** : Temps constant $\mathcal{O}(1)$ en moyenne, sous hypothèses de bonne répartition des clés.

Sources utilisées

- Average-case complexity — Wikipedia
- Worst-case vs Average-case Complexity Explained - GeeksforGeeks
- Complexity theory, MIT OpenCourseWare
- Motwani R., Raghavan P., “Randomized Algorithms”, Cambridge University Press, 1995.

En résumé

La distinction entre complexité moyenne et adversariale est fondamentale. Elle permet d’évaluer la performance d’algorithmes selon différents points de vue :

- En fonction des garanties recherchées.
- En fonction de la nature des données traitées.

2 - Méthodes d’analyse de la complexité sous adversaire

Qu’est-ce que l’analyse sous adversaire ?

- **Définition**
  - Évaluation de la performance d’un algorithme dans la situation la plus défavorable possible (le "pire cas").
- **Objectif**
  - Garantir des bornes maximales sur les ressources utilisées.
  - Indépendance des hypothèses sur la distribution des entrées.
- **Approche**
  - Analyse rigoureuse par diverses méthodes.

1. Méthodes d’analyse sous adversaire : L’Analyse directe

- **Principe**
  - Étude explicite du comportement de l’algorithme sur l’entrée qui produit le pire temps d’exécution.
- **Utilisation**
  - Approche classique.
  - Calcule une borne supérieure par dénombrement d’opérations coûteuses.
- **Exemple**
  - **Recherche linéaire** : Le pire cas survient lorsque l’élément recherché est absent (examen de tous les éléments).
  - **Complexité** : (\mathcal{O}(n)).

2. Méthodes d’analyse sous adversaire : La Méthode de l’adversaire (Adversarial Argument)

- **Principe**
  - Un adversaire imagine et choisit dynamiquement l’entrée la plus défavorable possible.
  - Les choix de l’adversaire sont faits en réponse aux décisions de l’algorithme.

- **Utilité**
  - Démontre souvent que l’algorithme ne peut pas être meilleur qu’une certaine borne inférieure.

- **Exemples**
  - **Recherche binaire sur séquence triée** : L’adversaire adapte la réponse ("plus grand" ou "plus petit") pour maximiser le nombre de comparaisons ((\Omega(\log n))).
  - **Tri par comparaisons** : L’adversaire construit une séquence d’entrée imposant (\Omega(n \log n)) comparaisons.

3. Méthodes d’analyse sous adversaire : L’Analyse amortie à adversaire

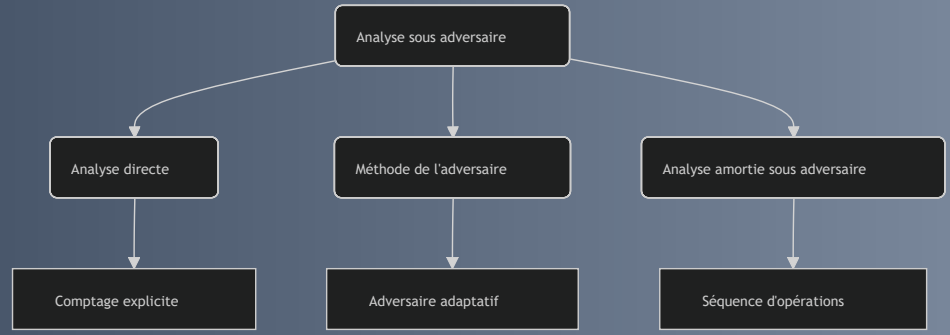
- **Principe**
  - Prend en compte des exécutions où un pire cas ponctuel est compensé par d’autres opérations plus rapides sur une séquence d’opérations.
  - Le coût est "amorti" sur l’ensemble des opérations.

- **Exemple**
  - **Opérations sur structures dynamiques** (tas, tableaux dynamiques) : La suppression ou l’ajout prend souvent un coût moyen plus faible que le pire cas isolé si l’on considère une série d’opérations.

Synthèse des méthodes et exemples précis

Méthode	Exemples	Description de l’analyse
Analyse directe	Recherche linéaire, tri fusion	Calcul du temps maximal explicitement
Méthode de l’adversaire	Recherche binaire, tri par comparaisons	Modélisation du choix de l’entrée la pire possible en réponse à l’algorithme
Analyse amortie sous adversaire	Table de hachage dynamique, pile dynamique	Moyenne sur une séquence d’opérations confrontée à un adversaire

Visualisation des méthodes d’analyse



Intérêt et Limites de l’analyse sous adversaire

- **Avantages**
  - Garantit une performance minimale quelle que soit l’entrée.
  - Outil essentiel pour prouver des bornes inférieures.

- **Limites**
  - Peut être pessimiste et ne pas refléter la performance en situation réelle.
  - Souvent difficile à appliquer sans simplifications.
  - Parfois complexe d’identifier une entrée réellement "pire" pour tous les algorithmes.

Sources utilisées

- Worst-case complexity — Wikipedia
- Adversarial argument - Complexity Zoo
- Amortized Analysis and the Accounting Method, CLRS
- Algorithmic Lower Bounds via an Adversary Argument

En résumé

L’analyse de la complexité sous adversaire permet de caractériser les performances des algorithmes dans leurs cas les plus défavorables. Elle s’appuie sur des stratégies rigoureuses, telles que l’analyse directe, la méthode de l’adversaire ou l’analyse amortie, pour établir des bornes de performance robustes.

Rappels et Introduction

Optimisation et Gestion Mémoire

Choix de structures de données : Tableau vs Liste Chaînée

Comprendre leurs compromis pour des implémentations efficaces

Le choix d'une structure de données impacte directement la performance, la gestion mémoire et la simplicité d'implémentation. Deux structures fondamentales sont le **tableau (array)** et la **liste chaînée (linked list)**.

Définitions

- **Tableau (Array) :**
  - Contigu en mémoire.
  - Accès direct en temps constant (O(1)) via index.
  - Taille généralement fixée à la création.
- **Liste chaînée (Linked List) :**
  - Chaîne d'éléments (noeuds) reliés par des pointeurs/références.
  - Taille dynamique.
  - Insertion/suppression aisée.

Tableau vs Liste Chaînée : Comparaison Détaillée

Critère	Tableau	Liste chaînée
Accès aléatoire	(O(1)) — accès direct par index	(O(n)) — doit parcourir la liste
Insertion	(O(n)) — décalage possible	(O(1)) — simple ré-affectation de pointeurs
Suppression	(O(n)) — décalage possible	(O(1)) — déconnexion du noeud
Gestion mémoire	Mémoire contiguë, overhead faible	Overhead mémoire important (pointeurs)
Taille	Fixe ou redimensionnement coûteux	Dynamique, flexible
Localité mémoire	Excellente, favorise cache CPU	Fragile — discontinuité mémoire
Complexité de la structure	Simple	Complexe (pointeurs, gestion des liens)

Exemples Concrets : Performances des Opérations

Accès direct dans un tableau

```c
int tab[5] = {1,2,3,4,5};
int val = tab[3]; // Accès direct en O(1)
```

- Accès ultra-rapide grâce à la contiguïté mémoire.

Insertion en début d’une liste chaînée

```c
typedef struct Node {
    int data;
    struct Node* next;
} Node;

void inserer_debut(Node** head, int val) {
    Node* nouveau = malloc(sizeof(Node));
    nouveau->data = val;
    nouveau->next = *head;
    *head = nouveau;
}
```

- Insérer un élément en début se fait en temps (O(1)), sans déplacement d’éléments.

Illustration Mémoire : Contiguïté vs Liens

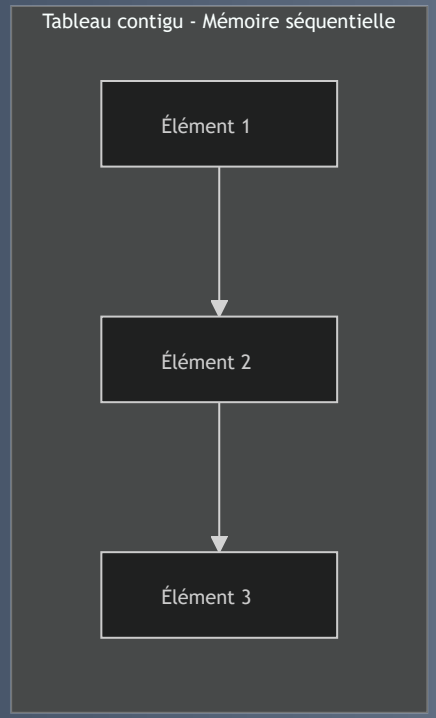
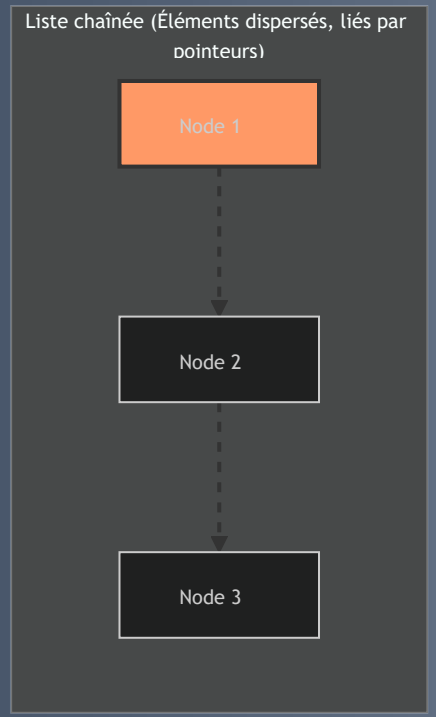


Illustration Mémoire : Contiguïté vs Liens



- **Tableau :** Les éléments sont adjacents, optimisant le cache CPU.
- **Liste chaînée :** Les éléments peuvent être dispersés en mémoire, nécessitant des pointeurs.

Choix Stratégique selon les Cas d’Usage

- **Quand privilégier le Tableau :**
  - La taille est connue ou ne varie que peu.
  - Accès fréquent par index (recherche rapide).
  - Exécution nécessitant rapidité mémoire et CPU (ex : calcul matriciel, vecteurs).
- **Quand privilégier la Liste Chaînée :**
  - La taille varie fréquemment (ajouts/suppressions régulières).
  - Les insertions/suppressions sont fréquentes, mais l’accès par index est rare.
  - Exemples : files d’attente (queues), piles (stacks), gestion dynamique de données.

Conclusion et Ressources

Ce qu’il faut retenir

Privilégier le tableau ou la liste chaînée dépend des besoins en accès rapide, gestion dynamique de la mémoire et fréquence des opérations d’insertion/suppression. Une bonne compréhension des compromis inspire des choix optimisés en algorithmique avancée.

Sources et lectures complémentaires

- GeeksforGeeks – Array vs Linked List
- Medium – When to use Linked List over Array
- Wikipedia – Data Structure
- Guru99 – Arrays and Linked Lists
- Computer Science Field Guide – Arrays and Linked Lists

Rappels : Pile et Tas (Stack et Heap)

Gestion Mémoire Avancée

- Dans la gestion mémoire d’un programme (langage C et la plupart des langages compilés), deux régions clés stockent les données :
  - **La pile (stack)**
  - **Le tas (heap)**
- Comprendre leurs caractéristiques permet d’optimiser la gestion mémoire, la sécurité et la performance.

La Pile (Stack) : Mémoire Automatique

Description & Comportement

- Zone mémoire **statique** et **structurée** en mode LIFO (Last In, First Out).
- Gère les variables locales, les paramètres de fonctions et le contexte d’appel.
- Allocation et désallocation automatiques à l’entrée et sortie de blocs/fonctions.

Caractéristiques clés

Caractéristique	Description
Allocation	Automatique
Taille	Limitée (~1Mo à quelques Mo)
Accès	Très rapide, adressage contigu
Durée de vie	Cycle d’une fonction/portée locale
Sécurité	Risque de débordement (stack overflow)

Le Tas (Heap) : Mémoire Dynamique

Description & Comportement

- Zone mémoire **dynamique** gérée manuellement via des appels d'allocation (`malloc`, `calloc`) et de libération (`free`).
- Permet d'allouer des espaces mémoire dont la taille peut être décidée à l'exécution.

Caractéristiques clés

Caractéristique	Description
Allocation	Manuelle (via malloc/free)
Taille	Plus grande, limitée surtout par mémoire physique disponible
Accès	Plus lent que la pile (fragmentation, non contiguë)
Durée de vie	Contrôlée par le programme
Sécurité	Risque de fuites mémoire, déréférencement

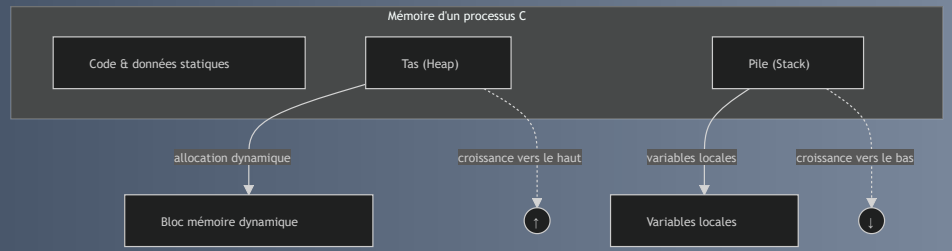
Pile vs Tas

Choix Stratégique selon les Besoins

Critère	Pile (Stack)	Tas (Heap)
Allocation	Automatique	Manuelle (malloc/free)
Vitesse d'allocation	Très rapide	Plus lente
Taille	Limitée	Plus importante
Fragmentation	Pas de fragmentation	Possible fragmentation mémoire
Durée de vie des données	Limité à la portée locale	Contrôlable globalement
Erreurs fréquentes	Débordement pile	Fuites mémoire, double free

Visualisation de la Gestion Mémoire

Comment les deux régions interagissent



Synthèse & Bonnes Pratiques

Optimisation et Stabilité des Programmes

- Pour des allocations temporaires et taille connue, privilégier la **pile** (variables locales).
- Pour des données dont la taille est dynamique ou persistante, utiliser le **tas**.
- Toujours libérer la mémoire allouée sur le tas pour éviter les fuites (ex.: utiliser `free`).
- Attention aux dépassements de pile (variables trop volumineuses) et aux erreurs sur le tas (déréférencement de pointeurs invalides, fuites).

Sources consultées

- GeeksforGeeks – Stack vs Heap
- Medium – Understanding Stack and Heap Memory in C
- Wikipedia – Stack (computer science)
- Wikipedia – Heap (data structure)
- TutorialsPoint – Stack vs Heap

Structures de données dynamiques avancées

Listes Doublement Chaînées et Circulaires

Les listes doublement chaînées offrent une manipulation efficace des éléments grâce à un accès direct et bidirectionnel. Elles permettent des opérations d'insertion, de suppression et de parcours en tête, en queue ou à une position arbitraire, avec une complexité temporelle optimale.

Structure d'un Nœud

Un nœud dans une liste doublement chaînée contient la donnée, un pointeur vers le nœud précédent (`prev`) et un pointeur vers le nœud suivant (`next`). Un pointeur vers la tête de la liste est essentiel pour son management.

```c
typedef struct Node {
    int data;
    struct Node *prev;
    struct Node *next;
} Node;
```

Opérations d’Insertion

L’insertion d’un élément nécessite la création d’un nouveau nœud et la mise à jour des pointeurs `prev` et `next` des nœuds adjacents.

1. Insertion en tête

1. Créer un nouveau nœud.
2. Le lier à l’ancienne tête (`nouveau→next = *head`).
3. Mettre à jour le pointeur `prev` de l’ancienne tête vers le nouveau nœud.
4. Le nouveau nœud devient la nouvelle tête (`*head = nouveau`).

```c
void inserer_tete(Node **head, int val) {
    Node *nouveau = malloc(sizeof(Node));
    if (!nouveau) return;
    nouveau→data = val;
    nouveau→prev = NULL;
    nouveau→next = *head;

    if (*head) (*head)→prev = nouveau;
    *head = nouveau;
}
```

2. Insertion en queue

Parcourt la liste jusqu'à la fin, puis lie le nouveau nœud au dernier, en gérant le cas de liste vide.

```c
void inserer_queue(Node **head, int val) {
    Node *nouveau = malloc(sizeof(Node));
    if (!nouveau) return;
    nouveau->data = val;
    nouveau->next = NULL;

    if (*head == NULL) {
        nouveau->prev = NULL;
        *head = nouveau;
        return;
    }

    Node *temp = *head;
    while (temp->next) temp = temp->next;

    temp->next = nouveau;
    nouveau->prev = temp;
}
```

3. Insertion à une position donnée

Parcourt la liste jusqu'à la position `pos`, puis insère le nouveau nœud **avant** le nœud à cette position, ajustant les pointeurs autour.

```c
void inserer_position(Node **head, int val, int pos) {
    if (pos == 0) {
        inserer_tete(head, val);
        return;
    }

    Node *temp = *head;
    for (int i = 0; i < pos - 1 && temp != NULL; i++)
        temp = temp->next;

    if (!temp) return; // position invalide

    Node *nouveau = malloc(sizeof(Node));
    if (!nouveau) return;
    nouveau->data = val;

    nouveau->next = temp->next;
    if (temp->next)
        temp->next->prev = nouveau;

    temp->next = nouveau;
    nouveau->prev = temp;
}
```

Opérations de Suppression

La suppression d’un élément implique la déconnexion du nœud de la liste et la libération de sa mémoire.

1. Suppression en tête

1. Vérifier que la liste n’est pas vide.
2. Le nœud à supprimer est l’actuelle tête.
3. La nouvelle tête devient le nœud suivant (`*head = temp→next`).
4. Si la nouvelle tête existe, son pointeur `prev` est mis à NULL.
5. Libérer la mémoire de l’ancien nœud tête.

```
void supprimer_tete(Node **head) {
    if (!head || !*head) return;

    Node *temp = *head;
    *head = temp→next;

    if (*head)
        (*head)→prev = NULL;

    free(temp);
}
```

2. Suppression en queue

Parcourt la liste jusqu'au dernier nœud, déconnecte ce dernier de son précédent, puis libère sa mémoire.

```c
void supprimer_queue(Node **head) {
    if (!head || !*head) return;

    Node *temp = *head;
    if (!temp->next) { // un seul nœud
        free(temp);
        *head = NULL;
        return;
    }

    while (temp->next) temp = temp->next;

    temp->prev->next = NULL;
    free(temp);
}
```

3. Suppression à une position donnée

Parcourt la liste jusqu'à la position `pos`, déconnecte le nœud correspondant en ajustant les pointeurs `next` et `prev` de ses voisins, puis libère sa mémoire.

```c
void supprimer_position(Node **head, int pos) {
    if (!head || !*head) return;
    Node *temp = *head;

    for (int i = 0; i < pos && temp != NULL; i++)
        temp = temp->next;

    if (!temp) return;

    if (temp->prev)
        temp->prev->next = temp->next;
    else
        *head = temp->next;

    if (temp->next)
        temp->next->prev = temp->prev;

    free(temp);
}
```

Parcours de la Liste et Vue d’Ensemble

Les listes doublement chaînées peuvent être parcourues dans les deux sens, offrant une flexibilité accrue.

1. Parcours en avant

Traverse la liste de la tête à la queue via le pointeur `next`.

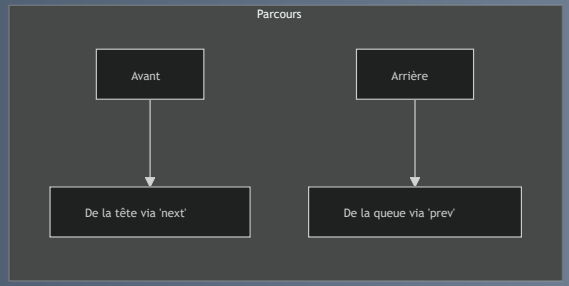
```
void afficher_avant(Node *head) {
    Node *temp = head;
    while (temp) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
```

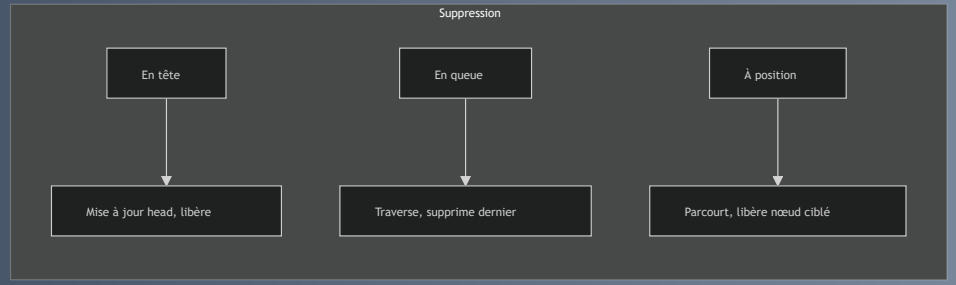
2. Parcours en arrière

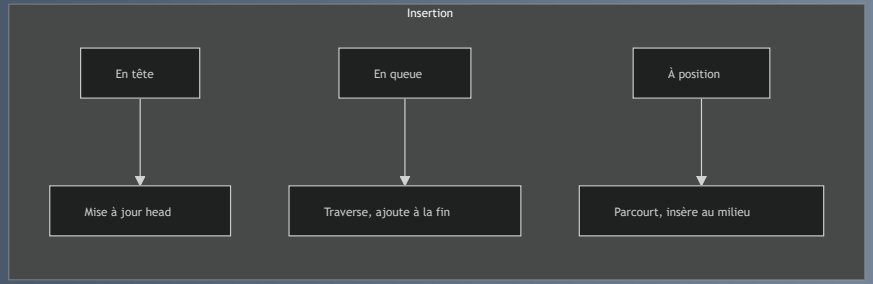
Nécessite de trouver le dernier nœud, puis de remonter vers la tête via le pointeur `prev`.

```c
void afficher_arriere(Node *head) {
    if (!head) return;
    Node *temp = head;
    while (temp->next) temp = temp->next; // Aller à la fin

    while (temp) { // Remonter
        printf("%d ", temp->data);
        temp = temp->prev;
    }
    printf("\n");
}
```

Schéma récapitulatif des opérations principales







Listes Doublement Chaînées Circulaires

Dans une liste circulaire, le pointeur `next` du dernier nœud pointe vers la tête, et le pointeur `prev` de la tête pointe vers le dernier nœud.

