COURS : STRUCTURES DE DONNÉES ET ALGORITHMES DE TRI

Le premier thème de l'année est consacré aux Systèmes de Gestion de Bases de Données (SGBD). Ces systèmes sont conçus pour organiser, stocker et manipuler des données structurées à l'aide de requêtes.

Pour assurer un accès rapide et efficace à ces informations, ils reposent sur différentes structures de données, telles que les arbres, les tables de hachage, ainsi que sur des algorithmes de recherche optimisés.

Ce cours a pour objectif de réviser les structures et algorithmes de recherche étudiés en première année et d'introduire de nouvelles structures de données qui jouent un rôle essentiel dans le fonctionnement interne des bases de données.

I) ALGORITHMES DE TRI

Avant d'aborder les structures de données utilisées dans les bases de données, il est essentiel de revenir sur les principaux algorithmes de tri. En effet, le tri constitue une opération fondamentale en informatique, car de nombreux traitements, comme la recherche, la fusion de données ou la création d'index, reposent sur des ensembles de données ordonnés.

Les bases de données, qu'elles soient relationnelles ou non, utilisent des mécanismes de tri internes pour organiser les enregistrements, optimiser les requêtes et améliorer les performances globales du système. Comprendre le fonctionnement et la complexité des différents algorithmes de tri permet donc de mieux appréhender les choix d'implémentation faits dans les SGBD.

Dans cette partie, nous allons donc rappeler et comparer plusieurs algorithmes de tri classiques, en étudiant leur principe de fonctionnement, leur complexité temporelle et spatiale, ainsi que leurs avantages et limites.

Le tableau ci-dessous récapitule les algorithmes de tri les plus populaires :

Algorithme	**Complexité temporelle**			**Complexité spatiale**	**Stable ?**	**Remarques**
	Meilleur des cas	**Moyenne**	**Pire des cas**	**Pire des cas**		
Tri à bulles (Bubble Sort)	O(n)	O(n^2)	O(n^2)	O(1)	Oui	Non recommandé
Tri par sélection (Selection Sort)	O(n^2)	O(n^2)	O(n^2)	O(1)	Oui	Même un tableau déjà trié nécessite un scan complet
Tri par insertion (Insertion Sort)	O(n)	O(n^2)	O(n^2)	O(1)	Oui	Dans le meilleur des cas (tableau déjà trié), chaque insertion nécessite un temps constant
Tri par fusion (Merge Sort)	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	Oui	Nécessite O(n) espace mémoire sur un tableau
Tri rapide (Quick Sort)	O(nlogn)	O(nlogn)	O(n^2)	O(logn)	Non	Choisir un pivot aléatoire améliore la complexité (moyenne) de l'algorithme
Tri par tas (Heap Sort)	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	Non	En utilisant un tableau comme mémoire, la complexité spatiale peut être constante
Tri par comptage (Counting Sort)	O(n+k)	O(n+k)	O(n+k)	O(k)	Oui	k est l'intervalle des valeurs des clés non négatives
Tri par base (Radix Sort)	O(dn)	O(dn)	O(dn)	O(d+n)	Oui	Le tri par base est stable si l'algorithme de tri sous-jacent est stable

Tableau 1: Caractéristiques des principaux algorithmes de tri

Les systèmes de gestion de bases de données (SGBD) utilisent ces concepts :

- Le tri fusion est au cœur du tri externe, utilisé pour trier de très gros volumes de données stockés sur disque.
- Le tri par tas est lié à la gestion des files de priorité dans les moteurs de requêtes.
- Le tri rapide est souvent utilisé pour les tris en mémoire.
- Les tris non comparatifs (par comptage ou par base) inspirent certains algorithmes de tri spécialisés pour des données numériques indexées.

I.1. Tri à bulles

C'est l'algorithme de tri le plus simple.

Le tri à bulles fonctionne en parcourant plusieurs fois la liste à trier et en comparant, à chaque passage, les éléments adjacents pour les échanger s'ils sont dans le mauvais ordre, de sorte que les plus grands (ou les plus petits) « remontent » progressivement vers la fin du tableau, comme des bulles à la surface.

L'unique avantage que peut apporter cet algorithme par rapport à d'autres est qu'il peut détecter si les valeurs du tableau en entrée sont déjà triées ou non.

Par défaut, la complexité de l'algorithme est $O(n^2)$, même dans le meilleur des cas. Cependant, on peut améliorer sa rapidité en arrêtant l'algorithme prématurément si aucun échange ne s'est produit lors de la dernière itération. Dans ce cas, la complexité dans le meilleur des cas est réduite à O(n).