Détails pour les listes circulaires :

- Le test `temp→next == NULL` est remplacé par `temp→next ≠ head` pour détecter la fin.
- Toutes les opérations (insertion, suppression, parcours) doivent gérer cette "boucle infinie" pour maintenir la cohérence de la structure.

Exemple d’insertion en queue (circulaire) :

```c
void inserer_queue_circ(Node **head, int val) {
    Node *nouveau = malloc(sizeof(Node));
    nouveau->data = val;

    if (*head == NULL) { // Si la liste est vide
        nouveau->next = nouveau->prev = nouveau; // Pointe sur lui-même
        *head = nouveau;
        return;
    }

    Node *tail = (*head)->prev; // Le dernier nœud

    tail->next = nouveau;      // Le précédent dernier pointe vers le nouveau
    nouveau->prev = tail;      // Le nouveau pointe vers l'ancien dernier
    nouveau->next = *head;     // Le nouveau pointe vers la tête
    (*head)->prev = nouveau;   // La tête pointe vers le nouveau comme précédent
}
```

Ce qu’il faut retenir et Sources

Ce qu’il faut retenir :

Les opérations sur les listes doublement chaînées impliquent une gestion attentive des pointeurs `prev` et `next` pour maintenir la cohérence des liens. La maîtrise de ces opérations est indispensable pour manipuler efficacement ces structures dynamiques, aussi bien en mode linéaire que circulaire.

Sources consultées :

- GeeksforGeeks — Doubly Linked List Overview and Operations
- Programiz — Doubly Linked List in C
- TutorialsPoint — Doubly Linked List Operations
- Wikipedia — Doubly linked list

Listes Chaînées : Rappels et Fondamentaux

Introduction aux structures dynamiques

- **Liste simplement chaînée :** Chaque nœud contient une donnée et un pointeur vers le nœud *suivant*.
- **Liste doublement chaînée :** Chaque nœud contient une donnée, un pointeur vers le nœud *suivant*, et un pointeur vers le nœud *précédent*.
- **Listes circulaires :** Le dernier nœud pointe vers le premier, pouvant être simplement ou doublement chaînées.

Les Atouts des Listes Doublement Chaînées

Pourquoi choisir la bidirectionnalité ?

- **Navigation bidirectionnelle :** Le pointeur `prev` permet de parcourir la liste en avant et en arrière, facilitant la recherche inverse ou l'itération depuis la fin.
- **Suppression simplifiée :** La suppression d'un nœud (hors tête) est plus efficace car l'accès au nœud précédent est direct, sans parcours initial.
- **Insertion plus flexible :** L'insertion avant ou après un nœud est directe, même avec seulement un pointeur vers ce nœud.
- **Support des listes circulaires :** Elles facilitent le parcours continu (rotatif), idéal pour les anneaux ou buffers.
- **Base de structures avancées :** Des structures comme les déques (double-ended queues) reposent sur cette bidirectionnalité.

Inconvénients et Compromis des Listes Doublement Chaînées

Le prix de la flexibilité

Inconvénients	Explications
Coût mémoire accru	Chaque nœud stocke un pointeur supplémentaire (`prev`), augmentant la consommation mémoire.
Complexité de gestion	La gestion de `prev` et `next` est plus complexe, augmentant les risques d'erreurs.
Performances moindres	Pour un parcours avant uniquement, une liste simple est plus légère et rapide en cache.
Opérations plus coûteuses	Insertion et suppression impliquent la mise à jour de deux pointeurs.

Comparaison Rapide : Simple vs. Doublement Chaînée

Un choix stratégique selon les besoins

Critère	Liste simplement chaînée	Liste doublement chaînée
Stockage	1 pointeur par nœud (next)	2 pointeurs par nœud (prev + next)
Navigation	Unidirectionnelle	Bidirectionnelle
Suppression	Nœud précédent non accessible	Nœud précédent accessible directement
Insertion	Pas d’accès facilité avant un nœud	Accès flexible avant et après un nœud
Complexité code	Plus simple	Plus complexe
Usage mémoire	Moins	Plus
Adaptation Algorithmes	Moins adaptée (ex: parcours inverse)	Plus adaptée (ex: déques, listes circulaires)

Cas Pratique : Efficacité de la Suppression

Suppression d'un nœud donné (hors tête) dans une liste simplement chaînée :

- Nécessite de parcourir la liste depuis le début pour trouver le nœud précédent.
- Coût en temps : (O(n)).

```c
void supprimerNode(Node **head, Node *node) {
    if (*head == node) {
        *head = (*head)->next;
        free(node);
        return;
    }

    Node *temp = *head;
    while (temp && temp->next != node) {
        temp = temp->next;
    }

    if (temp == NULL) return;

    temp->next = node->next;
    free(node);
}
```

Dans une liste doublement chaînée :

- Accès direct au nœud précédent via `node→prev`.
- Pas de parcours nécessaire.
- Coût en temps : (O(1)).

```c
void supprimerNodeD(Node **head, Node *node) {
    if (node→prev)
        node→prev→next = node→next; // Le nœud précédent pointe vers le suivant
    else
        *head = node→next; // Le nœud était la tête

    if (node→next)
        node→next→prev = node→prev; // Le nœud suivant pointe vers le précédent

    free(node);
}
```

Décision Stratégique & Ressources Complémentaires

Ce qu'il faut retenir :

La sélection entre liste simplement ou doublement chaînée dépend des exigences fonctionnelles : la bidirectionnalité apporte plus de flexibilité et efficacité dans certains cas, au prix d'une surcharge mémoire et d'une gestion de pointeurs plus minutieuse.

Sources consultées :

- GeeksforGeeks — Difference between Singly and Doubly Linked List
- TutorialsPoint — Singly vs Doubly Linked Lists
- Wikipedia — Doubly linked list
- Programiz — Doubly Linked List

Tables de Hachage

La table de hachage : Accès rapide aux données

- Structure de données dynamique permettant d'associer **clefs et valeurs**.
- Offre un accès proche de **(O(1)) en moyenne**.
- Repose sur une **fonction de hachage** qui transforme une clef en un **indice** dans un tableau.

Fonctionnement résumé :

1. Calcul d'un **hachage** (entier) de la clé.
2. Cet entier sert d'**indice** pour stocker ou récupérer la valeur.
3. Gestion des **collisions** (plusieurs clefs même hachage) par des techniques spécifiques.

1. Principe du Hachage & 2. Fonction de Hachage

Principe du hachage :

- Transforme une clef en un indice unique (si possible) pour un accès direct.
- Nécessite une gestion efficace des cas où différentes clefs génèrent le même indice (collisions).

Fonction de hachage :

- Prend une clef (k) et renvoie un indice dans une table de taille (m).
- **Propriétés recherchées :**
  - **Uniformité** : Répartition homogène des indices pour minimiser les collisions.
  - **Rapidité** : Calcul efficace et rapide de l'indice.
  - **Déterminisme** : Même clef doit toujours retourner la même valeur.
  - **Sensibilité aux différences** : Clefs légèrement différentes doivent produire des hachages distincts.

3. Exemples de fonctions de hachage

Pour des entiers :

- Une fonction simple et courante : [h(k) = k \mod m] où (m) est la taille de la table (souvent un nombre premier pour une meilleure répartition).

Pour des chaînes de caractères : Le hash DJB2

- Se base sur un calcul cumulatif pondéré des caractères.
- Exemple de principe (DJB2) :

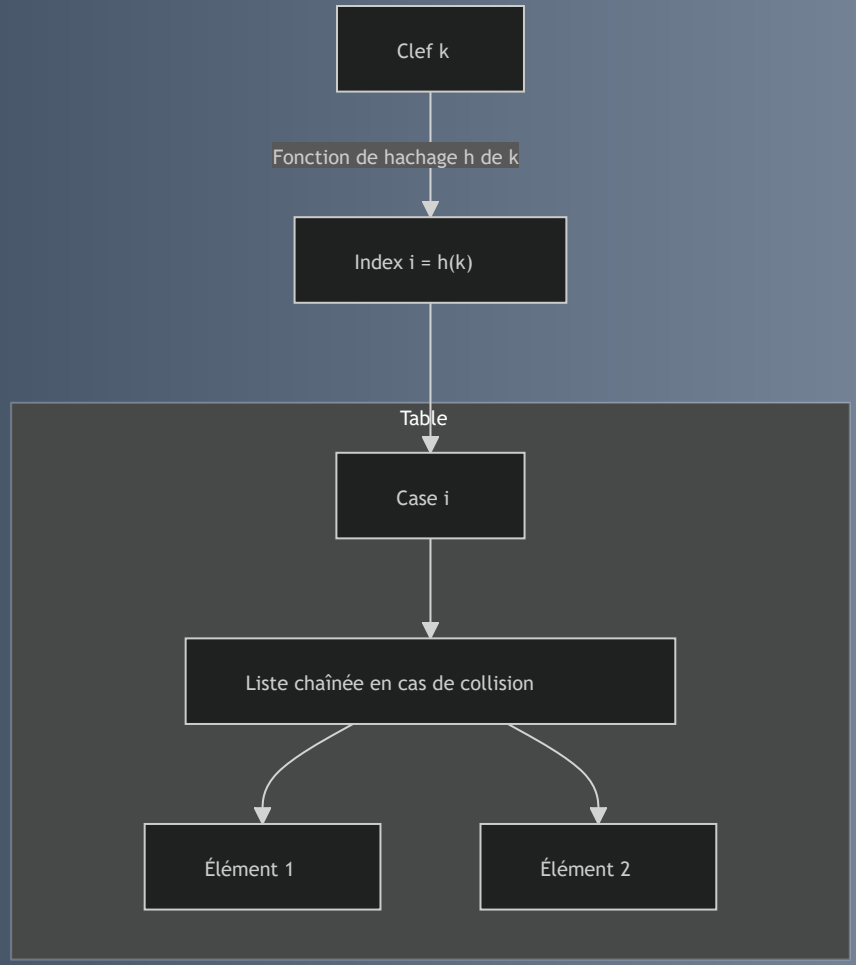
```c
unsigned long djb2(const char *str) {
    unsigned long hash = 5381; // Valeur initiale
    int c;
    while ((c = *str++))
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
    return hash;
}
```

- Cette fonction est simple, rapide et offre une bonne uniformité en pratique.

4. Gestion des collisions

Les collisions sont inévitables et doivent être gérées efficacement pour maintenir les performances (O(1)).

- **Chaînage** :
  - Chaque case de la table contient une liste (chaînée) d'éléments.
  - En cas de collision, l'élément est ajouté à la liste correspondant à l'indice.
- **Sondage (Probing)** :
  - En cas de collision, on cherche une autre case disponible.
  - Exemples : sondage linéaire, quadratique, double hachage.


Clef k
Fonction de hachage h de k
Index i = h(k)
Table
Case i
Liste chaînée en cas de collision
Élément 1
Élément 2

Exemple d’utilisation simple d’une table de hachage avec chaînage

```c
#define TABLE_SIZE 101 // Taille de la table

typedef struct Node {
    char *key;
    int value;
    struct Node *next;
} Node;

Node *hash_table[TABLE_SIZE]; // Le tableau des "cases" (têtes de listes)

// Fonction de hachage (DJB2 simplifié modulo TABLE_SIZE)
unsigned int hash(char *key) {
    unsigned long hash_val = 5381;
    int c;
    while ((c = *key++))
        hash_val = ((hash_val << 5) + hash_val) + c;
    return hash_val % TABLE_SIZE; // Retourne l'indice dans la table
}
// Insertion simple avec chaînage
void insert(char *key, int value) {
    unsigned int idx = hash(key); // Calcul de l'indice
    Node *new_node = malloc(sizeof(Node)); // Allocation d'un nouveau nœud
    new_node->key = strdup(key);
    new_node->value = value;
    new_node->next = hash_table[idx]; // Le nouveau nœud pointe vers l'ancienne tête de liste
    hash_table[idx] = new_node;       // Le nouveau nœud devient la tête de liste
}
```

Limites et considérations & Ce qu’il faut retenir

Limites importantes :

- Une **fonction de hachage mal conçue** génère de nombreuses collisions, dégradant les performances.
- La **taille de la table** est cruciale (facteur de charge (\alpha = \frac{n}{m})) et doit être **redimensionnée** en cas de surcharge.
- La **synchronisation** est nécessaire en contexte multithread pour éviter les incohérences.

Ce qu’il faut retenir :

- Le hachage et les fonctions de hachage sont des piliers pour un accès rapide aux données.
- Ils transforment les clés en indices mémoire de manière calculable et reproductible.

8. Sources consultées :

- GeeksforGeeks — Hashing Data Structure
- Wikipedia — Hash function
- Wikipedia — Hash table
- DJB2 Hash Function Explanation (Stack Overflow)
- Programiz — Hash Table and Hash Functions

Le Défi des Collisions

Une **collision** survient quand deux clefs différentes produisent la même valeur de hachage, pointant vers un même indice de la table.

La gestion efficace de ces collisions est **essentielle pour préserver la rapidité d'accès** des tables de hachage.

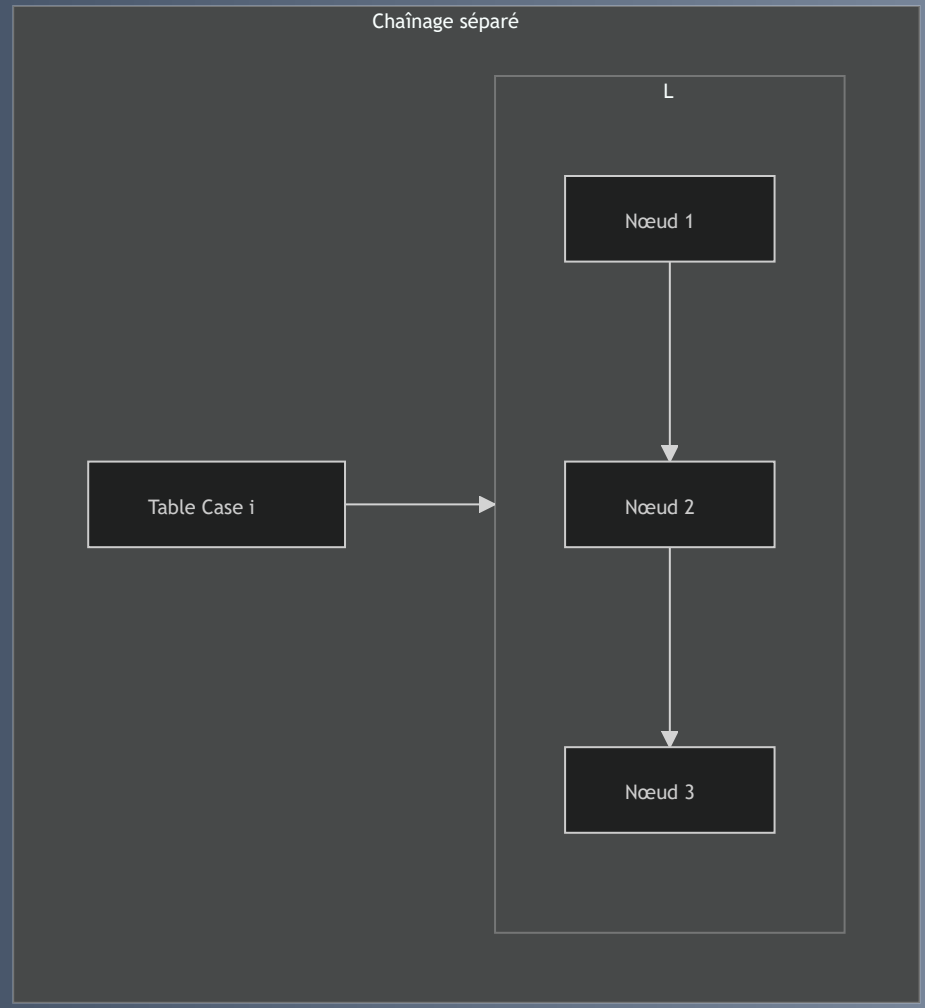
Méthode 1 : Le Chaînage séparé

Principe

Chaque case de la table contient une **liste chaînée** d'éléments. En cas de collision, les nouveaux éléments sont ajoutés à la liste de la case concernée.

Avantages

- **Facile à implémenter**.
- **Pratique** en cas de facteur de charge > 1 (plus d'éléments que de cases).
- **Performant** même dans le pire des cas (la recherche traverse uniquement la liste).
- **Complexité moyenne :** (O(1 + \alpha)), où (\alpha) est le facteur de charge.


Chaînage séparé
L
Nœud 1
Table Case i
Nœud 2
Nœud 3

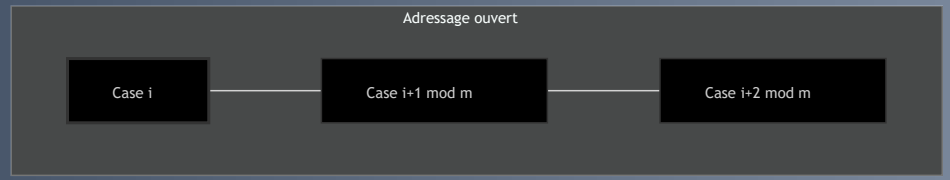
3. Méthode 2 : L’Adressage Ouvert

Principe

Les éléments sont stockés **directement dans la table**, sans listes externes. En cas de collision, on recherche une autre case "libre" selon une stratégie de **sondage**.

Stratégies principales :

- **Sondage linéaire**
- **Sondage quadratique**
- **Double hachage**



Adressage Ouvert : Les Stratégies de Sondage

1 Sondage linéaire

- Cherche la première case suivante vide (à l'indice `(h(k)+i) mod m`).
- **Inconvénient :** Peut conduire à des regroupements dits "**clustering primaire**", dégradant les performances.

```
int linear_probing_insert(int table[], int m, int key) {
    int idx = hash(key);
    int i = 0;
    while (table[(idx + i) % m] ≠ EMPTY) {
        i++;
    }
    table[(idx + i) % m] = key;
    return (idx + i) % m;
}
```

2 Sondage quadratique

- Cherche la prochaine case testée selon `(h(k) + i^2) mod m`.
- **Avantage :** Réduit le clustering primaire.
- **Inconvénient :** Peut générer du "**clustering secondaire**".

```c
int quadratic_probing_insert(int table[], int m, int key) {
    int idx = hash(key, m);
    int i = 0;

    while (table[(idx + i * i) % m] != EMPTY) {
        i++;
        if (i == m) { // table pleine
            printf("Table pleine, impossible d'insérer %d\n", key);
            return -1;
        }
    }

    table[(idx + i * i) % m] = key;
    return (idx + i * i) % m;
}
```

3 Double hachage

- Utilise une deuxième fonction de hachage (h_2(k)) pour décaler plus efficacement en cas de collision.

Le principe :

- On calcule deux fonctions de hashage :
- -h1(key) → position de départ.
- -h2(key) → pas de déplacement en cas de collision.
- Si la case est occupée, on essaie :

$$(h1(key)+i\times h2(key)) \bmod m$$

où i est le nombre d'essais (0, 1, 2…).

Chaînage vs Adressage Ouvert

Méthode	Mémoire supplémentaire	Complexité insertion	Gestion facteur de charge élevé	Implementation
Chaînage séparé	Oui (listes externes)	(O(1 + \alpha))	Très bonne	Simple
Adressage linéaire	Non	(O(1)) (moyenne)	Réduit efficacité au-delà de 0.7	Simple
Adressage quadratique	Non	Plus élevé que linéaire	Meilleur que linéaire	Moyennement simple
Double hachage	Non	Très bonne	Très bonne	Plus complexe

Ce qu’il faut retenir & Références

Choix et limites

- Le chaînage séparé facilite les **redimensionnements dynamiques**.
- L’adressage ouvert demande un **facteur de charge < 1** pour éviter les performances dégradées.
- Le choix dépend du **contexte** : mémoire disponible, complexité, dimensionnement, type de clés.

La gestion des collisions est la pierre angulaire des tables de hachage robustes. En fonction des contraintes mémoire et d’efficacité, chaînage ou adressage ouvert offrent des compromis adaptés.

Sources consultées

- GeeksforGeeks — Collision Resolution Techniques
- Wikipedia — Hash table
- Programiz — Hash Table Collision Handling
- Open Addressing with Linear, Quadratic, Double Hashing

Tables de Hachage : Complexité des Opérations

Les tables de hachage offrent un **accès rapide en moyenne**, proche de (O(1)). Cependant, la complexité dépend fortement de deux facteurs essentiels :

1. Le comportement de la **fonction de hachage**.
2. La **gestion des collisions**.

Le Facteur de Charge (α)

Si la table a m = 10 cases et qu'on a inséré n = 7 éléments :

$$\alpha = 0.7$$

Cela signifie que la table est remplie à 70 %.

c'est le rapport entre le nombre d'éléments (n) stockés et la taille (m) de la table. Il indique la densité d'occupation.

- **(α) faible (ex : 0.5)** : Performances optimales.
- **(α) proche de 1 ou plus** : Dégradation des performances, surtout avec l'adressage ouvert.

Chaînage Séparé : Complexité des Opérations

Cette méthode de résolution des collisions stocke les éléments en listes chaînées pour chaque indice de table.

Opération	Complexité moyenne	Complexité pire cas
Insertion	(O(1))	(O(n)) (tous éléments dans une liste)
Recherche	(O(1 + α))	(O(n))
Suppression	(O(1 + α))	(O(n))

Explication

Les opérations accèdent à la liste chaînée correspondant à l'indice calculé. La complexité moyenne est linéaire en (α), tandis que le pire cas est linéaire en (n) si la fonction de hachage est très mauvaise (tous les éléments dans une seule liste).

Adressage Ouvert : Complexité des Opérations

En adressage ouvert, toutes les entrées sont stockées directement dans la table, et les collisions sont gérées par des séquences de sondage.

Opération	Complexité
Insertion	O(1 / 1−α)
Recherche	O(1 / 1−α)
Suppression	O(1 / 1−α)

Explication

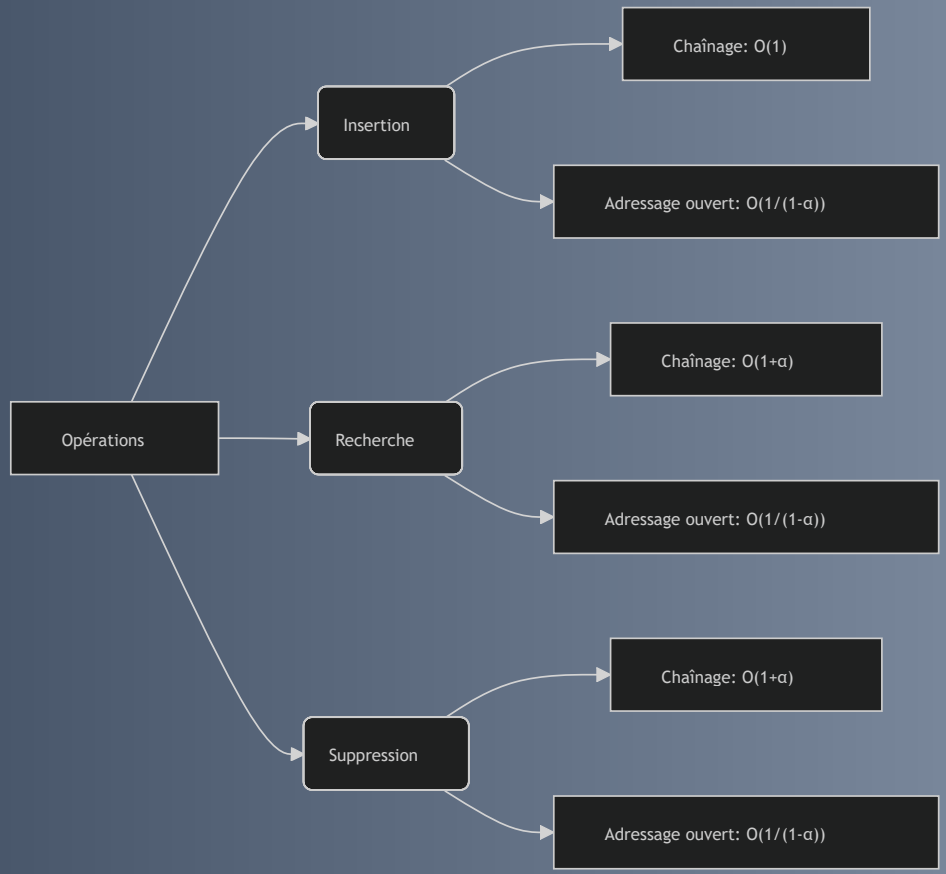
- Plus le facteur de charge (α) se rapproche de 1, plus le temps moyen d'exécution augmente de façon significative.
- Les méthodes de sondage (linéaire, quadratique, double hachage) ont un impact direct sur ces performances.

Exemples concrets

Recherche en chaînage Dans une table de taille 10 avec 5 éléments ($\alpha=0.5$), une recherche moyenne inspecte environ $1 + 0.5 = 1.5$ éléments par case.

Insertion en adressage linéaire Si $\alpha=0.8$, le nombre moyen de sondages avant insertion est d'environ $\frac{1}{1-0.8} = 5$. Cela représente 5 accès mémoire en moyenne avant l'insertion.

Complexité des opérations par type de gestion des collisions



L’Impact du choix de la fonction de hachage

- Une **fonction de hachage bien conçue** limite le nombre de collisions, réduisant ainsi le (\alpha) effectif et optimisant les performances moyennes.
- Un **mauvais hachage** concentre les éléments dans quelques cases, dégradant la complexité moyenne vers le pire cas (O(n)).

Clés de la Performance et Ressources

Ce qu’il faut retenir

La réalité d’une table de hachage performante repose sur :

1. Le **contrôle du facteur de charge** (\alpha).
2. La **qualité de la fonction de hachage**.

Ces deux paramètres déterminent directement le coût en temps moyen et en pire cas des opérations d’insertion, recherche et suppression.

Sources consultées

- GeeksforGeeks — Time Complexity of Hashing
- Wikipedia — Hash Table#Complexity_of_hash_tables
- Programiz — Hash Table Analysis
- MIT Lecture Notes — Hashing Performance

Arbres Binaires de Recherche (ABR)

1. Qu’est-ce qu’un Arbre Binaire de Recherche (ABR) ?

Un **Arbre Binaire de Recherche (BST)** est une structure de données arborescente respectant des règles précises :

- Chaque nœud contient une **clé** (valeur).
- Pour tout nœud (N):
  - Toutes les clés du **sous-arbre gauche** sont **strictement inférieures** à la clé de (N).
  - Toutes les clés du **sous-arbre droit** sont **supérieures** à la clé de (N).
- Chaque nœud a **au plus deux enfants** (gauche et droit).

2. Propriétés Fondamentales et Leurs Conséquences

Les caractéristiques d'un ABR déterminent son comportement et son efficacité.

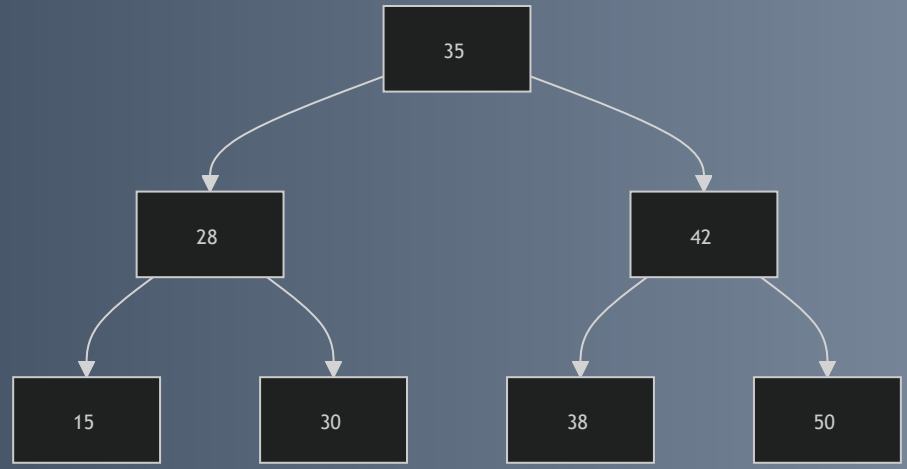
Propriété	Description
Recherche efficace	Opérations (recherche, insertion, suppression) en (O(h)) en moyenne, où (h) est la hauteur.
Ordre des clés	Le parcours infixe retourne les clés dans l'ordre croissant.
Hauteur variable	De (\log n) (équilibré) à (n) (dégénéré) pour (n) nœuds, selon l'insertion.
Structure récursive	Chaque sous-arbre est lui-même un ABR.

Conséquences :

- Algorithmes récursifs simples pour les opérations de base.
- La **complexité dépend fortement de la hauteur** de l'arbre.
- Les arbres non équilibrés dégradent les performances (temps linéaire, comme une liste chaînée).

3. Exemple Illustratif d’un ABR

Visualisons la construction d’un ABR après l’insertion successive des clés : **35, 28, 42, 15, 30, 38, 50**.



Ce diagramme montre comment les clés sont organisées pour respecter la propriété d’ordre des ABR.

4. Algorithme de Recherche dans un ABR

La recherche d'une clé dans un ABR tire parti de sa structure ordonnée, permettant une exploration rapide.

```c
Node* rechercher(Node *root, int key) {
    // Cas de base : arbre vide ou clé trouvée
    if (root == NULL || root->key == key)
        return root;

    // Si la clé est plus petite, chercher dans le sous-arbre gauche
    if (key < root->key)
        return rechercher(root->left, key);
    // Sinon, chercher dans le sous-arbre droit
    else
        return rechercher(root->right, key);
}
```

L'algorithme parcourt l'arbre de manière récursive, se dirigeant à gauche ou à droite en fonction de la comparaison avec la clé du nœud courant.

5. Parcours Infixe : Obtenir les Clés Triées

Le parcours infixe est une méthode fondamentale pour visiter les nœuds d'un ABR. Sa particularité est de retourner les clés dans un ordre croissant.

```c
void parcours_infixe(Node *root) {
    if (root != NULL) {
        // 1. Visiter le sous-arbre gauche
        parcours_infixe(root->left);
        // 2. Traiter le nœud courant
        printf("%d ", root->key);
        // 3. Visiter le sous-arbre droit
        parcours_infixe(root->right);
    }
}
```

Ce parcours est essentiel pour vérifier l'ordre des clés ou pour extraire une liste triée des éléments de l'arbre.

6. Points Cruciaux & Conclusion

Points à surveiller :

- L'**ordre d'insertion** détermine la forme de l'arbre et donc sa hauteur.
- L'**équilibrage** est indispensable pour garantir une complexité (O(\log n)) dans le pire cas (cf. arbres AVL, rouges-noirs).
- La **suppression** d'un nœud est complexe et doit gérer plusieurs cas (feuille, 1 enfant, 2 enfants).

Conclusion : Un ABR est une base essentielle en algorithmique servant de fondation pour des structures plus complexes. Son organisation garantit un accès ordonné efficace dès lors que sa hauteur est maîtrisée.

Sources consultées :

- GeeksforGeeks — Binary Search Tree (BST)
- Wikipedia — Binary Search Tree
- TutorialsPoint — Binary Search Tree
- Programiz — Binary Search Tree

Arbres Binaires de Recherche (ABR) : Les Fondamentaux

- **Qu’est-ce qu’un ABR ?**
  - Une structure de données conçue pour maintenir l’ordre.
  - Permet des opérations efficaces sur les données.

- **Les 3 opérations fondamentales :**
  1. Recherche
  2. Insertion
  3. Suppression

- **Propriété clé :** Chaque nœud sépare les clés plus petites (à gauche) et les plus grandes (à droite).

Recherche d’un Élément dans un ABR

L’opération de recherche exploite la propriété de l’ABR.

- **Algorithme (récursif) :**
  - Si l’arbre est vide, l’élément n’est pas trouvé.
  - Si la clé recherchée est celle du nœud courant, retourner le nœud.
  - Si la clé est < clé du nœud, rechercher dans le sous-arbre gauche.
  - Sinon, rechercher dans le sous-arbre droit.

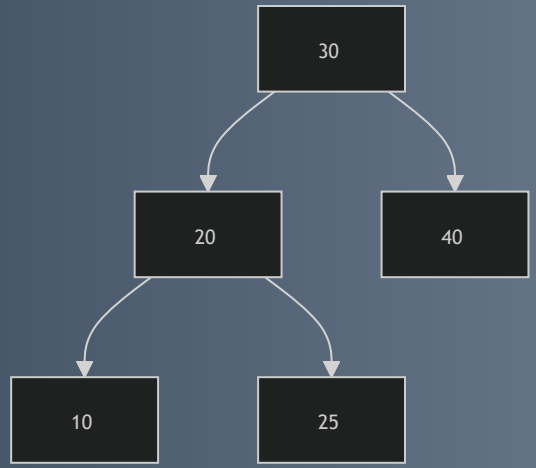
- **Exemple de code © :**

```c
Node* recherche(Node *root, int key) {
    if (root == NULL || root->key == key)
        return root;
    if (key < root->key)
        return recherche(root->left, key);
    else
        return recherche(root->right, key);
}
```

- **Complexité :**
  - Moyenne : (O(\log n)) (si l'arbre est équilibré)
  - Pire cas : (O(n)) (si l'arbre est dégénéré, comme une liste)

L’insertion respecte la structure de l’ABR en plaçant la nouvelle clé à une feuille.

- **Algorithme :**
  - Si l’arbre est vide, créer un nouveau nœud.
  - Comparer la clé avec la racine.
  - Insérer récursivement à gauche si la clé < racine, sinon à droite.
- **Illustration d’une insertion (ex: Insérer `25`) :**

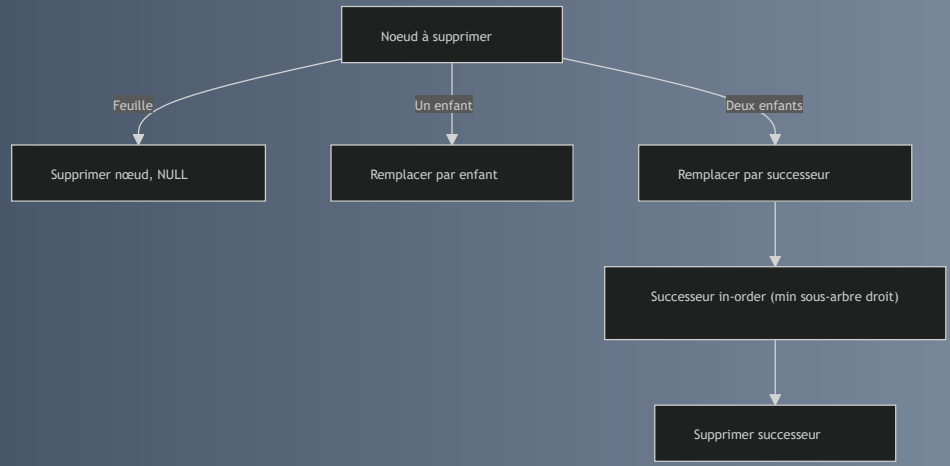


Suppression d’un Élément dans un ABR : Gestion des Cas

La suppression est l’opération la plus délicate, avec trois cas principaux.

- **Cas 1 : Le nœud est une feuille (0 enfant)**
  - Supprimer simplement le nœud ; le parent pointe vers `NULL`.
- **Cas 2 : Le nœud a un seul enfant**
  - Le parent relie directement cet enfant, "sautant" le nœud supprimé.
- **Cas 3 : Le nœud a deux enfants**
  - Trouver le successeur in-order (le plus petit nœud du sous-arbre droit).
  - Remplacer la clé du nœud supprimé par la clé de ce successeur.
  - Supprimer récursivement le successeur.

- **Illustration des cas de suppression :**



Synthèse des Complexités sur un ABR

La complexité des opérations dépend grandement de l'équilibre de l'arbre.

Opération	Temps moyen	Pire cas
Recherche	(O(\log n))	(O(n))
Insertion	(O(\log n))	(O(n))
Suppression	(O(\log n))	(O(n))

- **Remarque :** Les temps moyens sont atteints lorsque l'arbre est relativement équilibré. Dans le pire cas (arbre dégénéré), les performances chutent à celles d'une liste chaînée.

Points Clés & Ressources Utiles

- **Ce qu'il faut retenir :**
  - Les opérations sur un ABR nécessitent une bonne compréhension des différents cas, notamment pour la suppression.
  - La maîtrise de ces algorithmes garantit une manipulation efficace et cohérente des données structurées.
- **Sources consultées :**
  - GeeksforGeeks — BST operations
  - Wikipedia — Binary search tree#Deletion
  - Programiz — BST operations
  - TutorialsPoint — BST Deletion

Arbres Binaires de Recherche (ABR) : Les Parcours

Introduction aux parcours d'arbres Un parcours d'arbre consiste à visiter tous les nœuds selon un ordre donné. Pour les arbres binaires, il existe principalement trois parcours classiques :

- **Infixe (In-order)**
- **Préfixe (Pre-order)**
- **Postfixe (Post-order)**

Chacun a ses usages et propriétés spécifiques.

Parcours Infixe (In-order)

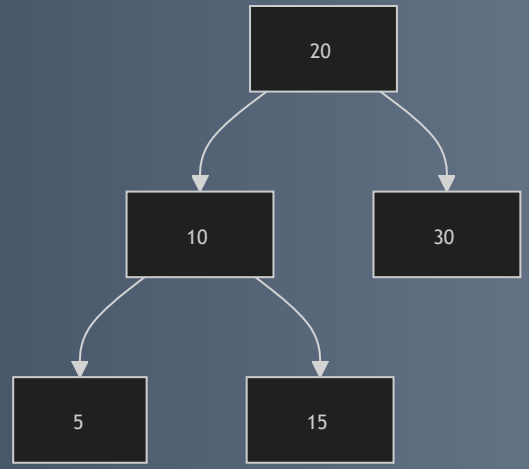
Définition Visite le sous-arbre gauche, puis le nœud courant, puis le sous-arbre droit.

Algorithme récursif

```c
void inOrder(Node *root) {
    if (root != NULL) {
        inOrder(root->left);
        printf("%d ", root->key);
        inOrder(root->right);
    }
}
```

Propriété importante Pour un **ABR**, le parcours infixe restitue les clés dans l'ordre **croissant**.

Exemple



Parcours infixe : `5 10 15 20 30`

Parcours Préfixe (Pre-order)

Définition Visite le nœud courant, puis le sous-arbre gauche, enfin le sous-arbre droit.

Algorithme récursif

```c
void preOrder(Node *root) {
    if (root ≠ NULL) {
        printf("%d ", root→key);
        preOrder(root→left);
        preOrder(root→right);
    }
}
```

Applications

- Sauvegarde ou clonage d'un arbre (recrée l'arbre dans le même ordre).
- Évaluation d'expressions en notation préfixe (notation polonaise).

Parcours Postfixe (Post-order)

Définition Visite le sous-arbre gauche, le sous-arbre droit, puis le nœud courant.

Algorithme récursif

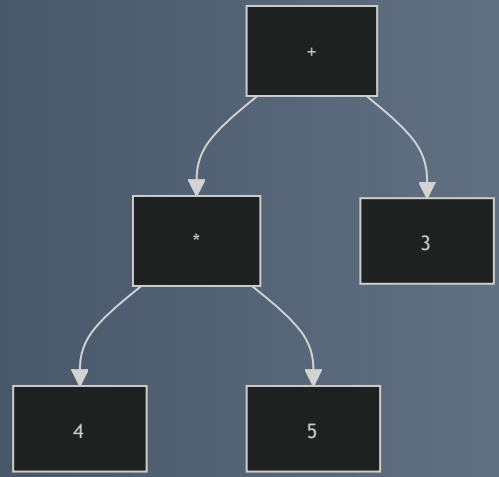
```c
void postOrder(Node *root) {
    if (root ≠ NULL) {
        postOrder(root→left);
        postOrder(root→right);
        printf("%d ", root→key);
    }
}
```

Applications

- Suppression ou libération d'un arbre (libération des fils avant le parent).
- Évaluation d'expressions en notation postfixe (notation polonaise inverse).

Synthèse Visuelle et Exemple Concret

Exemple : Expression arithmétique ((3 + (4 \times 5)))



- Parcours préfixe : `+ 3 * 4 5`
- Parcours infixe : `3 + 4 * 5` (respecte la priorité des opérateurs)
- Parcours postfixe : `3 4 5 * +`

Bilan : Complexité, Usages Clés et Références

Complexité Tous les parcours visitent chaque nœud exactement une fois, d'où une complexité linéaire : [O(n)]

Applications concrètes (Synthèse)

Parcours	Usage principal
Infixe	Récupérer les données triées dans un ABR
Préfixe	Copier un arbre, générer une expression préfixe
Postfixe	Libération mémoire, évaluation expression postfixe

Ce qu'il faut retenir Les parcours d'arbres sont des outils indispensables pour exploiter la structure des ABR, qu'il s'agisse d'accéder aux données triées, d'opérer sur des expressions ou de gérer la mémoire.