L'exemple ci-dessous illustre la première itération de cet algorithme :

Première passe →									**Remarques**
10	4	43	5	57	91	45	9	7	Échanger
4	10	43	5	57	91	45	9	7	Pas d'échange
4	10	43	5	57	91	45	9	7	Échanger
4	10	5	43	57	91	45	9	7	Pas d'échange
4	10	5	43	57	91	45	9	7	Pas d'échange
4	10	5	43	57	91	45	9	7	Échanger
4	10	5	43	57	45	91	9	7	Échanger
4	10	5	43	57	45	9	91	7	Échanger
4	10	5	43	57	45	9	7	91	À la fin du premier passage, 91 est à la bonne place

I.2. Tri par sélection

Le tri par sélection est un algorithme de tri en place. Il fonctionne bien pour les petits ensembles de données ou les listes presque triées.

Le tri par sélection améliore le tri à bulles en ne réalisant qu'un seul échange par passage dans la liste. Pour cela, le tri par sélection cherche la plus petite (ou la plus grande) valeur au cours d'un passage, puis, une fois ce passage terminé, place cette valeur à la bonne position. Comme pour le tri à bulles, après le premier passage, le plus grand élément est à sa place ;

après le deuxième passage, le deuxième plus grand est à sa place. Ce processus se poursuit et nécessite (n-1) passages pour trier n éléments, puisque le dernier élément se retrouve forcément à la bonne place après le (n-1)^{e} passage.

L'inconvénient du tri par sélection est sa complexité en O(n²).

L'exemple ci-dessous illustre le principe de tri sur l'ensemble des itérations en sélectionnant le plus grand élément à chaque passage. La première itération place 91, la seconde place 57, la troisième 45, etc … :

→									**Remarques : (sélectionne le plus grand élément)**
10	4	43	5	57	91	45	9	7	Échanger 91 et 7
10	4	43	5	57	7	45	9	91	Échanger 57 et 9
10	4	43	5	9	7	45	57	91	45 est le suivant plus grand, ignorer
10	4	43	5	9	7	45	57	91	Échanger 43 et 7
10	4	7	5	9	43	45	57	91	Échanger 10 et 9
9	4	7	5	10	43	45	57	91	Échanger 9 et 5
5	4	7	9	10	43	45	57	91	7 est le suivant plus grand, ignorer
5	4	7	9	10	43	45	57	91	Échanger 5 et 4
4	5	7	9	10	43	45	57	91	Liste triée

I.3. Tri par insertion

Le tri par insertion est un tri par comparaison simple et efficace. Dans cet algorithme, chaque itération retire un élément de la liste d'entrée et l'insère dans la sous-liste déjà triée. Le choix de l'élément retiré de la liste d'entrée est séquentiel, et ce processus se répète jusqu'à ce que tous les éléments aient été traités.

Il maintient en permanence une sous-liste triée dans les premières positions de la liste. Chaque nouvel élément est ensuite inséré dans cette sous-liste de manière à ce que la partie triée augmente d'un élément à chaque étape.

On commence par supposer qu'une liste contenant un seul élément (position 0) est déjà triée. À chaque passage, pour chaque élément de l'indice 1 à (n-1), l'élément courant est comparé à ceux de la sous-liste déjà triée. En parcourant la sous-liste vers la gauche, on décale à droite les éléments plus grands. Lorsque l'on atteint un élément plus petit ou la fin de la sous-liste, l'élément courant est inséré à cet endroit.

C'est un algorithme simple, efficace pour les petites listes ou les listes presque triées, mais plus lent pour les grandes listes.

Le tableau suivant montre le sixième passage en détail. À ce stade de l'algorithme, une sous-liste triée de six éléments [4,5,10,43,57,91] existe déjà. Nous voulons maintenant insérer 45 dans cette sous-liste triée. La première comparaison, avec 91, provoque le décalage de 91 vers la droite. Puis 57 est également décalé. Lorsque l'on atteint l'élément 43, le processus de décalage s'arrête, et 45 est inséré dans la position libre. On obtient alors une sous-liste triée de sept éléments.