Sources consultées

- GeeksforGeeks — Tree traversal
- Wikipedia — Tree traversal
- Programiz — Tree Traversal
- Visualgo — Tree traversals

La Hauteur de l’ABR : Clé de l’Efficacité

- L’efficacité des opérations (recherche, insertion, suppression) sur un ABR dépend directement de sa **hauteur (h)**.
- **Définition :** La hauteur (h) est la longueur maximale du chemin de la racine vers une feuille.
- Les opérations parcourent au maximum un chemin racine-feuille, d’où leur dépendance à (h).
- **Objectif :** Obtenir (h = O(\log n)) pour un arbre équilibré ((n) nœuds).
- **Risque :** Sans contrôle, l’arbre peut dégénérer en liste chaînée, menant à (h = O(n)).

Complexité des Opérations Standard sur ABR

Opération	Complexité moyenne	Pire cas
Recherche	$O(\log n)$	$O(n)$
Insertion	$O(\log n)$	$O(n)$
Suppression	$O(\log n)$	$O(n)$

Explications :

- **Recherche :** Implique une comparaison successive des clés en descendant, sur une hauteur maximale (h).
- **Insertion :** Suit la logique de la recherche pour identifier le point d'ajout.
- **Suppression :** Nécessite une recherche préalable, puis un ajustement des liens, potentiellement complexe si le nœud a deux enfants.

L’Impact de l’Ordre d’Insertion sur la Hauteur

L’ordre dans lequel les éléments sont insérés influence drastiquement la structure de l’ABR et donc sa hauteur :

- **Cas optimal : Arbre équilibré**

  (Éléments insérés dans un ordre qui équilibre l’arbre)

  Séquence : (1, 5, 3, 7, 2, 6, 4)

  Hauteur : (\approx \log_2 n)

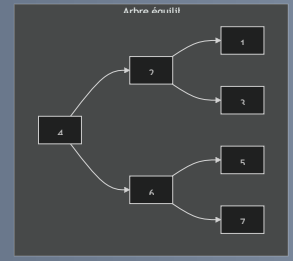
- **Cas pire : Arbre dégénéré**

  (Éléments insérés en ordre croissant ou décroissant)

  Séquence : (1, 2, 3, 4, 5, 6, 7)

  Hauteur : (n) (l’arbre se transforme en une liste chaînée)

Visualisation : ABR Équilibré vs. Dégénéré



Pourquoi l’équilibrage est crucial ?

- Une hauteur en (\Theta(\log n)) garantit une complexité algorithmique optimale.
- Les ABR non équilibrés entraînent des coûts linéaires ((O(n))), dégradant les performances.
- D’où l’importance des arbres auto-équilibrés (ex: AVL, arbres rouges-noirs) pour maintenir une hauteur logarithmique.

Comparaison avec d’autres structures :

Structure	Recherche	Insertion	Suppression
ABR non équilibré	(O(n))	(O(n))	(O(n))
ABR équilibré (AVL)	(O(\log n))	(O(\log n))	(O(\log n))
Table de hachage	(O(1)) (moyenne)	(O(1))	(O(1))

Ce qu’il faut retenir : Pour exploiter pleinement le potentiel d’un ABR, il est indispensable de comprendre que la hauteur conditionne la complexité des opérations. La gestion efficace de l’équilibre est la clé pour maintenir un comportement performant.

Sources consultées :

- GeeksforGeeks — BST Operations Time Complexity
- Wikipedia — Binary Search Tree#Time_complexity

Qu’est-ce qu’un ABR et quel est le problème ?

Un **Arbre Binaire de Recherche (ABR)** organise les données pour des opérations rapides (recherche, insertion, suppression). Sa performance dépend crucialement de sa **hauteur**.

Le problème survient avec les **ABR dégénérés** :

- La structure s’apparente à une **liste chaînée**.
- Chaque nœud n’a qu’**un seul enfant**.
- La hauteur (h) devient linéaire par rapport au nombre de nœuds (n), soit (h=O(n)).

Comprendre la Dégénérescence

Un ABR dégénéré perd ses avantages car ses opérations deviennent aussi coûteuses que dans une liste chaînée.

Caractéristiques clés :

- Chaque nœud ne possède qu'un seul enfant (à gauche ou à droite).
- La hauteur de l'arbre est proportionnelle au nombre de nœuds ((h=O(n))).

Origine fréquente : Ce cas se produit souvent lors de l'insertion de clés dans un ordre **strictement croissant ou décroissant** sans mécanisme de rééquilibrage.

Impact sur les Performances : ABR Dégénérés vs. Équilibrés

La dégénérescence dégrade la complexité des opérations, transformant un arbre performant en une structure inefficace.

Aspect	ABR équilibré	ABR dégénéré
Hauteur (h)	$O(\log n)$	$O(n)$
Complexité recherche	$O(\log n)$	$O(n)$
Complexité insertion	$O(\log n)$	$O(n)$
Complexité suppression	$O(\log n)$	$O(n)$

Visualiser la Dégénérescence

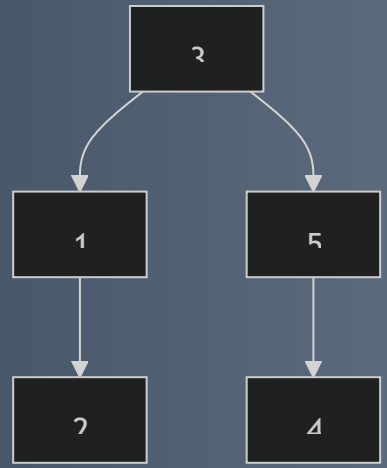
Insertion dans l’ordre croissant (dégénéré)

Insérer 1, 2, 3, 4, 5 :

La hauteur est égale au nombre de nœuds.

Insertion dans un ordre "équilibré"

Insérer 3, 1, 5, 2, 4 :



Structure plus plate, chemins d’accès plus courts.

Origines du Problème et Nécessité d’une Solution

La dégénérescence est le résultat direct de l’absence de stratégies de rééquilibrage.

Causes principales :

- **Insertions triées** (croissantes ou décroissantes).
- **Suppressions non optimisées** sans ajustement de la structure.

Cette faiblesse fondamentale des ABR simples motive la conception de structures plus robustes.

Arbres Équilibrés : La Solution

Pour éviter la dégénérescence, des structures d'arbres **auto-équilibrés** ont été développées. Elles s'adaptent dynamiquement pour maintenir une hauteur ($h = O(\log n)$), garantissant des performances optimales.

Exemples :

- Arbres AVL
- Arbres rouges-noirs
- Arbres B

Ce qu’il faut retenir & Références

En bref : La notion de dégénérescence dans un ABR révèle ses limites sans mécanismes d’équilibrage. Cette contrainte majeure motive le développement des arbres équilibrés, essentiels pour des performances stables et efficaces sur de larges ensembles de données.

Sources consultées :

- GeeksforGeeks — BST degenerated
- Wikipedia — Binary Search Tree#Degenerate tree
- Programiz — BST Problems
- Coursera Algorithms — Tree height and balance

Pourquoi équilibrer un ABR ? Le problème fondamental

- **Le danger de la dégénérescence :** Un Arbre Binaire de Recherche (ABR) peut devenir une simple liste chaînée si les données sont insérées dans un ordre trié.
- **Impact sur la hauteur :** Cette dégénérescence entraîne une hauteur (h = O(n)), où (n) est le nombre de nœuds.
- **Dégradation des performances :** Les opérations (recherche, insertion, suppression) deviennent alors linéaires ((O(n))) au lieu de logarithmiques ((O(\log n))).

Objectif de l'équilibrage : Assurer que la hauteur de l'arbre reste proportionnelle à (\log n), garantissant un temps d'accès efficace quelle que soit l'insertion.

Qu’est-ce que l’équilibrage ? Principes et Mesure

- **Définition :** L’**équilibrage** est un ensemble de techniques qui maintiennent la structure de l’arbre pour éviter qu’elle ne devienne trop déséquilibrée.
- **Mesure de l’équilibre :** Il s’agit de la différence de hauteur entre les sous-arbres gauche et droit d’un nœud.
  - *Exemple (Arbre AVL) :* Cette différence (appelée facteur d’équilibre) doit être strictement comprise entre -1 et +1.
- **Comment l’opérer ?** Principalement par une réorganisation de l’arbre au cours des opérations (insertion ou suppression).

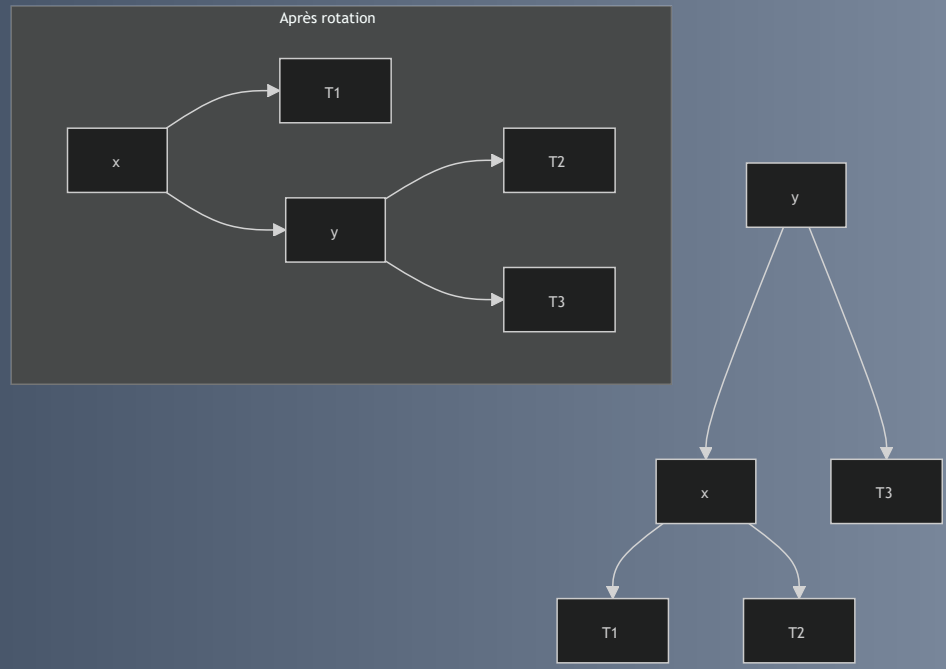
Les Rotations : Le mécanisme clé de l’équilibrage

- **Principe :** Ce sont des opérations locales qui changent la structure de l’arbre sans violer sa propriété ABR (Binary Search Tree).
- **Quand ?** Après une insertion ou une suppression, si l’arbre est déséquilibré, des rotations sont appliquées pour restaurer l’équilibre.

Type de rotation	Description
Rotation simple gauche	Rééquilibre un sous-arbre lourd à droite
Rotation simple droite	Rééquilibre un sous-arbre lourd à gauche
Rotation double gauche-droite	Combinaison de rotations
Rotation double droite-gauche	Combinaison inverse

Illustration : Exemple de Rotation Droite

Une rotation droite sur un nœud (y) intervient quand son sous-arbre gauche (x) est plus lourd.



Cette rotation diminue la hauteur du côté gauche, rééquilibrant ainsi l’arbre en remontant (x).

Types d’Arbres Équilibrés & L’Impact sur la Complexité

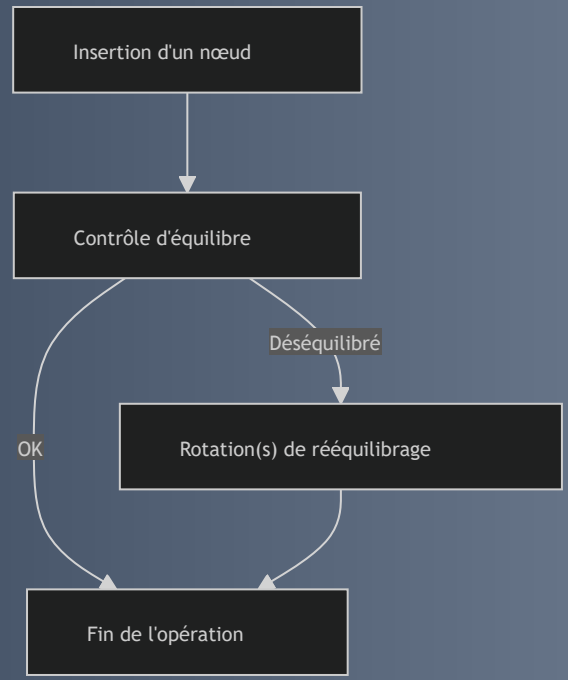
Type	Méthode d’équilibrage	Caractéristiques
Arbre AVL	Maintien strict du facteur d’équilibre (-1, 0, 1)	Rotations fréquentes, très strict
Arbre Rouge-Noir	Utilise la couleur des nœuds et propriétés spécifiques	Plus souple, opérations plus amorties

Impact sur la complexité après équilibrage :

- La hauteur (h) est garantie en (O(\log n)).
- Les opérations de recherche, insertion et suppression ont systématiquement une complexité (O(\log n)).

Synthèse du Processus et Ressources

Opération d'équilibrage après insertion :



Ce qu'il faut retenir : Maintenir un arbre équilibré est la clé pour préserver l'efficacité des opérations sur un ABR. Les rotations et autres mécanismes automatiques permettent à la structure de s'adapter dynamiquement, garantissant des performances optimales.

Sources consultées :

- GeeksforGeeks — AVL Tree
- Wikipedia — Self-balancing binary search tree
- Programiz — AVL Tree Rotations
- TutorialsPoint — Red-Black Tree

Arbres AVL : L’Équilibrage Automatique

Introduction aux Arbres Équilibrés

Les arbres **AVL** (Adelson-Velskiĭ et Landis, 1962) sont des arbres binaires de recherche auto-équilibrés.

Ils garantissent que pour chaque nœud :

- La hauteur de son sous-arbre gauche
- Et la hauteur de son sous-arbre droit

…diffèrent d’au plus **1**.

Ce mécanisme assure que l’arbre reste compact et efficient pour toutes les opérations.

Définition du Facteur d’Équilibre (FE)

Pour chaque nœud (n), son **facteur d’équilibre** est :

[FE(n) = \text{hauteur}(sous_arbre_gauche) - \text{hauteur}(sous_arbre_droit)]

Condition d’équilibre AVL

Un arbre est AVL si, et seulement si, pour tout nœud (n) : [FE(n) \in {-1,,0,,1}]

Si (FE(n)) est en dehors de cet intervalle, l’arbre est déséquilibré à ce nœud.

Pourquoi ce facteur ?

- Mesure localement l’équilibre de l’arbre.
- Permet une détection rapide des déséquilibres après des insertions ou suppressions.
- Assure une hauteur logarithmique (\approx O(\log n)), garantissant des performances optimales.

Rotations AVL : Mécanismes de Rééquilibrage

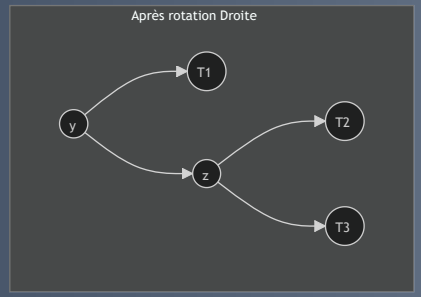
Les rotations sont les opérations fondamentales pour corriger les déséquilibres dans un arbre AVL. Elles s'effectuent localement autour du nœud déséquilibré.

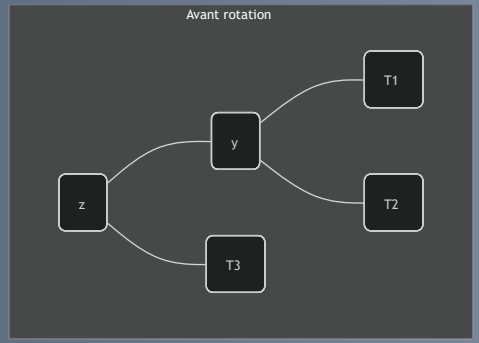
Types de Rotations

Rotation	Description	Quand l'utiliser ?
Simple Droite (RR)	Corrige un déséquilibre causé par insertion à gauche du sous-arbre gauche	`FE(z) = +2` et `FE(y) = +1`
Simple Gauche (LL)	Corrige un déséquilibre causé par insertion à droite du sous-arbre droit	`FE(z) = -2` et `FE(y) = -1`
Double Gauche-Droite (LR)	Déséquilibre causé par insertion à droite du sous-arbre gauche	`FE(z) = +2` et `FE(y) = -1`
Double Droite-Gauche (RL)	Déséquilibre causé par insertion à gauche du sous-arbre droit	`FE(z) = -2` et `FE(y) = +1`

Rotation Simple Droite (RR)

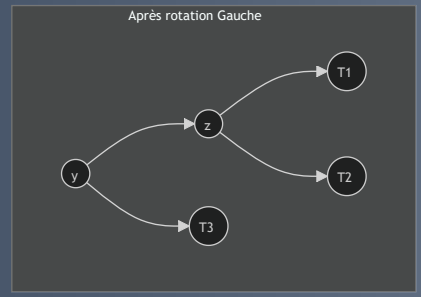
Utilisée si `FE(z) = +2` et `FE(y) = +1`. On effectue une rotation droite autour de (z).

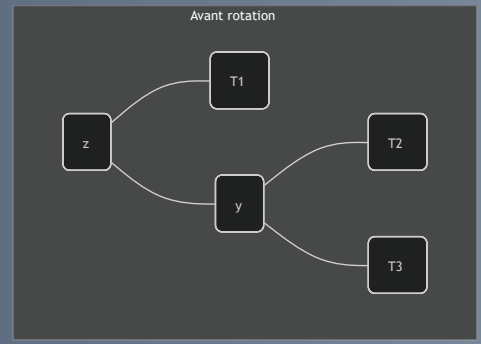




Rotation Simple Gauche (LL)

Utilisée si `FE(z) = -2` et `FE(y) = -1`. On effectue une rotation gauche autour de (z).





Rotations Doubles

Utilisées lorsque le déséquilibre est causé par une insertion sur le sous-arbre opposé du fils du nœud déséquilibré.

Rotation Double Gauche-Droite (LR)

- Déséquilibre: `FE(z) = +2` et `FE(y) = -1`
- Procédure:
  1. Rotation gauche sur `y`
  2. Rotation droite sur `z`

Rotation Double Droite-Gauche (RL)

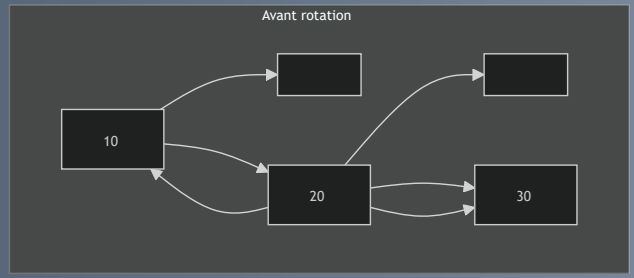
- Déséquilibre: `FE(z) = -2` et `FE(y) = +1`
- Procédure:
  1. Rotation droite sur `y`
  2. Rotation gauche sur `z`

Exemple d’insertion avec rotation (LL)

Insertion successive des clés : 10, 20, 30 dans un ABR vide.

- Après 10 et 20, l’arbre est équilibré.
- L’insertion de 30 crée un déséquilibre en 10 (`FE(10) = -2`).
- Une rotation simple gauche (LL) est effectuée au nœud 10.

Aprés rotation gauche LL



Complexité des Opérations

Grâce à la garantie d'une hauteur logarithmique, toutes les opérations fondamentales (recherche, insertion, suppression) s'exécutent en temps :

$$ O(\log n) $$

Ce qu'il faut retenir

La structure des arbres AVL, basée sur le facteur d'équilibre et ses rotations, garantit une auto-réorganisation dynamique. Cette propriété préserve les performances optimales sur toutes les opérations de l'arbre binaire de recherche.

Sources consultées

- GeeksforGeeks — AVL Tree
- Wikipedia — AVL Tree
- Programiz — AVL Tree
- TutorialsPoint — AVL Tree Insertions and Rotations

Arbres Équilibrés : Les Arbres Rouge-Noir

Un mécanisme auto-équilibré pour des opérations efficaces

Un **arbre Rouge-Noir (RN)** est un arbre binaire de recherche auto-équilibré. Il utilise des **contraintes de propriétés colorées** qui garantissent :

- Une hauteur logarithmique ((O(\log n))).
- Des opérations (recherche, insertion, suppression) efficaces.

Les 5 Règles d’Or des Arbres Rouge-Noir

Des propriétés pour un équilibre garanti

1. **Chaque nœud est soit rouge, soit noir.**
2. **La racine est noire.**
3. **Toutes les feuilles (NIL ou NULL) sont noires.**
4. **Un nœud rouge ne peut pas avoir de nœud rouge comme enfant** (pas de deux rouges consécutifs).
5. **Pour chaque nœud, tous les chemins simples de ce nœud aux feuilles descendants contiennent le même nombre de nœuds noirs** (black-height).

Conséquences :

- Ces propriétés imposent un équilibrage global.
- La hauteur (h) d’un arbre Rouge-Noir est (O(\log n)), où (n) est le nombre de nœuds.
- La complexité des recherches, insertions et suppressions est en (O(\log n)).

Rouge-Noir ou AVL ? Deux approches de l’équilibrage

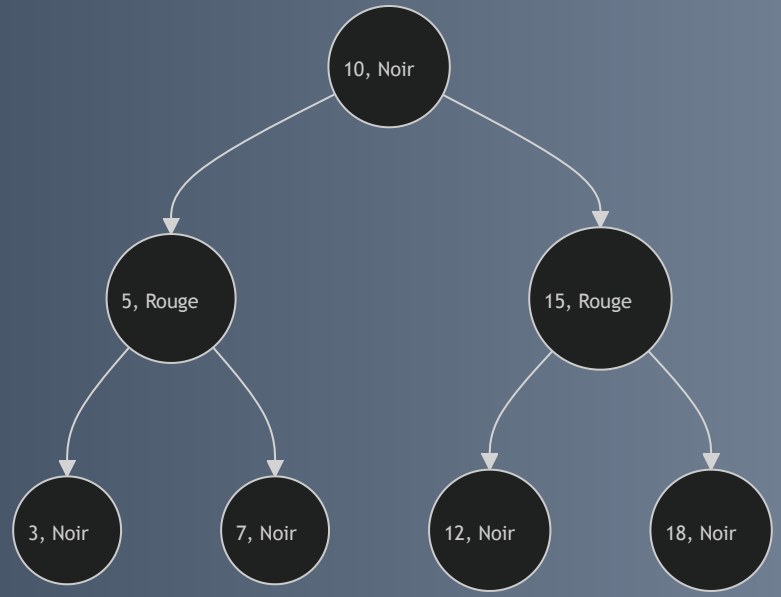
Des choix adaptés à des besoins différents

Critère	Arbre AVL	Arbre Rouge-Noir
Rigidité de l’équilibre	Très stricte (facteur -1,0,1)	Plus souple (propriétés colorées)
Nombre de rotations	Plus élevé	Moins de rotations, amorti
Cas d’usage typique	Lecture intensive	Insertion/suppression fréquente

Un Arbre Rouge-Noir en Action et le Défi de l'Équilibre

Comment les propriétés se manifestent et se maintiennent

Illustration simple :

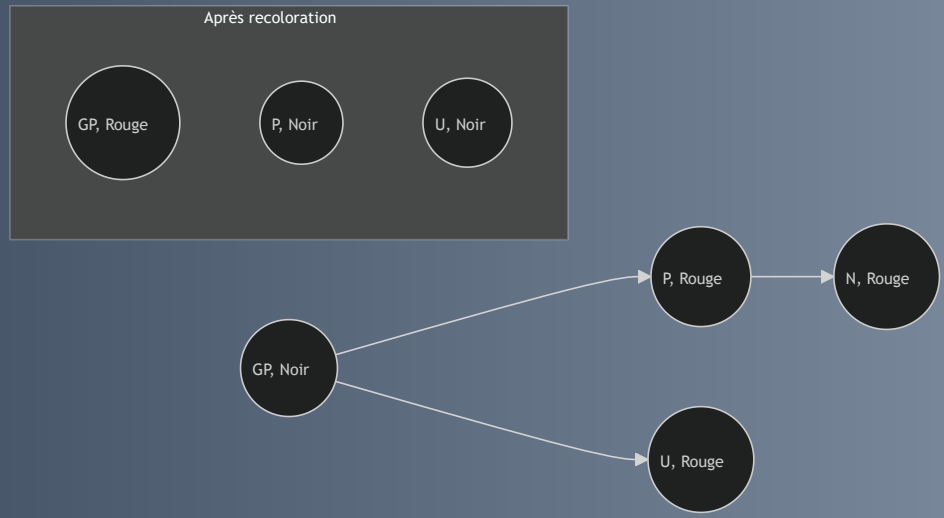


L'arbre respecte les propriétés : racine noire, nœuds rouges (5,15) n'ont pas d'enfant rouge, chaque chemin vers feuille a même nombre de noirs.

Rééquilibrage à l'insertion :

- Après insertion d'un nœud (toujours rouge initialement), une propriété peut être violée (notamment 2 rouges consécutifs).
- On rétablit l'équilibre en utilisant :
  - **Recoloration** des nœuds.
  - **Rotations** (simples ou doubles).

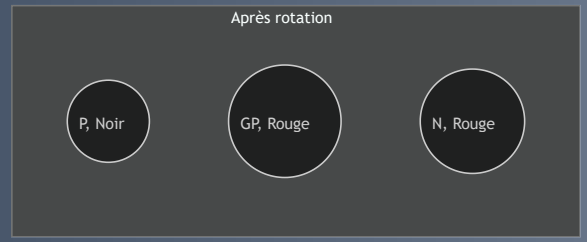
Cas 1 : Oncle du nœud inséré est rouge (recoloration)



- Parents et oncles rouges passent au noir.
- Grand-parent devient rouge.
- Le processus remonte en récursion si besoin.

Cas 2 : Oncle du nœud inséré est noir (rotation nécessaire)

- **Sous-cas 2a :** Nœud inséré à l'intérieur (rotation double).
- **Sous-cas 2b :** Nœud inséré à l'extérieur (rotation simple).



Graphes : Concepts Fondamentaux et Terminologie

- Un **graphe** est une structure composée de deux ensembles :
  - **Sommets (nœuds)** : éléments fondamentaux représentant des objets ou entités.
  - **Arêtes (liens)** : relations ou connexions entre les sommets.

- **Notation :** (G = (V, E))
  - (V): ensemble des sommets.
  - (E): ensemble des arêtes.

- **Ordre du graphe :** (|V| = n) (nombre de sommets).
- **Taille du graphe :** (|E| = m) (nombre d'arêtes).

Les Composants : Sommets et Arêtes en Détail

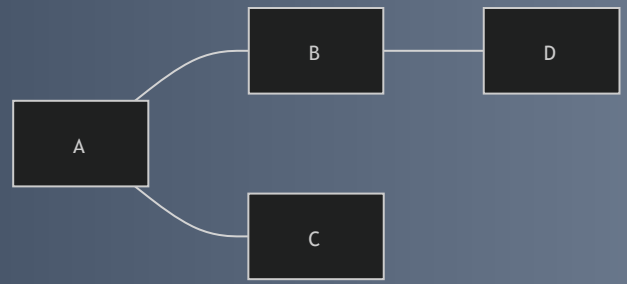
- **Sommets :**
  - Une entité identifiable dans le graphe.
  - *Exemples :* villes dans un réseau routier, pages web sur Internet.

- **Arêtes :**
  - Relient deux sommets.
  - Peuvent être caractérisées par :
    - **Entrée et sortie** (avec des sommets d'arrivée/départ).
    - **Poids** (valeur numérique associée).

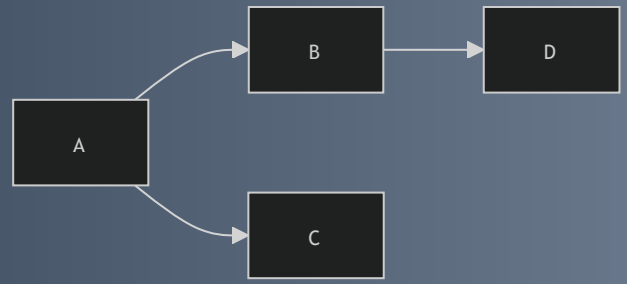
Graphes Orientés et Non-Orientés : Distinctions Cruciales

Type de graphe	Description	Implication
Graphe non orienté	Arêtes sans direction, bidirectionnelles	L'arête ({u,v}) représente une connexion entre (u) et (v) dans les deux sens.
Graphe orienté	Arêtes avec direction (flèches)	L'arête ((u,v)) va de (u) à (v), sens unique.

Exemple - Graphe non orienté :



Exemple - Graphe orienté :

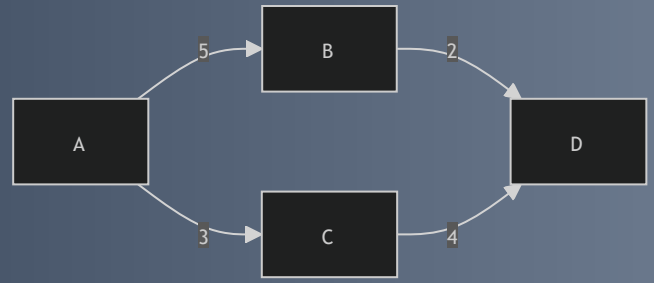


Graphes Pondérés : Ajouter de l’Information aux Arêtes

- Un **graphe pondéré** attribue à chaque arête un **poids** (valeur numérique) représentant par exemple :
  - Distance
  - Coût
  - Temps
  - Capacité

- Un graphe pondéré peut être orienté ou non-orienté.

Illustration d’un graphe pondéré orienté :



Terminologie Essentielle pour l’Analyse des Graphes

Terme	Description
Degré d’un sommet	Nombre d'arêtes incidentes (non orienté) ou sortantes/entrantes (orienté).
Chemin	Suite de sommets reliés par des arêtes.
Cycle	Chemin fermé, commençant et finissant au même sommet.
Composante connexe	Sous-graphe où tout sommet est accessible depuis tout autre (graphe non orienté).

Synthèse et Ressources

Ce qu'il faut retenir :

- Un graphe est un modèle puissant pour représenter des relations complexes entre objets variés.
- La compréhension claire des sommets, arêtes, orientations et poids est fondamentale pour exploiter ces structures dans les algorithmes avancés.

Sources consultées :

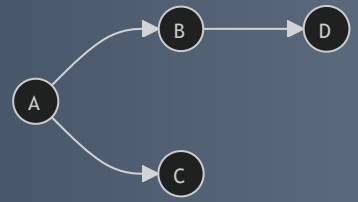
- Wikipedia — Graph (graph theory)
- GeeksforGeeks — Introduction to Graph and Representation
- Programiz — Graph Data Structure

Qu’est-ce qu’une Matrice d’Adjacence ?

- **Définition :** Représentation d’un graphe (G=(V,E)) par une matrice carrée (n \times n), où (n = |V|) est le nombre de sommets.
  - Chaque ligne et colonne correspond à un sommet.
  - (M[i][j]) indique l’existence (et parfois le poids) d’une arête entre (i) et (j).
- **Implémentation Basique (Non Pondérée) :** [M[i][j] = \begin{cases} 1 & \text{si une arête existe de } i \text{ vers } j \ 0 & \text{sinon} \end{cases}]
- **Matrice Pondérée :**
  - Stocke le poids de l’arête. [M[i][j] = \begin{cases} w_{ij} & \text{si arête existe} \ 0 \text{ ou } \infty & \text{sinon} \end{cases}]

Exemple Pratique : Graphe Orienté

Considérons un graphe avec sommets (A, B, C, D) et arêtes (A \to B, A \to C, B \to D).



Matrice d’Adjacence Correspondante :

	A	B	C	D
A	0	1	1	0
B	0	0	0	1
C	0	0	0	0
D	0	0	0	0

Avantages de la Matrice d’Adjacence

- **Accès rapide :** Test instantané en (O(1)) de l’existence d’une arête entre deux sommets.
- **Simplicité d’implémentation :** Utilise des tableaux 2D standards.
- **Efficace pour les graphes denses :** Idéale lorsque le nombre d’arêtes est proche de (n^2).
- **Facilite certains algorithmes :** Pratique pour des algorithmes comme Floyd-Warshall (plus courts chemins).

Inconvénients et Comparaison

- **Inconvénients :**
  - **Espace mémoire important :** Nécessite (O(n^2)) mémoire, inefficace pour les graphes peu denses (clairsemés).
  - **Parcours des voisins lent :** Trouver tous les voisins d'un sommet prend (O(n)) (parcourir une ligne complète).
  - Peu adaptée aux graphes très grands et clairsemés.
- **Comparaison rapide avec la Liste d'Adjacence :**

Critère	Matrice d'adjacence	Liste d'adjacence
Complexité mémoire	(O(n^2))	(O(n + m))
Recherche d'arête	(O(1))	(O(\delta))
Parcours des voisins	(O(n))	(O(\delta))

Synthèse et Sources

- **Ce qu’il faut retenir :**
  - La matrice d’adjacence est une structure simple et puissante.
  - Elle est particulièrement efficace pour les graphes petits à moyennement denses.
  - Sa compréhension est fondamentale pour l’étude et la mise en œuvre d’algorithmes sur graphes.
- **Sources consultées :**
  - GeeksforGeeks — Adjacency Matrix
  - Wikipedia — Adjacency matrix
  - Programiz — Graph representation
  - TutorialsPoint — Graphs Representation

Représentation de Graphes : Les Listes d’Adjacence

Qu’est-ce qu’une Liste d’Adjacence ?

La **liste d’adjacence** est une structure de données qui représente un graphe. Elle associe à chaque sommet une liste des sommets adjacents (voisins) auxquels il est connecté par une arête.

Formellement, pour un graphe (G=(V,E)) : C’est un tableau (ou dictionnaire) où chaque indice ou clé correspond à un sommet, et contient une liste des sommets reliés par une arête.

Implémentation et Exemple Concret

Comment la représenter ?

Pour chaque sommet $v \in V$, on stocke une liste de tous les sommets u tels que $(v, u) \in E$.

Exemple de structure (en C) :

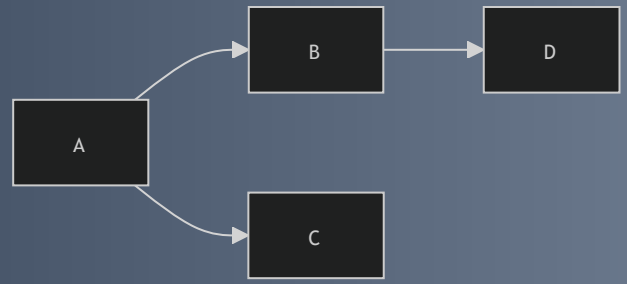
```c
typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

Node* adjacencyList[MAX_VERTICES]; // tableau de pointeurs
```

Chaque `adjacencyList[v]` pointe vers une liste chaînée de nœuds représentant les sommets voisins.

Un cas pratique : Graphe orienté

Arêtes: (A \to B), (A \to C), (B \to D).

Sommet	Voisins
A	B, C
B	D
C	(aucun)
D	(aucun)



Atouts des Listes d’Adjacence

Les listes d’adjacence offrent plusieurs avantages clés :

- **Économie de mémoire** :
  - Nécessite (O(n + m)) où (n) est le nombre de sommets et (m) le nombre d’arêtes.
  - Particulièrement adaptée aux **graphes clairsemés** (peu d’arêtes par rapport au nombre maximal possible).
- **Parcours rapide des voisins** :
  - Accès direct aux voisins d’un sommet sans parcourir une structure entière.
  - Coût de (O(\delta)) où (\delta) est le degré du sommet.
- **Adaptabilité** :
  - Facile à étendre pour stocker des poids d’arêtes, des propriétés de sommets, etc.

Limites des Listes d’Adjacence

Malgré leurs atouts, les listes d’adjacence présentent quelques inconvénients :

- **Recherche d’une arête particulière** :
  - Vérifier l’existence d’une arête entre deux sommets donnés peut être plus coûteux.
  - Nécessite de parcourir la liste des voisins d’un sommet, soit (O(\delta)) avec (\delta) le degré du sommet.
- **Accès aléatoire** :
  - Moins immédiat qu’avec une matrice d’adjacence pour l’accès direct aux informations entre deux sommets.
- **Complexité d’implémentation** :
  - Un peu plus complexe à implémenter, notamment pour la gestion dynamique de la mémoire (listes chaînées, redimensionnement).

Choisir la Bonne Représentation

Comparaison Matrice vs Liste d’adjacence

Critère	Matrice d’adjacence	Liste d’adjacence
Utilisation mémoire	(O(n^2))	(O(n + m))
Test présence arête	(O(1))	(O(\delta))
Parcours voisins	(O(n))	(O(\delta))
Convient pour	Graphes denses	Graphes clairsemés

Cas d’utilisation privilégiés

- Les listes d’adjacence sont la base de la plupart des algorithmes sur graphes :
  - **DFS** (Parcours en profondeur)
  - **BFS** (Parcours en largeur)
  - **Dijkstra** (Plus court chemin)

Ce qu’il faut retenir et pour aller plus loin

En bref

La liste d’adjacence est une représentation **flexible en mémoire** et **efficace pour parcourir les voisins**. Elle est incontournable pour manipuler des graphes de taille variable, surtout s’ils sont **clairsemés**.

Ressources consultées

- GeeksforGeeks — Graph Representation
- Wikipedia — Adjacency List
- Programiz — Graph Data Structure
- TutorialsPoint — Graphs Representation

Représentations de Graphes

Le choix de la représentation d'un graphe est **essentiel** pour l'efficacité des algorithmes qui lui seront appliqués.

Ce choix dépend principalement de la **densité du graphe**.

Qu’est-ce que la Densité d’un Graphe ?

La **densité (D)** est le ratio du nombre d’arêtes (m) par rapport au nombre maximal d’arêtes possibles.

Pour un graphe non orienté sans boucle, avec (n) sommets : [D = \frac{2m}{n(n-1)}]

- **Graphe dense** : (D) proche de 1 (beaucoup d’arêtes).
- **Graphe creux (ou sparse)** : (D) faible (nombre d’arêtes bien inférieur au maximum).

Les Représentations Courantes

Représentation	Complexité mémoire	Accès vérification arête	Parcours voisins	Adapté pour
Matrice d’adjacence	(O(n^2))	(O(1))	(O(n))	Graphes denses
Listes d’adjacence	(O(n + m))	(O(\delta))	(O(\delta))	Graphes creux (sparse)

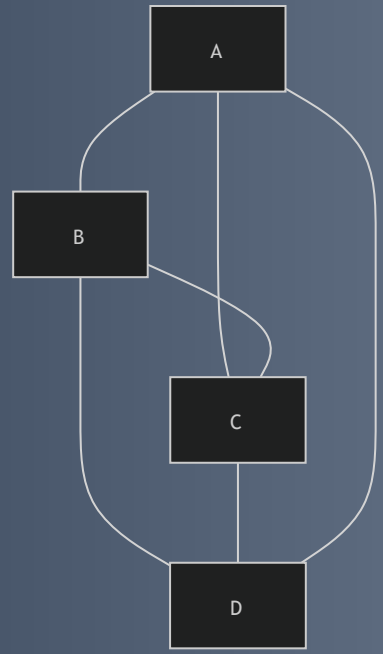
(\delta) = degré du sommet concerné.

Un compromis crucial entre mémoire et rapidité d’accès aux informations.

Choix pour les Graphes Denses

- **Critères** : Nombre d’arêtes proche de (n^2). Presque tous les sommets sont reliés.
- **Représentation privilégiée** : **Matrice d’adjacence**
  - Accès direct (O(1)) pour vérifier une arête.
  - Justifiée malgré son coût mémoire (O(n^2)).

Exemple : Matrice d’adjacence d’un graphe dense

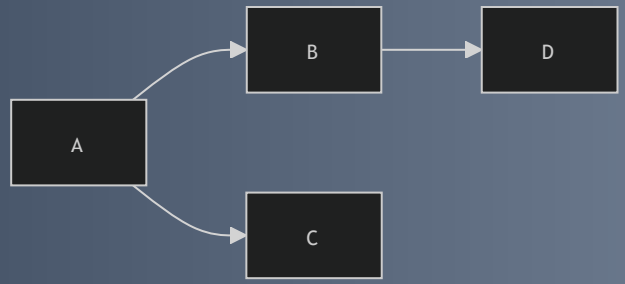


	A	B	C	D
A	0	1	1	1
B	1	0	1	1
C	1	1	0	1
D	1	1	1	0

Choix pour les Graphes Creux (Sparse)

- **Critères** : Nombre d'arêtes proche de (O(n)).
- **Représentation privilégiée** : **Listes d'adjacence**
  - Économie de mémoire : (O(n + m)).
  - Parcours direct des voisins en (O(\delta)) (degré du sommet).

Exemple : Listes d’adjacence d’un graphe creux



Listes d’adjacence :

- A : B, C
- B : D
- C : -
- D : -

Impact sur les Algorithmes, Synthèse & Conclusion

Impact sur les Algorithmes

- **Accès rapide aux arêtes** (matrice d'adjacence) favorise certains algorithmes (ex: Floyd-Warshall).
- **Parcours efficace des voisins** (listes) optimise DFS, BFS, Dijkstra sur graphes creux.

Synthèse des Représentations

Caractéristique	Matrice d'adjacence	Liste d'adjacence
Utilisation mémoire	(O(n^2))	(O(n + m))
Recherche d'arête	(O(1))	(O(\delta))
Parcours voisins	(O(n))	(O(\delta))
Efficace pour	Graphes petits et denses	Grands graphes clairsemés

Ce qu’il faut retenir

Le choix judicieux entre matrice d’adjacence et liste d’adjacence dépend avant tout de la nature du graphe à traiter. Une bonne adéquation entre structure et problème garantit simplicité, efficacité et évolutivité des algorithmes.