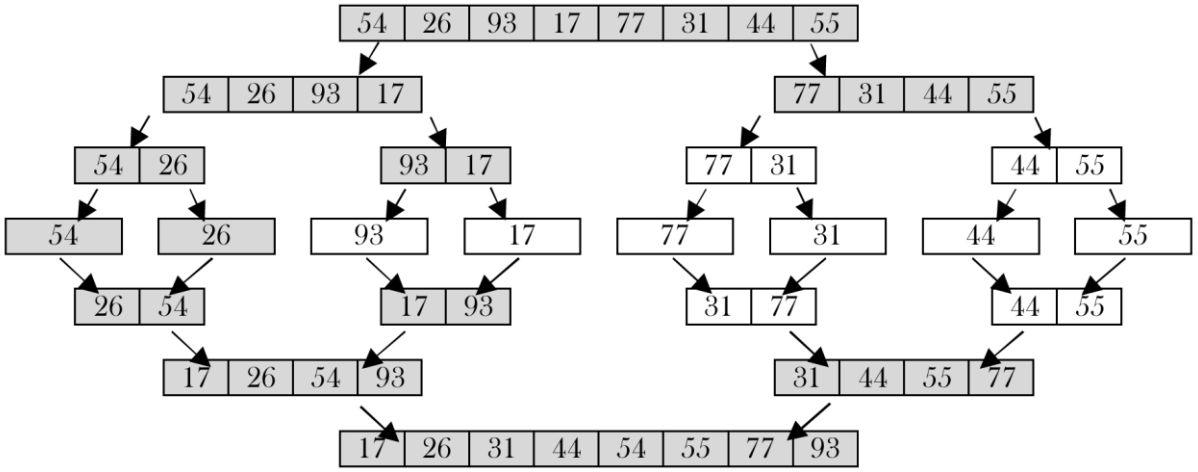
→									**Remarques : [4,5,10,43,57,91] existe déjà**
0	1	2	3	4	5	6	7	8	Conserver l'élément actuel A[6] dans une variable
4	5	10	43	57	91	**45**	9	7	Il faut insérer 45 dans la liste triée.
4	5	10	43	57	91	91	9	7	Copie de 91 dans A[6] car 91 > 45
4	5	10	43	57	57	91	9	7	Copie de 57 dans A[5] car 57 > 45
4	5	10	43	**45**	57	91	9	7	45 > 43 : insère 45 en A[4], la sous-liste est triée

I.4. Tri par fusion

Le tri par fusion (merge sort) est un exemple classique de la stratégie « diviser pour régner ». Il commence par diviser le tableau en deux moitiés égales, puis les combine ensuite de manière triée. C'est un algorithme récursif qui divise continuellement le tableau en deux. Si le tableau est vide ou ne contient qu'un seul élément, il est considéré comme trié par définition (c'est le cas de base). Si le tableau contient plus d'un élément, on le divise en deux parties et on applique récursivement le tri par fusion sur chacune. Une fois les deux moitiés triées, on effectue l'opération fondamentale appelée fusion, qui consiste à combiner deux sous-tableaux triés en un seul tableau trié.

C'est un algorithme efficace et stable, avec une complexité moyenne de O(n log n).

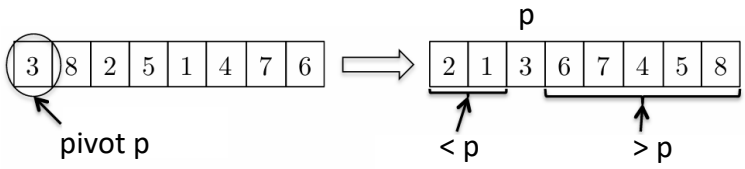
L'exemple ci-dessous montre les différentes étapes de cet algorithme :



I.5. Tri rapide

Le tri rapide (quick sort) est l'un des algorithmes de tri par comparaison les plus connus. Comme le tri par fusion, il utilise la stratégie du « diviser pour régner », et c'est donc un algorithme récursif. Cependant, la manière dont le tri rapide applique cette stratégie est un peu différente de celle du tri par fusion. Le tri rapide emploie cette approche afin de bénéficier des mêmes avantages que le tri par fusion, sans utiliser de mémoire supplémentaire. En revanche, la liste n'est pas toujours divisée en deux parties égales, ce qui peut dégrader les performances. Il trie les éléments en place, c'est-à-dire sans espace mémoire additionnel, ce qui le rend très efficace en général.

Le principe est de choisir un pivot puis de répartir les autres éléments en deux sous-listes : ceux inférieurs au pivot et ceux supérieurs au pivot. On applique ensuite récursivement le tri rapide sur chacune de ces sous-listes puis on combine les résultats : sous-liste triée de gauche + pivot + sous-liste triée de droite :



Le tri rapide est très efficace en pratique, mais peut être lent si le choix du pivot est défavorable. C'est son inconvénient principal et dans le pire des cas, ses performances sont comparables à la moyenne du tri à bulles, du tri par insertion ou du tri par sélection, c'est-à-dire une complexité de $O(n^2)$.

On peut toutefois sélectionner le pivot de manière aléatoire afin de réduire la probabilité d'obtenir le pire cas. Puisque l'élément pivot est choisi aléatoirement, on peut s'attendre à ce que la division du tableau d'entrée soit globalement bien équilibrée en moyenne. Cela permet d'éviter le pire des cas du tri rapide, qui se produit lorsque la partition est très déséquilibrée.

Même si la version randomisée du tri rapide améliore la probabilité d'éviter le pire cas, sa complexité dans le pire des cas reste $O(n^2)$.

Une façon d'améliorer les performances consiste à choisir le pivot plus soigneusement que par une simple sélection aléatoire d'un élément du tableau. Une méthode courante consiste à choisir le pivot comme la médiane d'un ensemble de trois éléments sélectionnés aléatoirement dans le tableau.

I.6. Tri par tas

I.6.1. Principe général

Le tri par tas (Heap Sort) est un algorithme de tri en place, basé sur la structure de données « tas binaire » (heap). Nous étudierons plus en détails le fonctionnement des tas par la suite mais voici une petite description qui explique pourquoi cette structure est importante, en particulier pour réaliser un tri dont la complexité temporelle est de O(nlogn), même dans le pire des cas.

I.6.2. Qu'est-ce qu'un tas ?

Un tas (heap) est une structure de données qui gère un ensemble évolutif d'objets associés à des clés, et qui peut identifier rapidement l'objet ayant la plus petite clé. Par exemple, les objets peuvent représenter des fiches d'employés, avec leurs numéros d'identification comme clés. Ou ils peuvent être les arêtes d'un graphe, avec des clés correspondant à leurs longueurs.

I.6.3. L'importance des opérations et de leur complexité

Les points les plus importants à retenir concernant toute structure de données sont les opérations qu'elle permet d'effectuer et le temps nécessaire pour chacune d'elles.

Par exemple, la complexité d'une opération d'insertion d'un élément dans une liste en Python (à une position i) est de O(n) car il faut décaler les éléments après la position i. Par contre, insérer un élément dans un dictionnaire est d'une complexité moyenne de O(1).

Nous verrons cette année que les dictionnaires utilisent une table de hachage, ce qui leur permet de réaliser cette opération en un temps constant (les tables de hachage sont introduites dans ce cours, mais nous étudierons plus en détails le concept de hachage au cours de l'année). Concernant la complexité d'une opération d'extraction d'un élément minimum, elle est de O(n) pour les listes et les dictionnaires, car dans les deux cas il faut parcourir tous les éléments.

Concernant les tas, les deux opérations principales qu'ils prennent en charge sont l'insertion d'un nouvel élément et l'extraction de l'élément ayant la plus petite clé. Et là où le tas est une structure très intéressante, c'est que la complexité pour rechercher un minium dans un tas est de O(1) et elle est de O(log n) pour l'extraire.

I.6.4. Pourquoi le tas rend le tri plus efficace

Le tri par tas est donc plus rapide que les méthodes simples (comme le tri à bulles ou par insertion) parce qu'il utilise la structure de données « tas » pour trouver et extraire efficacement le maximum ou le minimum à chaque étape.

L'algorithme du tri par tas consiste à insérer tous les éléments d'un tableau non trié dans un tas, puis à retirer les éléments un par un à partir de la racine du tas jusqu'à ce qu'il soit vide. La construction du tas à partir du tableau est de complexité O(n), et l'extraction successive des n éléments les plus petits est n·O(logn), soit O(nlogn).

I.7. Tri par comptage

Le tri par comptage (Counting Sort) n'est pas un algorithme de tri par comparaison et offre une complexité en O(n). Pour atteindre cette complexité, le tri par comptage suppose que chaque élément est un entier compris entre 1 et k, pour un certain entier k. Lorsque k=O(n), le tri par comptage s'exécute donc en temps linéaire O(n). Le principe de base du tri par comptage est de compter, pour chaque élément x, combien d'éléments sont inférieurs à x. Cette information permet ensuite de placer directement chaque élément à sa position correcte dans le tableau trié. Par exemple, pour trier [4,2,2,8,3], on compte les occurrences de chaque nombre, puis on réécrit : [2,2,3,4,8].

Cet algorithme est très rapide quand les valeurs sont des entiers dans une plage limitée mais il est inefficace si les valeurs possibles sont trop nombreuses (comme dans le cas de très grands entiers dispersés).

I.8. Tri par base

Comme le tri par comptage et d'autres algorithmes similaires, le tri par base repose sur certaines hypothèses concernant les données d'entrée. Supposons que les valeurs à trier soient des nombres exprimés dans une base d, c'est-à-dire que chaque nombre est codé avec d motifs.

Dans le tri par base, on commence par trier les éléments selon leur dernier chiffre (le chiffre le moins significatif). Les résultats obtenus sont ensuite triés à nouveau selon le chiffre suivant, et ainsi de suite, jusqu'à atteindre le chiffre le plus significatif. On utilise un tri stable pour trier selon le dernier chiffre, puis à nouveau selon l'avant-dernier, le troisième, etc…

Le temps total d'exécution est de O(n·d), ce qui est approximativement O(n) lorsque le nombre de motifs d est petit. Le tri utilisé à chaque étape doit être stable (comme le tri par comptage). Mais cet algorithme ne fonctionne efficacement que pour des entiers ou des chaînes de longueur fixe.