Sources consultées

- GeeksforGeeks — Graph representations
- Wikipedia — Graph density
- Programiz — Graph Data Structure
- TutorialsPoint — Graphs Representation

Le Breadth-First Search (BFS)

Contenu de cette séance :

- Comprendre le principe du parcours en largeur (BFS).
- Découvrir son implémentation clé via une file.
- Explorer ses diverses applications pratiques.

Le BFS est un algorithme fondamental pour l'exploration des graphes, particulièrement utile pour déterminer les plus courts chemins dans les graphes non pondérés.

Le Principe du Breadth-First Search (BFS) et le Rôle de la File

1. Exploration en Couches Successives :

- Le BFS explore les sommets en "vagues", comme l'eau se propage.
- Il visite d'abord tous les voisins immédiats d'un sommet de départ.
- Puis, il explore les voisins de ces voisins, et ainsi de suite.
- Cette méthode garantit la découverte de chaque sommet dans l'ordre croissant de sa distance minimale (en nombre d'arêtes) au sommet source.

2. Utilisation d'une File (Queue) :

- L'algorithme utilise une **file (queue)** pour conserver l'ordre des sommets à visiter.
- Le sommet source est inséré en premier dans la file.
- Quand un sommet est traité (dépilé de la file), tous ses voisins non encore visités sont marqués et ajoutés à la fin de la file.
- Ce mécanisme assure l'exploration "couche par couche".

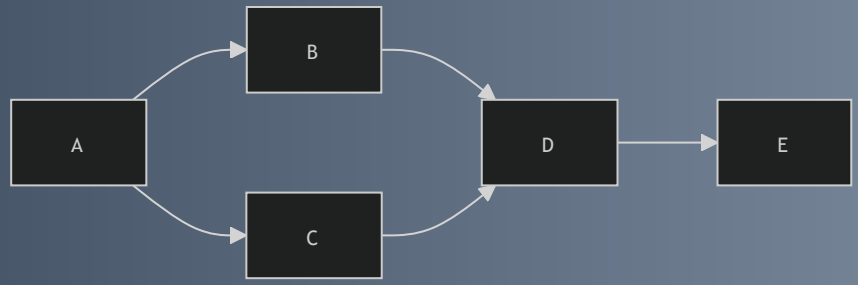
Implémentation du BFS : Pseudo-code

L'algorithme s'appuie sur une file pour gérer l'ordre de visite et un mécanisme de marquage pour éviter les boucles et les visites répétées.

```
BFS(G, s)
  // G : le graphe, s : le sommet de départ
  create empty queue Q
  mark s as visited // Marque le sommet de départ comme visité
  enqueue s into Q  // Ajoute le sommet de départ à la file

  while Q is not empty:
    u = dequeue Q // Retire le premier sommet de la file
    // Traiter ou afficher u
    for each neighbor v of u: // Pour chaque voisin de u
      if v not visited:
        mark v as visited // Marque v comme visité
        enqueue v into Q  // Ajoute v à la file pour une exploration future
```

Exemple de Parcours BFS

Visualisons le parcours BFS sur un graphe orienté à partir du sommet A.



Parcours BFS à partir de A :

1. **File initiale :** `[A]`
2. **Visite A :** enfile B, C → **File :** `[B, C]`
3. **Visite B :** enfile D → **File :** `[C, D]`
4. **Visite C :** D est déjà dans la file (déjà marqué visité), pas d'ajout → **File :** `[D]`
5. **Visite D :** enfile E → **File :** `[E]`
6. **Visite E :** aucun voisin non visité → **File :** `[]` (vide)

Ordre de visite final : A → B → C → D → E

Complexité et Performance du BFS

Le BFS est un algorithme très efficace pour l'exploration de graphes.

- **Complexité Temporelle :** (O(n + m))
  - Où (n) est le nombre de sommets et (m) le nombre d'arêtes du graphe.
  - Chaque sommet est enfilé et dépilé au plus une fois.
  - Chaque arête est examinée au plus deux fois (une fois pour chaque direction dans un graphe non orienté, ou une fois dans un graphe orienté).
- **Complexité Spatiale :** (O(n))
  - Nécessite de la mémoire pour stocker :
    - Les marquages des sommets visités.
    - La file, qui peut contenir au maximum tous les sommets du graphe.

Applications Clés & Ce qu’il faut retenir

Applications de BFS :

- **Calcul des plus courts chemins** en graphes non pondérés (nombre minimal d’arêtes).
- **Détection de composantes connexes** dans un graphe non orienté.
- **Test de bipartité** d’un graphe (peut-on colorier le graphe avec deux couleurs sans que deux sommets adjacents aient la même couleur).
- **Parcours couche par couche** utile dans divers algorithmes de graphes (ex : réseau, intelligence artificielle).
- **Propagation dans les réseaux** (ex : diffusion d’une information ou d’un virus).

Ce qu’il faut retenir : Le BFS révèle l’organisation en couches d’un graphe à partir d’une source et est la base de nombreux algorithmes fondamentaux liés aux graphes. Son implémentation via une file garantit un ordre d’exploration contrôlé, adapté à la découverte des plus courts chemins simples dans les graphes non pondérés.

Sources consultées :

- GeeksforGeeks — BFS
- Wikipedia — Breadth-first search
- Programiz — BFS

Graphes : Le Parcours en Profondeur (DFS)

Le Parcours en Profondeur (DFS) : Principe Fondamental

Le **Depth-First Search (DFS)** explore un graphe en suivant un chemin jusqu'à son terme avant de revenir en arrière (backtracking) pour en explorer d'autres.

- À chaque sommet, on choisit un voisin non visité.
- On poursuit récursivement cette exploration en profondeur.
- Jusqu'à ce qu'on atteigne un sommet sans voisin non visité.
- Puis on revient sur les sommets précédents pour continuer (backtracking).

Ce principe permet d'explorer pleinement toutes les branches depuis un sommet initial.

Implémentation du DFS : Récursivité ou Pile

Le DFS peut être implémenté de deux manières principales :

1. Version récursive (naturelle)

```
DFS(G, u)
  mark u as visited
  for each neighbor v of u:
    if v not visited:
      DFS(G, v)
```

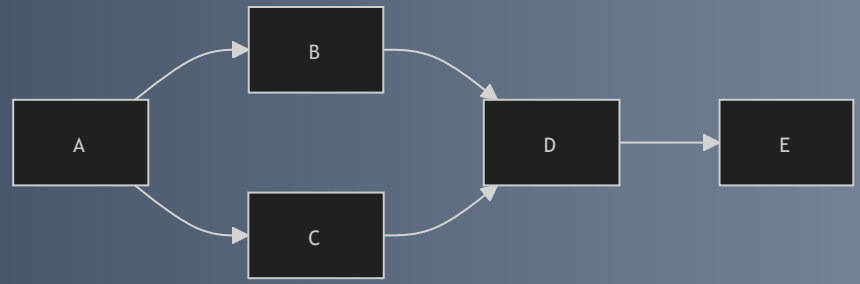
2. Version itérative avec pile (stack)

```
DFS(G, s)
  create empty stack S
  push s onto S

  while S not empty:
    u = pop S
    if u not visited:
      mark u as visited
      for each neighbor v of u:
        push v onto S
```

DFS en Action & Efficacité

Exemple Illustré :

Considérons ce graphe orienté. Un DFS initié en A peut visiter dans l'ordre (selon choix) : **A, B, D, E, C**.



- Part du sommet A
- Explore B (voisin de A)
- Puis D (voisin de B)
- Puis E (voisin de D)
- Enfin C (voisin non visité de A, après retour sur A)

Caractéristiques et Complexité :

- **Temps** : (O(n + m)), où (n) est le nombre de sommets et (m) le nombre d'arêtes. Chaque sommet et arête est exploré une fois.
- **Espace** :
  - Récursion : proportionnel à la profondeur maximale du parcours.
  - Itérative : la pile peut contenir autant d'éléments que la profondeur maximale.

Applications Multiples du DFS

Le parcours en profondeur est un algorithme fondamental avec de nombreuses applications pratiques :

- **Détection de cycles** dans les graphes orientés et non orientés.
- Identification des **composantes fortement connexes** (ex: algorithme de Tarjan).
- **Tri topologique** des graphes acycliques orientés (DAGs).
- Résolution de **labyrinthes et puzzles**.
- **Analyse de connectivité** et extraction de sous-graphes.

Mise en Pratique, Bilan & Ressources

Exemple de code C (version récursive) :

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

// Représentation simplifiée de la liste d'adjacence
typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

Node* adjacencyList[MAX];
int visited[MAX]; // Tableau pour marquer les sommets visités

void DFS(int u) {
    visited[u] = 1;
    printf("%d ", u); // Traitement du sommet (ici, affichage)

    Node* temp = adjacencyList[u];
    while (temp != NULL) {
        if (!visited[temp->vertex]) {
            DFS(temp->vertex); // Appel récursif pour les voisins non visités
        }
        temp = temp->next;
```

Ce qu’il faut retenir :

Le DFS est un parcours fondamental qui explore exhaustivement tous les chemins possibles d’un graphe en suivant la profondeur d’une branche. Sa mise en œuvre simple, récursive ou avec pile, permet d’aborder des problèmes complexes de structure et de connectivité dans les graphes.

Sources consultées :

- GeeksforGeeks — DFS
- Wikipedia — Depth-first search
- Programiz — DFS

Théorie — Algorithmes de Parcours

Analyse de complexité des algorithmes de parcours (BFS, DFS)

Les parcours de graphes (BFS et DFS) sont des outils fondamentaux pour explorer sommets et arêtes.

Leur efficacité, mesurée par la complexité, dépend directement :

- De la **représentation du graphe** (matrice ou liste d'adjacence).
- De ses **caractéristiques** (nombre de sommets (n), nombre d'arêtes (m)).

Comprendre cette dépendance est essentiel pour optimiser vos algorithmes.

Complexité avec la Matrice d’Adjacence

La matrice d’adjacence, de taille (n \times n), représente directement toutes les connexions possibles.

- **Mécanisme** : Pour trouver les voisins d’un sommet, il faut parcourir sa ligne complète de la matrice. Ce coût est de (O(n)) par sommet.
- **Complexité totale** : Étant donné que nous inspectons (n) sommets, le coût global est : [O(n \times n) = O(n^2)]
- **Indépendance** : Cette complexité est indépendante du nombre réel d’arêtes (m), car chaque emplacement de la matrice est vérifié.

Complexité avec la Liste d’Adjacence

La liste d’adjacence stocke uniquement les voisins existants pour chaque sommet.

- **Mécanisme** : Parcourir les voisins d’un sommet coûte (O(\delta)), où (\delta) est son degré (nombre de voisins).
- **Somme des degrés** : La somme des degrés de tous les sommets est égale à (2m) pour un graphe non orienté, ou (m) pour un graphe orienté.
- **Complexité totale** : Puisque chaque sommet est visité une fois et chaque arête est parcourue au plus une fois, la complexité est : [O(n + m)]
- Cette méthode est intrinsèquement plus efficace pour les graphes "clairsemés" (peu d’arêtes par rapport au nombre de sommets).

Synthèse des Complexités pour BFS et DFS

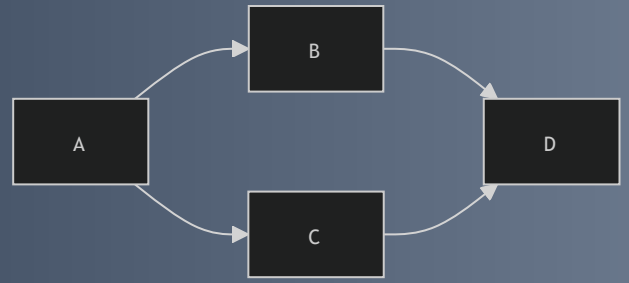
Le choix de la représentation impacte directement la performance de BFS et DFS.

Algorithme	Complexité avec Matrice d’adjacence	Complexité avec Liste d’adjacence
BFS	(O(n^2))	(O(n + m))
DFS	(O(n^2))	(O(n + m))

- Ces complexités optimales sont atteintes en tirant parti d’une représentation adaptée.
- Pour les **graphes clairsemés** (où (m \ll n^2)), la liste d’adjacence est significativement plus efficace.

Illustration Pratique et Impact Chiffré

Graphe orienté simple : (n=4), (m=4)



- **Liste d'adjacence** : Parcourt 4 arêtes.
- **Matrice d'adjacence** : Parcourt (n^2 = 16) éléments.

Exemple chiffré : Graphe avec 1000 sommets Prenons un graphe de 1000 sommets et seulement 5000 arêtes (densité faible).

- **Matrice d'adjacence** : (O(1000^2) = \textbf{1,000,000}) opérations.
- **Liste d'adjacence** : (O(1000 + 5000) = \textbf{6000}) opérations.

Un **gain très net** en temps de calcul et en mémoire est observé avec la liste d'adjacence pour ce type de graphe.

Ce qu’il faut retenir & Sources

Conclusion : Le choix de la représentation est intrinsèque à l’efficacité

- La **Matrice d’adjacence** est acceptable pour les petits graphes ou les graphes très denses.
- La **Liste d’adjacence** est largement préférable pour les graphes de grande taille et clairsemés, ce qui est très courant en pratique.

Cette analyse est cruciale pour guider l’implémentation et garantir la meilleure efficacité algorithmique lors de l’exploration d’un graphe.

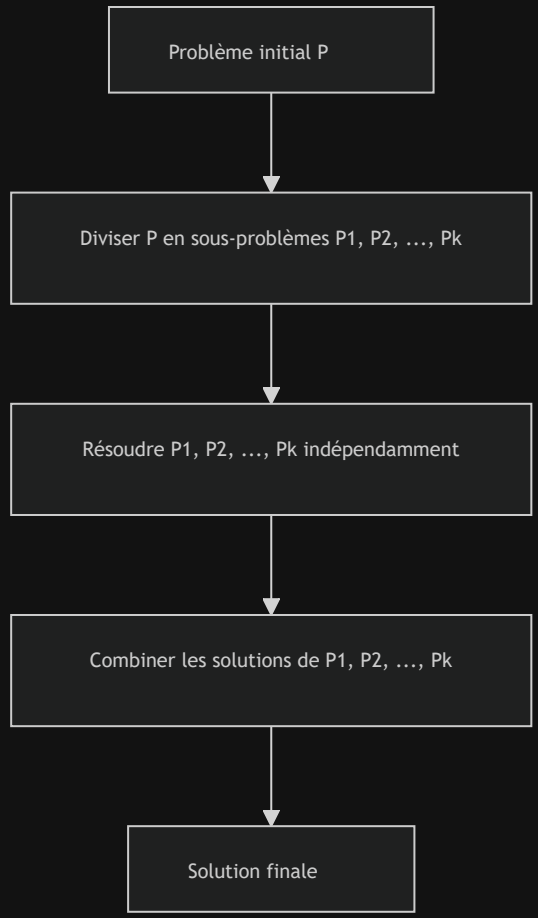
Sources consultées

- GeeksforGeeks — BFS and DFS Complexity
- Wikipedia — Graph traversal
- Programiz — Graph BFS and DFS
- TutorialsPoint — BFS and DFS

Le Principe "Diviser pour Régner"

Le paradigme **Divide & Conquer** (Diviser pour Régner) est une méthode fondamentale pour résoudre des problèmes complexes. Il repose sur trois étapes clés :

1. **Diviser** : Décomposer le problème en sous-problèmes plus petits, de même nature.
2. **Conquérir** : Résoudre récursivement chacun de ces sous-problèmes.
3. **Combiner** : Fusionner les solutions partielles pour obtenir la solution finale.


Problème initial P
Diviser P en sous-problèmes P1, P2, ..., Pk
Résoudre P1, P2, ..., Pk indépendamment
Combiner les solutions de P1, P2, ..., Pk
Solution finale

Exemple Classique : Le Tri Fusion (Merge Sort)

Le Tri Fusion est une illustration parfaite du Diviser pour Régner :

- **Diviser** : Le tableau à trier est scindé en deux moitiés.
- **Conquérir** : Chaque moitié est triée récursivement.
- **Combiner** : Les deux moitiés triées sont fusionnées pour former un tableau trié complet.

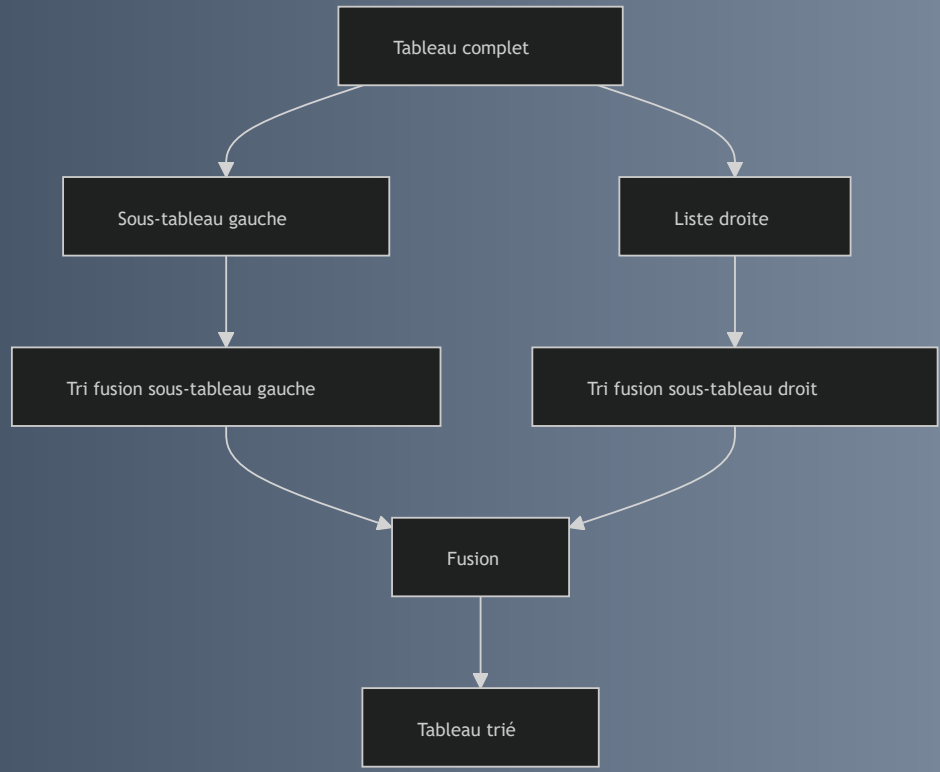


Tableau complet
Sous-tableau gauche
Liste droite
Tri fusion sous-tableau gauche
Tri fusion sous-tableau droit
Fusion
Tableau trié

Pourquoi utiliser "Diviser pour Régner" ?

Ce paradigme offre plusieurs avantages significatifs :

- **Simplifie la résolution** de problèmes complexes.
- Souvent, il conduit à des algorithmes **très efficaces** (ex: (O(n \log n)) pour le tri fusion).
- Favorise une **approche récursive**, améliorant la clarté et la modularité du code.
- Permet des **optimisations** sur les architectures de calcul parallèles.

Applications et Complexité

Le "Diviser pour Régner" se retrouve dans de nombreux algorithmes clés :

- **Recherche binaire** : Réduit l’espace de recherche de moitié.
- **Multiplication rapide de matrices** (algorithme de Strassen).
- **FFT (Fast Fourier Transform)** : Décomposition du problème en moitiés.

Pour analyser la complexité d'un problème de taille (n) divisé en (k) sous-problèmes de taille (n/b) avec une opération de fusion en (O(n^d)), on utilise la récurrence :

$$T(n) = k * T(n/b) + O(n^d)$$

Interprétation terme par terme

- $T(n)$: Temps nécessaire pour résoudre un problème de taille n
- k : Nombre de sous-problèmes créés → on découpe le problème en k morceaux
- n/b : Taille de chaque sous-problème → chaque sous-problème est b fois plus petit
- $T(n/b)$: Temps pour résoudre un sous-problème
- $O(n^d)$: Coût du travail effectué en dehors de la récursion (découpage + fusion des résultats)

Ce qu’il faut retenir & Ressources

Le Diviser pour Régner est un paradigme essentiel : Il est la base de nombreux algorithmes performants. Sa compréhension est cruciale pour concevoir et analyser des algorithmes récursifs efficaces et bien structurés.

Sources consultées :

- GeeksforGeeks — Divide and Conquer Algorithm
- Wikipedia — Divide and Conquer
- Programiz — Divide and Conquer
- TopCoder Tutorial — Divide and Conquer

Théorème Maître pour Divide & Conquer

Signification de la récurrence

On considère un problème de taille (n).

La formule générale

$$T(n) = k * T(n/b) + O(n^d)$$

Interprétation terme par terme

- **(T(n))** Temps nécessaire pour résoudre un problème de taille (n)
- **(k)** Nombre de sous-problèmes créés → on découpe le problème en **(k)** morceaux
- **(n/b)** Taille de chaque sous-problème → chaque sous-problème est **(b) fois plus petit**
- **T(n/b)** Temps pour résoudre un sous-problème
- **(O(n^d))** Coût du travail effectué **en dehors de la récursion** (découpage + fusion des résultats)

Résultat de l’analyse (théorème maître)

On compare : [k et b^d]

Cas 1 — Sous-problèmes dominants

[k > b^d ==> T(n) = O(n^{log_b k})]

Cas 2 — Équilibre parfait

[k = b^d ==> T(n) = O(n^d log n)]

Cas 3 — Fusion dominante

[k < b^d ==> T(n) = O(n^d)]

Exemples classiques

Tri fusion

- $k = 2$
- $b = 2$
- $d = 1$

$$T(n) = 2T(n/2) + O(n)$$

$$T(n) = O(n \log n)$$

Recherche binaire

- $k = 1$
- $b = 2$
- $d = 0$

$$T(n) = T(n/2) + O(1)$$

$$T(n) = O(\log n)$$

Version « pédagogique » à retenir

> *Le temps de traitement d'un algorithme diviser pour régner se modélise par une récurrence de la forme*
> *T(n) = (nombre de sous-problèmes) x T(taille) + {coût hors récursion}*

Synthèse et Ressources

- **Ce qu'il faut retenir :**
  - Comprendre et appliquer le Théorème Maître permet d'évaluer rapidement la complexité des algorithmes récursifs de type Divide & Conquer sans résoudre explicitement la récurrence.
  - Cette méthode est un outil fondamental en analyse d'algorithmes récursifs.
- **Sources consultées :**
  - CLRS, Introduction to Algorithms — Master Theorem
  - GeeksforGeeks — Master Theorem
  - Big-O Cheat Sheet — Master Theorem
  - Programiz — Divide and Conquer

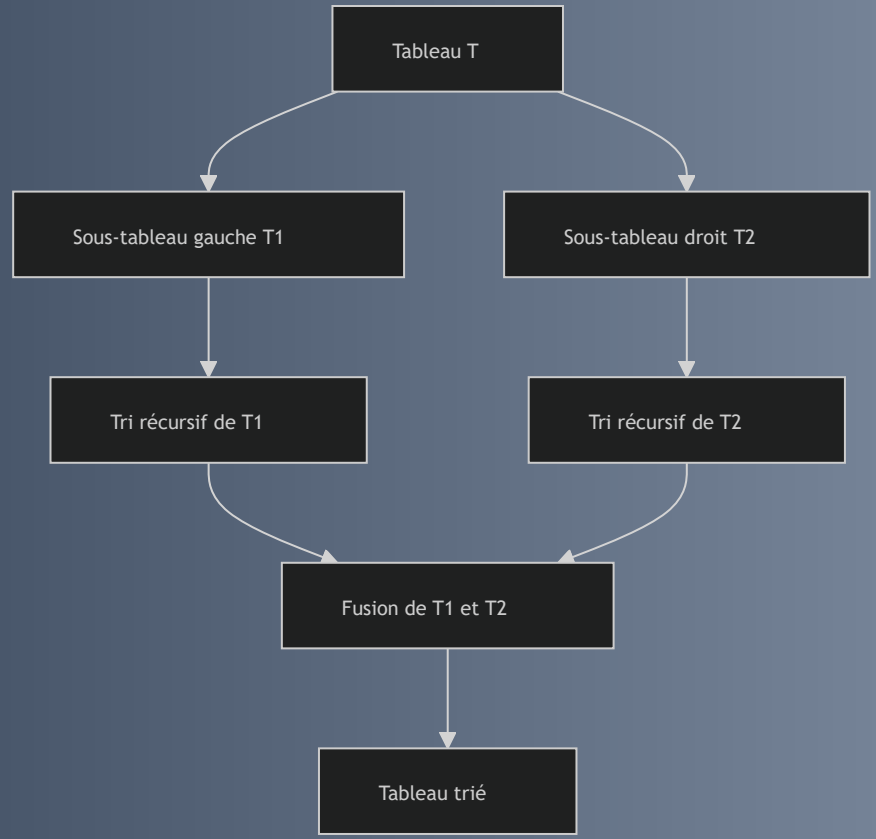
Merge Sort : Tri par Divide & Conquer

Algorithme de tri basé sur le paradigme **Divide & Conquer**.

Fonctionnement en 3 étapes :

1. **Diviser** le tableau à trier en deux moitiés.
2. **Trier récursivement** chaque moitié.
3. **Fusionner** les deux tableaux partiellement triés pour obtenir un tableau entièrement trié.

Propriété : Assure un tri **stable**, ce qui signifie que l'ordre des éléments égaux est conservé.



- La division s’effectue jusqu’à obtenir des sous-tableaux unitaires (n=1), automatiquement triés.
- La fusion combine deux tableaux triés en temps linéaire (O(n)).

Logique de Fusion :

- Compare les éléments des deux sous-tableaux triés (`tab[i]` et `tab[j]`), insérant le plus petit dans un tableau temporaire (`temp[]`).
- Copie les éléments restants d'un sous-tableau si l'autre est épuisé.
- Recopie le contenu du tableau temporaire trié dans le tableau original (`tab[]`).
- *Nécessite de la mémoire auxiliaire pour le tableau temporaire.*

Exemple de fusion de deux tableaux triés : Fusionner `[2,5,7]` et `[1,3,6,8]` produit `[1,2,3,5,6,7,8]`.

- **Mécanisme :** Comparer les éléments tête à tête, prendre le plus petit et avancer dans son tableau, répéter jusqu'à épuisement.

Analyse de la Complexité

La complexité est caractérisée par la récurrence :

$$T(n) = 2 * T(n/2) + O(n)$$

- Deux sous-problèmes de taille (n/).
- Fusion des résultats en temps linéaire (O(n)).

Via le théorème maître, on obtient :

$$T(n) = O(n \log n)$$

Note : Cette complexité est **stable**, peu importe la disposition initiale des éléments, garantissant (O(n log n)) même dans le pire cas.

Implémentation du Merge Sort

Logique Récursive (`mergeSort`) :