Par exemple, si on souhaite trier [170, 45, 75, 90, 802, 24, 2, 66] :

- Tri selon les unités : [170, 90, 802, 2, 24, 45, 75, 66]
- Tri selon les dizaines : [802, 2, 24, 45, 66, 170, 75, 90]
- Tri selon les centaines : [2, 24, 45, 66, 75, 90, 170, 802]

II) LES STRUCTURES DE DONNÉES

Les algorithmes de tri que nous avons étudiés illustrent l'importance de la manière dont les données sont organisées et manipulées en mémoire.

Dans un Système de Gestion de Base de Données (SGBD), cette organisation est encore plus cruciale : le système doit être capable de stocker, rechercher et retrouver rapidement des informations parmi d'importants volumes de données.

Pour atteindre cette efficacité, les SGBD s'appuient sur des structures de données spécialisées, telles que les arbres, les tables de hachage et d'autres mécanismes d'indexation, qui permettent d'accéder rapidement à l'information sans parcourir l'ensemble des enregistrements.

Les structures de données sont utilisées dans presque tous les grands logiciels. Une structure de données permet d'organiser l'information afin de pouvoir y accéder rapidement et efficacement. Vous en avez déjà rencontré plusieurs exemples :

- La liste (en Python), qui est une structure de données dynamique, implémentée sous la forme d'un tableau redimensionnable en mémoire.
- Le dictionnaire (en Python), qui est une structure de données associative (ou table de hachage). Il stocke des paires clé-valeur, ce qui permet d'accéder à une valeur directement via sa clé, sans parcourir tout le contenu.
- La file (queue), utilisée dans l'implémentation en temps linéaire de la recherche en largeur dans un graph (BFS), permet d'organiser les données de façon séquentielle, de sorte que l'ajout d'un élément à l'arrière ou le retrait d'un élément à l'avant se fasse en temps constant ;
- La pile (stack), essentielle dans l'implémentation itérative de la recherche en profondeur (DFS), permet de retirer ou d'ajouter un élément en tête également en temps constant.

Différentes structures de données prennent en charge différents ensembles d'opérations, ce qui les rend mieux adaptées à certains types de tâches de programmation. Par exemple, les algorithmes de recherche en largeur (BFS) et de recherche en profondeur (DFS) ont des besoins distincts, ce qui nécessite l'utilisation de deux structures de données différentes.

De manière générale, plus une structure de données prend en charge d'opérations différentes, plus ces opérations sont lentes et plus la surcharge en mémoire est importante.

Après avoir fait quelques rappels sur ces structures vues l'an dernier et avant d'aborder les celles utilisées dans les SGBD, nous commencerons par étudier les tas (heaps). Même s'ils ne sont pas directement utilisés dans les SGBD, les tas constituent une excellente introduction aux arbres binaires, sur lesquels reposent de nombreuses structures internes aux bases de données.

De plus, les tas ont une grande valeur algorithmique : ils permettent d'accélérer la recherche du minimum ou du maximum, et donc d'optimiser l'exécution d'algorithmes classiques, comme l'algorithme de Dijkstra pour la recherche du plus court chemin dans un graphe.

Le tableau ci-dessous récapitule les structures de données les plus populaires :

Structure	Complexité en temps (moyenne et pire des cas)				Complexité spatiale (pire des cas)	Remarques
	Accès	Recherche	Insertion	Suppression		
Tableaux (Arrays)	O(1) O(1)	O(n) O(n)	O(n) O(n)	O(n) O(n)	O(n)	Structure contiguë, très rapide pour l'accès direct, mais rigide (taille fixe).
Liste (List)	O(1) O(1)	O(n) O(n)	O(1)[1] O(n)	O(1)[1] O(n)	O(n)	Structure dynamique et flexible, mais les opérations internes sont souvent linéaires.
Piles (Stack)	O(1) O(1)	O(n) O(n)	O(1) O(1)	O(1) O(1)	O(n)	Accès LIFO : dernier entré, premier sorti. Utilisée pour les appels de fonctions ou le DFS.
Files (Queues)	O(1) O(1)	O(n) O(n)	O(1) O(1)	O(1) O(1)	O(n)	Accès FIFO : premier entré, premier sorti. Utilisée pour les ordonnancements et BFS.
Dictionnaires (Dict)	O(1) O(n)[2]	O(1) O(n)[2]	O(1) O(n)[2]	O(1) O(n)[2]	O(n)[3]	Basés sur une table de hachage : accès par clé en temps constant en moyenne.
Tables de hachage (Hash Tables)	O(1) O(n)[2]	O(1) O(n)[2]	O(1) O(n)[2]	O(1) O(n)[2]	O(n)	Permettent une recherche rapide sans tri, mais aucun ordre entre les données.
Tas (Heaps)	O(1) O(1)	O(n) O(n)	O(logn) O(logn)	O(logn) O(logn)	O(n)	Accès rapide au min ou max. Utilisé dans les files de priorité et le tri par tas.
Abres de recherche binaires (Binary Search Trees)	O(logn) O(n)	O(logn) O(n)	O(logn) O(n)	O(logn) O(n)	O(n)	Permettent une recherche ordonnée rapide, mais peuvent se déséquilibrer.
Arbres de recherche binaires équilibrés (Balanced Binary Search Trees)	O(logn) O(logn)	O(logn) O(logn)	O(logn) O(logn)	O(logn) O(logn)	O(logn)	Maintiennent un équilibre automatique, garantissant des performances stables.

Tableau 2: Caractéristiques des principales structures de données

O(1)[1] = complexité amortie (grâce à l'allocation dynamique)
O(n)[2] = complexité dégradée en cas de nombreuses collisions dans la table de hachage
O(n)[3] = Python redimensionne automatiquement sa table de hachage en allouant un espace en ~2n pour maintenir un bon facteur de charge et les opération en O(1)

Certaines de ces structures, comme les tableaux et les listes, servent de base à la représentation physique des enregistrements en mémoire ou sur disque.

D'autres, comme les piles et les files, sont utilisées pour gérer les opérations internes du SGBD, par exemple le traitement séquentiel des requêtes, la gestion des transactions, ou encore la planification des tâches.

Les tables de hachage et les dictionnaires jouent un rôle central dans la recherche rapide d'informations, notamment pour les index par hachage qui permettent d'accéder directement à une donnée à partir d'une clé.

Enfin, les arbres binaires de recherche et surtout leurs versions équilibrées constituent la base des structures d'indexation arborescentes (comme les B-trees et B+trees) utilisées dans la plupart des bases de données modernes pour retrouver rapidement des enregistrements triés ou accélérer les jointures (une jointure est une opération qui combine les données de plusieurs tables à partir d'une clé commune).

II.1. Les tableaux

Un tableau est une structure de données fondamentale permettant de stocker une collection d'éléments du même type. Les éléments sont placés dans des emplacements consécutifs en mémoire, ce qui permet d'y accéder rapidement grâce à leur indice.

En Python, les tableaux sont implémentés sous la forme de listes dynamiques, mais dans d'autres langages comme C ou Java, un tableau possède une taille fixe qui ne peut pas être modifiée après sa création.

L'accès à un élément d'un tableau se fait en temps constant O(1), car il suffit de connaître l'adresse de base et l'indice de l'élément à atteindre. Cependant, certaines opérations comme l'insertion ou la suppression d'un élément au milieu du tableau nécessitent de décaler les autres éléments, ce qui entraîne une complexité en O(n).

II.2. Les listes

Les listes permettent de surmonter la rigidité des tableaux et de pouvoir ajouter ou retirer des éléments plus facilement. Elles offrent une structure séquentielle plus flexible.

En Python, la liste est implémentée comme un tableau dynamique : les éléments sont stockés en mémoire de manière contiguë, comme un tableau classique, mais leur taille peut évoluer et elles permettent d'insérer, supprimer ou parcourir des éléments facilement.

Ce comportement implique une gestion intelligente de la mémoire, qu'on appelle redimensionnement amorti. L'idée est d'anticiper le besoin en mémoire et d'allouer plus d'espace mémoire que nécessaire lors de la création d'une liste. En cas d'ajout d'un élément, cela évite d'allouer un nouveau bloc mémoire, de copier tous les anciens éléments dans ce nouvel emplacement et d'y ajouter l'élément nouveau. Le coût en temps est alors en O(1), sauf lorsqu'il faut réallouer de l'espace mémoire (ce qui arrive de temps en temps).

L'accès à un élément par index dans une liste Python a un coût en O(1) car chaque élément de la liste possède une adresse mémoire spécifique et Python connait cette adresse.

Les opérations de recherche par valeur sont en revanche en moyenne de O(n) car Python doit parcourir la liste de manière séquentielle du début à la fin.

En fonction du mode d'accès souhaité aux éléments (dernier arrivé, premier sorti ou inversement), on spécialise la liste pour créer deux structures : la pile et la file.

II.3. Les piles et les files

La pile suit le principe LIFO (Last In, First Out) : le dernier élément ajouté est le premier retiré. Elle est utilisée dans la gestion de la mémoire, les appels de fonctions récursives et certains algorithmes de parcours (comme la recherche en profondeur dans les graphes - DFS).