```
void mergeSort(int tab[], int deb, int fin) {
    if (deb < fin) {
        int mid = (deb + fin) / 2;
        mergeSort(tab, deb, mid);       // Diviser & Trier (gauche)
        mergeSort(tab, mid +1, fin);    // Diviser & Trier (droite)
        fusion(tab, deb, mid, fin);     // Combiner (fusion)
    }
}
```

L’Essentiel et Ressources

Points Clés à retenir :

- **Stable** : préserve l’ordre des éléments égaux.
- **Complexité garantie** (O(n \log n)) même dans le pire cas.
- Usage intensif de **mémoire auxiliaire** due à la fusion.

Conclusion : Merge Sort illustre la puissance du paradigme Divide & Conquer : diviser un problème en sous-problèmes plus gérables et combiner efficacement leurs solutions permet un tri optimal, stable et prévisible en temps d’exécution.

Sources Consultées :

- Wikipedia — Merge Sort
- GeeksforGeeks — Merge Sort
- Programiz — Merge Sort
- Big O Cheat Sheet — Merge Sort

Quick Sort — Divide & Conquer

- **Le Quick Sort** est un algorithme de tri basé sur le paradigme **Divide & Conquer**.
- Il vise à trier un tableau efficacement en le divisant.

Son principe :

1. **Choisir un pivot** dans le tableau.
2. **Partitionner** les éléments : <= pivot à gauche, > pivot à droite.
3. **Trier récursivement** chaque sous-tableau ainsi créé.
4. La concaténation de ces sous-tableaux donne le tableau entièrement trié.

Quick Sort : Mécanisme Détaillé

- **Le Partitionnement :**
  - Le pivot divise le tableau en deux sous-parties distinctes.
  - Tous les éléments inférieurs ou égaux au pivot sont placés à sa gauche.
  - Tous les éléments supérieurs au pivot sont placés à sa droite.
  - Cette étape est souvent réalisée par un algorithme qui échange les éléments en place (sans mémoire additionnelle significative).
- **La Récursivité :**
  - Le processus de Quick Sort est appliqué de manière récursive sur le sous-tableau gauche et le sous-tableau droit.
  - La récursivité s’arrête lorsque la taille d’un sous-tableau est de 1 (un élément est considéré comme trié).

L’Impact du Choix du Pivot

Le choix du pivot influence fortement la performance de l’algorithme :

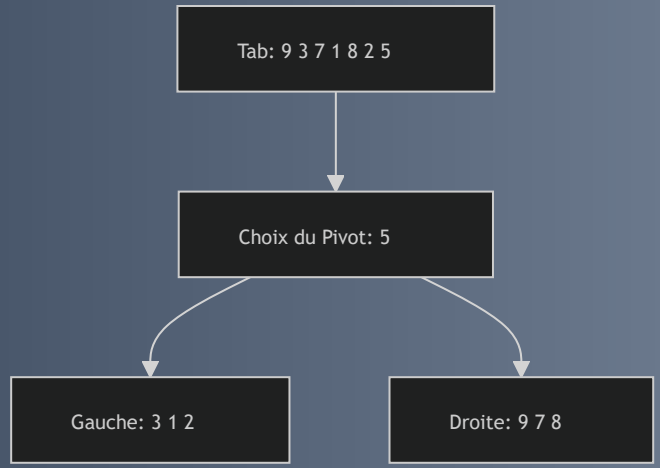
- **Pivot médian :** Assure un bon équilibre des sous-tableaux, menant à une performance optimale. En pratique, il est souvent approximé.
- **Premier élément / Dernier élément :** Facile à implémenter, mais peut entraîner des déséquilibres importants si les données sont déjà partiellement triées.
- **Pivot aléatoire :** Améliore la chance d’obtenir un équilibrage moyen des sous-tableaux, réduisant le risque du pire cas.
- **Médiane de trois :** Technique heuristique consistant à choisir la médiane entre le premier, le milieu et le dernier élément du tableau. Elle vise à mieux équilibrer les partitions.

Analyse de Complexité

Soit n la taille du tableau.

Cas	Complexité en temps	Commentaire
Meilleur cas	O(n log n)	Le pivot équilibre idéalement le tableau.
Cas moyen	O(n log n)	Performance attendue en moyenne.
Pire cas	O(n^2)	Le pivot est très mal choisi (ex: tableau déjà trié et pivot toujours le min/max).

Exemple de partitionnement :



```c
#include <stdio.h>
void echanger(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int tab[], int debut, int fin) {
    int pivot = tab[fin]; // Le dernier élément est choisi comme pivot
    int i = debut - 1;

    for (int j = debut; j < fin; j++) {
        if (tab[j] ⩽ pivot) {
            i++;
            echanger(&tab[i], &tab[j]); // Échange pour placer les petits éléments à gauche
        }
    }
    echanger(&tab[i + 1], &tab[fin]); // Place le pivot à sa position finale
    return i + 1; // Retourne l'indice du pivot
}

void quickSort(int tab[], int debut, int fin) {
    if (debut < fin) {
        int pi = partition(tab, debut, fin); // Effectue le partitionnement
        quickSort(tab, debut, pi - 1);       // Trie récursivement le sous-tableau gauche
        quickSort(tab, pi + 1, fin);         // Trie récursivement le sous-tableau droit
    }
}
// int main() { /* ... */ } // Exemple d'utilisation non détaillé ici
```

Quick Sort : Ce qu’il faut retenir et aller plus loin

Points clés :

- L’algorithme trie **en place**, minimisant l’utilisation de mémoire auxiliaire.
- Sa performance est fortement conditionnée par la qualité du **choix du pivot**.
- Il est sensible à l’ordre initial des données.
- Le Quick Sort est très répandu en pratique pour sa rapidité moyenne et son faible encombrement mémoire.

En bref : Quick Sort est un tri rapide souvent préféré au tri fusion lorsqu’une faible utilisation mémoire est requise. Son efficacité repose sur un bon choix du pivot pour éviter la dégradation en temps quadratique.

Sources consultées :

- Wikipedia — Quicksort
- GeeksforGeeks — Quick Sort Algorithm
- Programiz — Quick Sort
- Big O Cheat Sheet — Quick Sort

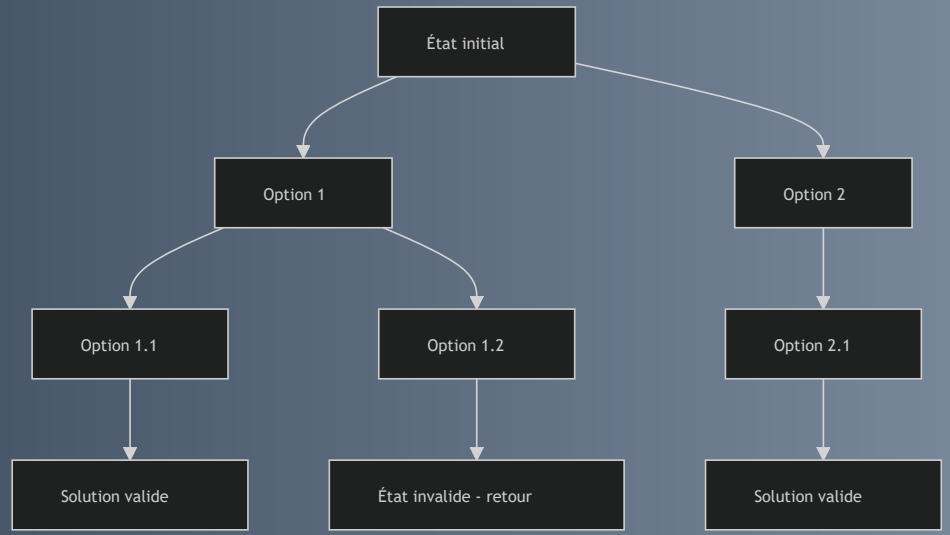
Introduction au Backtracking

Le **Backtracking** est une méthode d'exploration systématique d'un espace d'états pour résoudre des problèmes combinatoires.

Principe général : Il consiste à construire progressivement des solutions candidates en **explorant** un arbre implicite d'états, et à revenir **en arrière** (retour sur trace) dès que la continuité d'une solution paraît impossible.

Mécanisme clé : Exploration et Retour sur Trace

1. **Modélisation :** Le problème est modélisé par un **espace d'états** où chaque nœud représente une solution partielle.
2. **Construction récursive :** On construit récursivement la solution en ajoutant des éléments.
3. **Retour arrière :** Si un choix mène à un état invalide, on **recule** (backtrack) pour essayer une autre option.



Dès qu'un **état invalide** (exemple D2) est rencontré, l'algorithme revient à la décision précédente pour en tester une autre.

Exemple : Le Problème des N Reines

Problème : Placer (n) reines sur un échiquier (n x n) sans qu'aucune ne se menace mutuellement.

Application du Backtracking :

- L'exploration se fait **ligne par ligne**.
- On tente de placer une reine dans chaque colonne de la ligne courante.
- Dès qu'une reine ne peut être placée dans aucune colonne d'une ligne donnée (car toutes les positions sont menacées), on **revient à la ligne précédente** pour changer la position de la reine qui s'y trouve et explorer d'autres chemins.

Caractéristiques du Backtracking

Le Backtracking est une approche puissante grâce à ses propriétés :

- **Complet :** Il explore toutes les solutions possibles si aucune coupe n'est appliquée.
- **Efficace :** Son efficacité est fortement améliorée avec une bonne **stratégie de coupe** (pruning) pour éliminer rapidement les branches invalides de l'arbre d'exploration.
- **Approche récursive :** Il s'implémente naturellement de manière récursive.

Algorithme du Backtracking (Pseudocode)

La structure générale d'un algorithme de Backtracking :

```
fonction backtrack(état courant)
    si état courant est solution complète
        afficher solution
    sinon
        pour chaque choix possible à partir de état courant
            si choix possible est valide
                backtrack(prolongement)  // Appel récursif avec le nouveau choix
                annuler choix            // Retour sur trace : annuler le choix pour essayer d'autres options
```

Ce pseudocode illustre le processus d'exploration et de "retour sur trace" essentiel au Backtracking.

Ce qu’il faut retenir & Références

Ce qu’il faut retenir : Le Backtracking est un paradigme puissant pour résoudre les problèmes où la solution peut être construite étape par étape et invalidée dynamiquement. La maîtrise du retour sur trace et des stratégies d’élagage (pruning) optimise considérablement la recherche des solutions.

Sources consultées :

- GeeksforGeeks - Backtracking
- Wikipedia - Backtracking
- Programiz - Backtracking
- TopCoder - Backtracking Introduction

Paradigmes Avancés - Backtracking

- **Cas typiques du Backtracking :**
  - Le Problème des N-Dames
  - La Résolution de labyrinthe
  - Le Sudoku

1. Le Problème des N-Dames

Description

Placer (N) dames sur un échiquier (N x N) de sorte qu'aucune ne menace une autre (pas sur la même ligne, colonne ou diagonale).

Caractéristiques du Backtracking appliquées

- **Exploration ligne par ligne :** On place une reine par ligne.
- **Test systématique :** À chaque ligne, on teste chaque colonne possible.
- **Retour sur trace (Recul) :** Dès qu'aucune colonne ne permet une position valide (conflit avec une reine déjà placée), on revient sur le choix précédent.

Le Problème des N-Dames : Illustration & Complexité

Flux de décision (Simplifié)

Complexité

- **Exponentielle** en général.
- Les "coupes" (branches invalides non explorées) améliorent significativement l'efficacité en évitant d'explorer des chemins inutiles.

2. Résolution de Labyrinthe

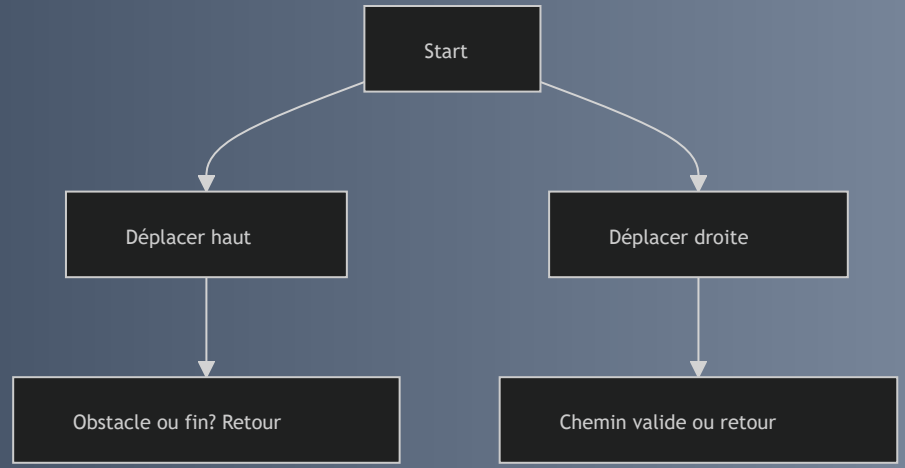
Description

Trouver un chemin de la cellule de départ à la cellule d'arrivée dans une grille avec obstacles.

Modélisation et Application du Backtracking

- **Espace d'état :** Représente les positions possibles dans la grille.
- **Génération d'états :** Chaque état permet des mouvements valides dans les directions autorisées.
- **Stratégie de recherche :** Le Backtracking essaye une direction, et si un cul-de-sac est rencontré (pas de chemin valide), il revient sur ses pas pour explorer une autre direction.

Flux de décision (Simplifié)



3. Le Sudoku

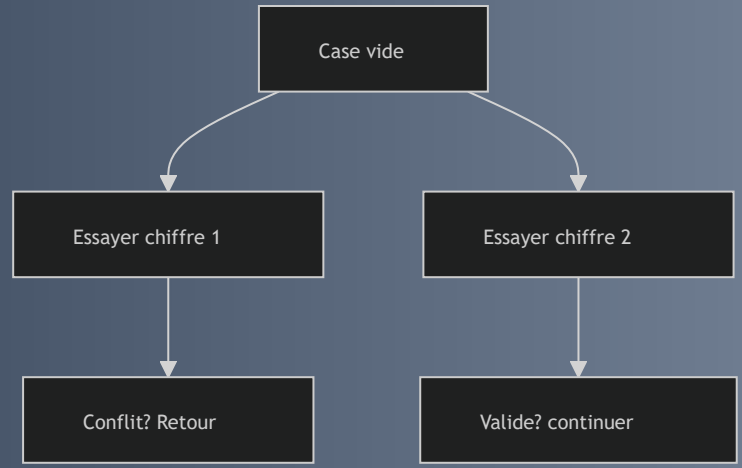
Description

Remplir la grille (9 x 9) pour que chaque ligne, colonne et région 3x3 contienne les chiffres 1 à 9 sans répétition.

Backtracking appliqué

- **Recherche :** Localiser une case vide.
- **Essai :** Tenter d'y placer les chiffres valides un par un (respectant les règles du Sudoku).
- **Retour sur trace :** Si un choix bloque la complétion ultérieure de la grille, le système revient sur le choix et essaie le chiffre suivant.

Flux de décision (Simplifié)



Remarque

Le backtracking peut être optimisé avec des heuristiques (ex: choisir les cases avec le plus de contraintes en premier, ou propager l'impact d'un choix).

Conclusion & Références

Ce qu’il faut retenir

Ces problèmes illustrent la force du backtracking dans des espaces de recherche vastes. La méthode explore systématiquement les solutions possibles tout en revenant sur ses choix dès qu’un blocage est détecté. Le succès au-delà de la naïveté repose sur l’utilisation d’heuristiques pour guider l’exploration.

Sources consultées

- GeeksforGeeks — N Queens Problem
- Wikipedia — Maze solving algorithm
- Sudoku Solving — Backtracking
- Programiz — Backtracking example Sudoku

Exploration des Solutions Combinatoires

- **Rappel :** Le backtracking explore récursivement toutes les solutions candidates à un problème combinatoire.
- Il construit des solutions partielles étape par étape.
- Il revient en arrière ("backtracking") dès qu’une solution partielle ne peut mener à une solution complète valide.

Les 5 Étapes Clés d’un Algorithme de Backtracking

Structure Générique du Backtracking Récursif

1. **Test de terminaison ou solution complète :** L’état courant représente-t-il une solution valide ?
2. **Génération des choix possibles :** Lister les prochaines décisions possibles à partir de l’état courant.
3. **Validation des choix :** Vérifier la compatibilité de chaque choix avec les contraintes du problème.
4. **Récursion :** Appliquer récursivement l’algorithme sur l’état modifié pour chaque choix valide.
5. **Retour sur trace (undo) :** Annuler le choix avant d’explorer les autres possibilités.

Modèle Pseudocode de l’Algorithme

La Logique Récursive en Action