Une pile d'assiettes dans une cafétéria en est un bon exemple : les assiettes sont empilées au fur et à mesure qu'elles sont nettoyées, et ajoutées par le dessus. Lorsqu'une assiette est nécessaire, elle est prise sur le dessus de la pile. Ainsi, la première assiette posée est la dernière à être utilisée.

La file suit le principe FIFO (First In, First Out) : le premier élément inséré est le premier traité. Elle est très utilisée dans les systèmes de gestion de tâches, les requêtes en attente ou les algorithmes de parcours en largeur (BFS).

De manière générale, une file représente une suite de personnes ou d'éléments attendant d'être traités dans un ordre séquentiel, en commençant par le premier élément arrivé.

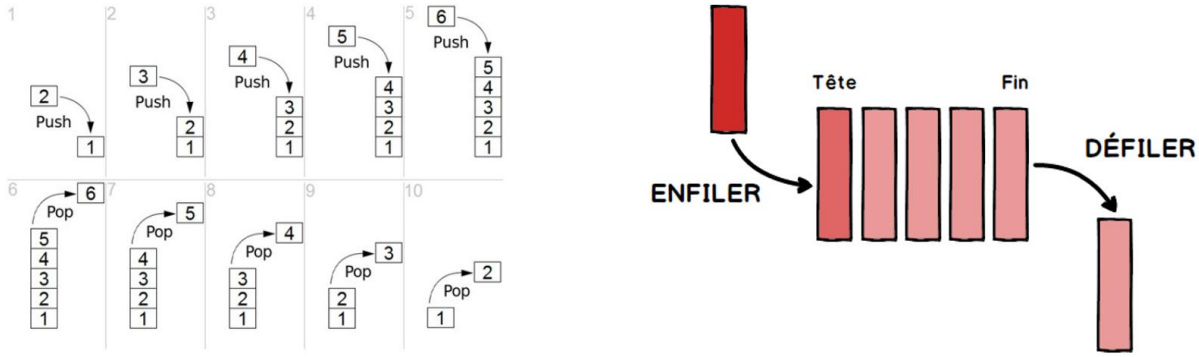


Figure 1 : Illustration du principe de fonctionnement d'une pile (à gauche) et d'une file (à droite)

En Python, le module `collections.deque` fournit une structure de file optimisée permettant d'ajouter ou de retirer des éléments aux deux extrémités en temps constant O(1).
Contrairement aux listes Python, où les opérations en début de liste sont coûteuses (O(n)), les deque sont conçues pour un accès rapide en tête et en queue, ce qui les rend idéales pour implémenter efficacement des piles (LIFO) ou des files (FIFO). Elles sont particulièrement utilisées dans les algorithmes de parcours (comme BFS) ou pour gérer des buffers circulaires.

II.4. Les tables de hachage

II.4.1. Définition et rôle dans les SGBD

Une table de hachage permet des opérations d'accès, de recherche, d'insertion et de suppression en O(1) en moyenne. C'est une structure essentielle dans les SGBD pour les index par hachage et la recherche rapide d'enregistrements.

II.4.2. Principe général

Le but d'une table de hachage est de gérer un ensemble d'objets évolutif associés à des clés, tout en permettant des recherches rapides par clé. Elle permet ainsi de vérifier facilement si un élément est présent ou non.

Supposons que l'on veuille gérer l'enregistrement du nombre de visites d'un site web par adresse IP. L'univers U dans cet exemple serait l'ensemble des 2^{32} adresses IPv4 possibles, les clés seraient les adresses IP et les objets à gérer seraient le nombre de visites. Une façon conceptuellement simple d'implémenter les opérations d'accès, d'insertion et de suppression rapides consisterait à stocker le nombre de visites dans une grande liste, avec une case pour chaque adresse IP possible de l'ensemble U.

Bien entendu, la taille de l'univers U est énorme et cela demanderait beaucoup trop d'espace mémoire. La solution serait donc que la liste gère un sous-ensemble $S \subseteq U$ de taille raisonnable plutôt que l'univers U dans son ensemble. Ainsi, dans notre exemple, l'ensemble S représenterait uniquement les adresses IP ayant visité une page web au cours des dernières 24 heures. Dans ce cas, l'espace mémoire utilisé serait uniquement proportionnel à $|S|$, c'est-à-dire au nombre réel d'éléments stockés. Dans la plupart des applications utilisant des tables de hachage, la taille de l'univers U est immense, tandis que celle du sous-ensemble S reste gérable.