```
fonction backtrack(état_courant)
    si état_courant est une solution complète
        afficher_solution(état_courant)
        retourner

    pour chaque choix possible dans génération_choix(état_courant)
        si choix est valide
            appliquer_choix(état_courant, choix)
            backtrack(état_courant)
            annuler_choix(état_courant, choix)
```

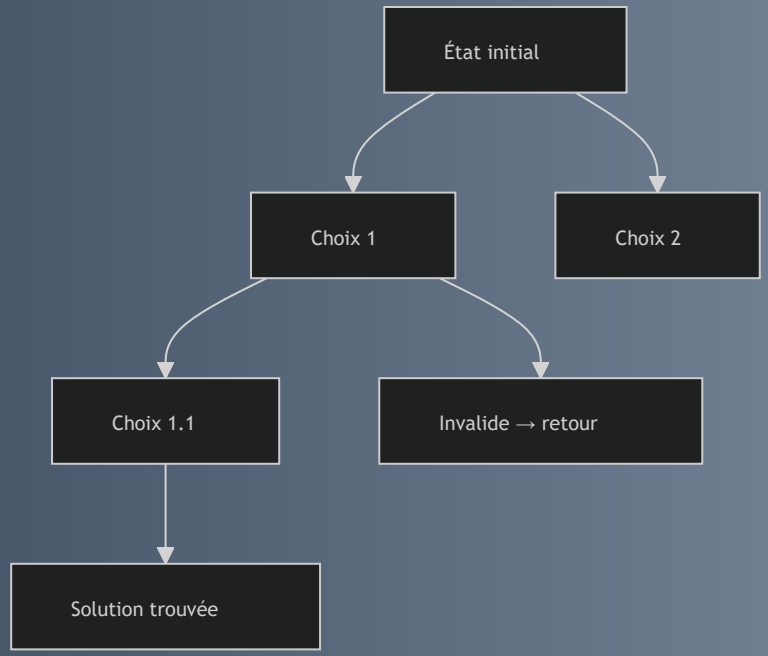
Application Concrète : Le Problème des N-Dames

Adapter le Backtracking à un Cas Pratique

- **État courant :** Positions des reines déjà placées sur les lignes précédentes.
- **Génération des choix :** Colonnes libres sur la ligne courante.
- **Validité :** Aucune attaque diagonale, horizontale ou verticale avec les reines existantes.
- **Application du choix :** Placer la reine sur une colonne valide.
- **Annulation du choix :** Retirer la reine (backtrack).

Comprendre l’Exploration et le Retour

Visualisation du Processus de Backtracking



L’algorithme explore une branche, revient en arrière lorsqu’elle est invalide ou qu’une solution est trouvée.

Ce qu’il Faut Retenir & Ressources

Synthèse et Aller Plus Loin

Points importants :

- **Gestion explicite du retour :** Chaque choix doit être annulé pour permettre l’exploration d’autres possibilités.
- **Efficacité grâce à l’élagage :** La validation anticipée des choix évite d’explorer des branches inutiles.
- **Utilisation de structures de données adaptées :** Elles facilitent l’annulation des choix (ex: pile, tableau).

Cette architecture récursive du backtracking constitue un cadre général adaptable à de nombreux problèmes combinatoires, permettant d’explorer toutes les solutions possibles de manière méthodique et contrôlée.

Sources consultées :

- GeeksforGeeks — Backtracking Pseudocode
- Wikipedia — Backtracking
- Programiz — Backtracking Tutorial
- TopCoder — Backtracking guide

Introduction à la Programmation Dynamique (PD)

- Une méthode algorithmique pour résoudre efficacement des problèmes de décision, optimisation ou de comptage.
- **Objectif clé :** Éviter les calculs redondants en mémorisant les résultats de sous-problèmes déjà résolus.

Les Fondements : Quand utiliser la PD ?

Deux propriétés fondamentales doivent être vérifiées :

1. **Sous-problèmes chevauchants (Overlapping subproblems)**
    1. Le problème peut être décomposé en sous-problèmes de plus petite taille.
    2. Ces sous-problèmes se répètent plusieurs fois dans la résolution globale.
    3. **Sans PD :** Risque de recalculer les mêmes résultats de nombreuses fois (complexité exponentielle).

Exemple : Calcul du (n)-ième terme de la suite de Fibonacci

- Définition récursive : (F(n) = F(n-1) + F(n-2))
- Les sous-problèmes (F(n-1)) et (F(n-2)) se recoupent lors de nombreux appels.
- Mémoriser (F(k)) pour tout (k) évite cette répétition.

Les Fondements : Quand utiliser la PD ?

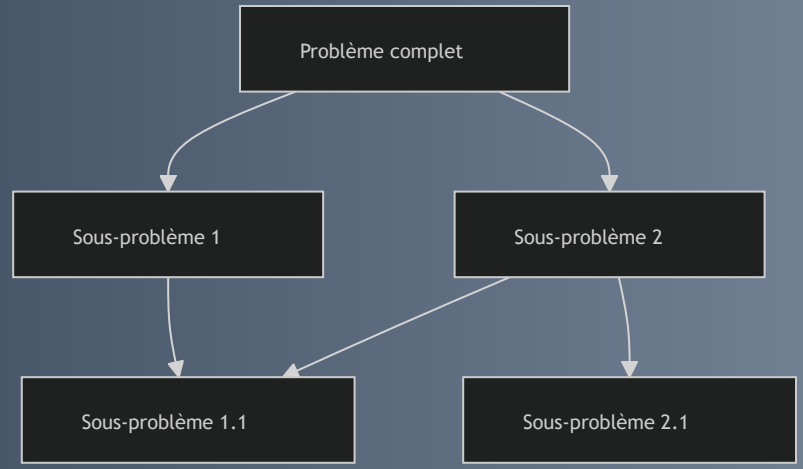
Deux propriétés fondamentales doivent être vérifiées :

2. **Sous-structure optimale (Optimal substructure)**
    1. La solution optimale globale se construit à partir des solutions optimales de ses sous-problèmes.
    2. Toutes les solutions partielles optimales concourent à la solution finale optimale.

Exemple : Problème du plus long sous-ensemble croissant

- La séquence optimale jusqu'à un indice (i) dépend des solutions optimales jusqu'à (j < i).

Relation entre les sous-problèmes



- Un même sous-problème peut être atteint par plusieurs chemins de décomposition
- Ces sous-problèmes sont dits chevauchants
- Le sous-problème D est utilisé à la fois par B et C
- Sans mémorisation, il serait recalculé plusieurs fois

Exemple Pratique : Fibonacci Naïf vs PD

Fibonacci naïf (récursif simple)

```
int fib(int n) {
    if (n ⩽ 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

- Complexité exponentielle O(2^n).
- Beaucoup de recalculs.

Fibonacci avec programmation dynamique (mémorisation)

```
int fib(int n, int memo[]) {
    if (memo[n] ≠ -1)
        return memo[n];
    if (n ⩽ 1)
        memo[n] = n;
    else
        memo[n] = fib(n-1, memo) + fib(n-2, memo);
    return memo[n];
}
```

- Complexité O(n).
- Résultats intermédiaires calculés une seule fois.

Synthèse des Propriétés Clés

Propriété	Description	Détection pratique
Sous-problèmes chevauchants	Les mêmes sous-problèmes apparaissent souvent dans la récursion.	Vérifier si le graphe de récursion contient des nœuds partagés.
Sous-structure optimale	La solution globale se construit à partir de solutions optimales de ses sous-problèmes.	Chercher une relation récursive qui optimise à chaque étape.

Ce qu’il faut retenir & Sources

Ce qu’il faut retenir

- La programmation dynamique transforme des problèmes combinatoires potentiellement exponentiels en solutions plus maniables.
- Elle est applicable si les propriétés de sous-problèmes chevauchants et de sous-structure optimale sont présentes.
- La mémorisation systématique des résultats intermédiaires est la clé de sa puissance.

Sources consultées

- Wikipedia — Programming Dynamic
- GeeksforGeeks — Introduction to Dynamic Programming
- Programiz — Dynamic Programming
- Big O Cheat Sheet — Dynamic Programming

Programmation Dynamique : Au-delà de la Récursion Naïve

Qu’est-ce que la récursion simple ?

Une technique où une fonction s’appelle elle-même pour résoudre des problèmes plus petits. C’est intuitif, mais…

Ses limites

La récursion naïve peut entraîner une **explosion du nombre d'appels**.

Exemple classique : Calcul de la suite de Fibonacci

La Récursion Simple en Pratique : Le Cas Fibonacci

```c
int fib(int n) {
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

Problème majeur : Redondance des calculs

- De nombreux sous-problèmes sont recalculés plusieurs fois.
- L'arbre d'appels est exponentiel en taille O(2^n).
- Impact fort sur les performances.

Programmation Dynamique : L’Art d’Éliminer la Redondance

Principe

La Programmation Dynamique (PD) optimise la récursion en **mémorisant** les résultats des sous-problèmes dès qu’ils sont calculés. Cette technique s’appelle la **mémoïsation**.

Comment ça marche ?

1. **Stocker** dans une table (tableau, dictionnaire) les résultats partiels déjà calculés.
2. **Vérifier** si le résultat d’un argument donné existe déjà avant tout appel récursif.
3. **Retourner directement** le résultat si celui-ci est déjà connu.

La Programmation Dynamique en Action : Fibonacci Optimisé

```
int fibMemo(int n, int memo[]) {
    if (memo[n] ≠ -1) // Résultat déjà calculé ?
        return memo[n];
    if (n ≤ 1)
        memo[n] = n;
    else
        memo[n] = fibMemo(n-1, memo) + fibMemo(n-2, memo);
    return memo[n];
}
```

Explication

- Le tableau `memo` est initialisé à (-1).
- Chaque valeur calculée est stockée.
- Chaque sous-problème est résolu une seule fois.

Avantage clé : Complexité améliorée : (O(n)) !

Pourquoi la Programmation Dynamique est Essentielle ?

Autres avantages

- **Décomposition claire du problème :** facilite la compréhension et la structure.
- **Réduction drastique du temps d'exécution** pour les problèmes avec sous-problèmes chevauchants.
- Permet aussi une **implémentation itérative** (sans récursion), appelée "bottom-up".

Récapitulatif & Ressources

Résumé des différences

Aspect	Récursion simple	Programmation Dynamique
Répétition calculs	Oui, calculs redondants fréquents	Non, résultats mémorisés
Temps d’exécution	Exponentiel pour certains problèmes	Polynomial en général
Complexité	Souvent impraticable sur grands cas	Optimisation efficace des calculs interdépendants
Mémoire	Faible (pile)	Nécessite une structure mémoire supplémentaire

Ce qu’il faut retenir

L’optimisation offerte par la programmation dynamique rend possible la résolution de problèmes complexes, dont la récursion pure serait inefficace à cause de la redondance des calculs. La mémoïsation ou la construction de solutions par "bottom-up" sont les clés de cette efficacité.

Sources consultées

- GeeksforGeeks — Difference Between Recursion and Dynamic Programming
- Wikipedia — Dynamic Programming
- Programiz — Dynamic Programming Tutorial
- TopCoder Tutorial — Dynamic Programming vs Recursion

Programmation Dynamique : Mémoïsation (Top-Down)

Qu’est-ce que la Mémoïsation ?

- **Définition**
  - Technique algorithmique consistant à **mémoriser les résultats de sous-problèmes calculés** pour éviter de les recalculer plusieurs fois.
  - Fait partie de la **Programmation Dynamique**.

- **Approche Top-Down**
  - Résout le problème global en partant de l’état complet.
  - Décompose le problème en sous-problèmes.

Principe Général et Structure

Principe

1. Écrire l'algorithme en **récursion naturelle**.
2. Avant de calculer un sous-problème, **vérifier s'il est déjà résolu** (consultation mémoire).
3. Si oui, renvoyer le résultat mémorisé.
4. Sinon, calculer le résultat, puis le **stocker** pour usage futur.
    1. *Objectif :* Éviter la redondance de calculs, caractéristique de la récursion simple.

Structure Typique (pseudo-code C)

```c
int fonctionMemo(int n, int memo[]) {
    if (memo[n] ≠ -1) // Déjà calculé ?
        return memo[n]; // Retourne le résultat mémorisé

    if (condition_terminaison)
        memo[n] = valeur_base;
    else
        memo[n] = combinaison_des_appels_recursifs;

    return memo[n]; // Stocke et retourne le résultat
}
```

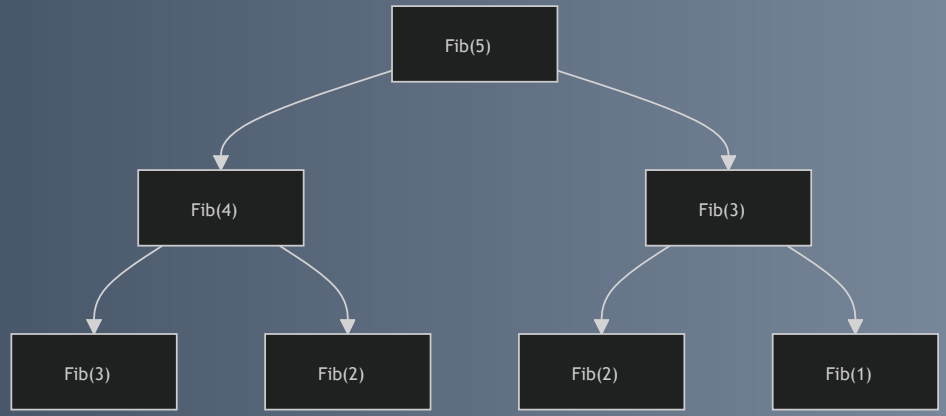
`memo[]` est un tableau initialisé à une valeur indiquant "non traité" (ex: -1).

Exemple : Calcul de Fibonacci avec Mémoïsation

Implémentation en C

```c
#include <stdio.h>

int fibMemo(int n, int memo[]) {
    if (memo[n] =/= -1) // Vérifie si Fib(n) est déjà calculé
        return memo[n];

    if (n ⩽ 1)
        memo[n] = n; // Cas de base : Fib(0)=0, Fib(1)=1
    else
        // Calcul et mémorisation du résultat
        memo[n] = fibMemo(n-1, memo) + fibMemo(n-2, memo);

    return memo[n]; // Retourne le résultat mémorisé
}

int main() {
    int n = 10;
    int memo[11];
    // Initialisation du tableau memo à -1 (non calculé)
    for (int i = 0; i ⩽ n; i++)
        memo[i] = -1;

    printf("Fibonacci(%d) = %d\n", n, fibMemo(n, memo));
    return 0;
}
```

Visualisation de l’approche Top-Down (Fibonacci)



- **Optimisation:** Les calculs de `Fib(3)` et `Fib(2)` sont effectués **une seule fois**.
- Leurs résultats sont ensuite **réutilisés** depuis la mémoire lorsque de nouvelles branches de récursion en ont besoin, évitant ainsi des recalculs coûteux.

Comparaison et Extensibilité

Mémoïsation vs. Récursion Simple

Aspect	Récursion simple	Mémoïsation (top-down)
Complexité	Exponentielle (ex: Fibonacci (O(2^n)))	Polynomiale (ex: Fibonacci (O(n)))
Sous-problèmes	Calcul multiple des mêmes	Calcul unique, mémorisation
Espace mémoire	Réduit (pile récursive)	Supplémentaire pour `memo[]`

Extensibilité de la Mémoïsation

- Peut gérer des sous-problèmes **multidimensionnels** (ex: `memo[i][j]`).
- Applicable aux problèmes d'**optimisation** et de **comptage**.
- Facilement adaptable avec des structures associatives (dictionnaires/hashmaps).

Ce qu’il faut retenir & Ressources

En bref

La mémoïsation optimise la récursion en exploitant la persistance des résultats intermédiaires. Elle permet d’aborder des problèmes complexes qui seraient inaccessibles par une récursion naïve, tout en conservant une structure algorithmique claire et naturelle.

Sources Consultées

- GeeksforGeeks — Memoization in Dynamic Programming
- Wikipedia — Memoization
- Programiz — Dynamic Programming Tutorial
- TopCoder — Introduction to Dynamic Programming

Programmation Dynamique : Approche Bottom-Up

La **tabulation** est une approche de programmation dynamique qui :

- **Résout explicitement les plus petits sous-problèmes en premier.**
- Construit progressivement la solution finale.
- Part "du bas" (bottom-up), évitant la récursion.
- Remplace la structure top-down par un simple parcours itératif.

Tabulation vs. Mémoïsation : Quelles Différences ?

Deux approches de Programmation Dynamique

Caractéristique	Mémoïsation (Top-down)	Tabulation (Bottom-up)
Mode d’exécution	Récursion avec cache	Itération avec tableau
Ordre de résolution	Sous-problèmes nécessaires à la demande	De petits sous-problèmes vers grands
Complexité mémorielle	Peut souffrir du coût récursif	Allocation et parcours contrôlés
Facilite l’analyse	Plus naturelle à écrire	Plus facile à comprendre les dépendances

Le Modèle Générique de la Tabulation

Comment implémenter une solution bottom-up

1. **Initialiser un tableau** : Souvent à une taille correspondant à la taille du problème.
2. **Initialiser les cas de base** : Les solutions des plus petits sous-problèmes.
3. **Itérer sur les indices/cas** : S’assurer que les sous-problèmes nécessaires sont déjà résolus.
4. **Remplir le tableau progressivement** : Chaque cellule dépend des précédentes.
5. **Retourner la solution** : Stockée dans la cellule finale du tableau.

Exemple : Fibonacci par Tabulation

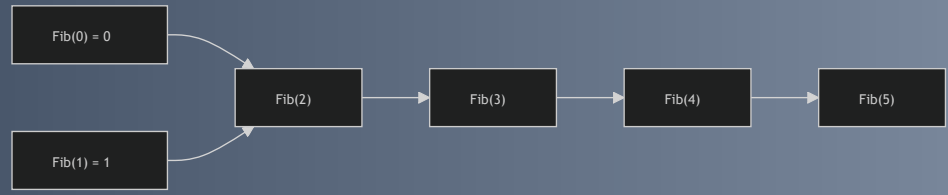
Du calcul récursif à l'itération bottom-up

Algorithme

- On calcule (F(0) = 0), (F(1) = 1) en base.
- Puis on construit (F(k) = F(k-1) + F(k-2)) pour (k) de 2 à (n).

Implémentation C

```c
int fib(int n) {
    if (n <= 1) return n;
    int fibTable[n+1];
    fibTable[0] = 0;
    fibTable[1] = 1;
    for (int i = 2; i <= n; i++) {
        fibTable[i] = fibTable[i-1] + fibTable[i-2];
    }
    return fibTable[n];
}
```



Chaque valeur est calculée uniquement après avoir calculé les dépendances (les deux précédentes).

Tabulation : Avantages, Limites & Applications

Quand et comment l’utiliser

Avantages

- Pas de surcharge liée à la récursion (pile d’appels).
- Contrôle explicite sur l’ordre d’évaluation.
- Parfaite maîtrise mémoire (allouer tableau suffisant).
- Plus simple à optimiser dans certains langages ou environnements.

Limites

- Nécessite de comprendre la **dépendance entre sous-problèmes**.
- Parfois moins intuitive que la récursion.
- Pour certains problèmes, moins flexible pour gestion cas particuliers.

Applications Classiques

- Problème du sac à dos (Knapsack).
- Plus longue sous-séquence commune (LCS).
- Traversée de grille, chemins minimaux.
- Nombre de manières de faire un montant donné avec des pièces.

Ce qu’il faut retenir & Ressources

La Tabulation : Efficacité et Maîtrise

La tabulation est une méthode rigoureuse et efficace qui exploite pleinement les dépendances entre sous-problèmes, s’appuyant sur une construction ordonnée des solutions partielles et souvent privilégiée pour son exécution rapide et sa simplicité d’implémentation itérative.

Sources consultées

- GeeksforGeeks — Dynamic Programming Tabulation
- Wikipedia — Dynamic programming
- Programiz — Tabulation in Dynamic Programming
- TopCoder Tutorial — Dynamic Programming