Cependant, les temps d'exécution des opérations de recherche avec les listes augmentent linéairement avec $|S|$ car l'indice (ou l'adresse) de l'élément recherché n'est pas connu à l'avance. Python doit donc parcourir la liste élément par élément jusqu'à trouver la bonne valeur, ou arriver à la fin. La liste ne possède aucune information interne sur la position de chaque valeur.

Les tables de hachage apportent une solution à ce problème : au lieu de parcourir toutes les valeurs, elles utilisent une clé pour calculer directement la position où se trouve la donnée. Ce calcul passe par une fonction de hachage, qui transforme la clé en un indice numérique (ou adresse).

II.4.3. Le rôle de la fonction de hachage

Une fonction de hachage effectue la traduction entre ce qui nous intéresse réellement, comme les noms de nos amis, les adresses IP ayant visité un site web au cours des dernières 24 heures, etc. et les positions dans la table de hachage. De manière formelle, une fonction de hachage est une fonction qui associe chaque clé possible (appartenant à l'ensemble U) à une position dans le tableau. Dans une table de hachage, les positions sont généralement

numérotées à partir de 0, ce qui fait que l'ensemble des positions du tableau est : {0, 1, 2, …, n−1}.

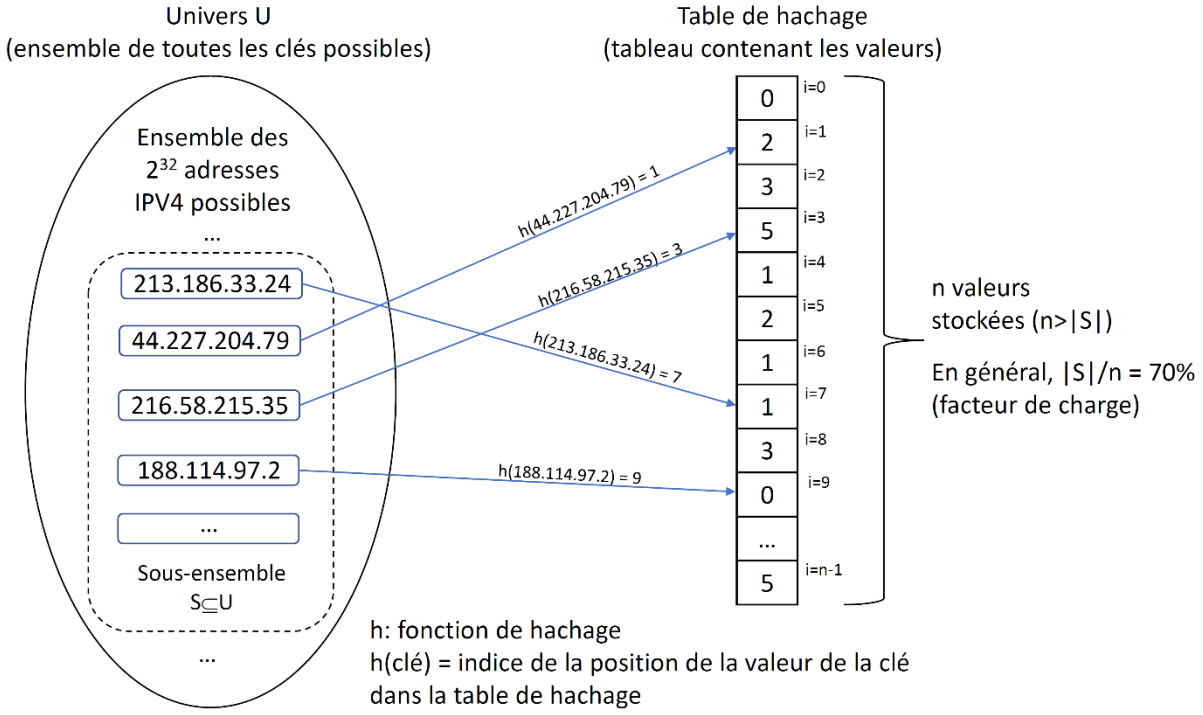


Figure 2: Principe de fonctionnement d'une table de hachage

Une fonction de hachage indique où commencer la recherche d'un objet. Dans notre exemple, si on choisit une fonction de hachage h telle que h(213.186.33.24) = 7, alors la position 7 du tableau est l'endroit où il faut commencer à chercher le nombre de visites effectuées. De même, la position 7 sera le premier emplacement à essayer pour insérer le nouveau nombre de visites de cette IP dans la table de hachage.

II.4.4. Le problème des collisions

Que se passe-t-il si deux clés différentes (par exemple 213.186.33.24 et 66.16.10.23) sont hachées vers la même position (par exemple 7) ? Si vous cherchez le nombre de visites de l'IP 66.16.10.23 mais que vous trouvez le nombre de visites de l'IP 213.186.33.24 dans la table de hachage à l'indice i = 7, comment savoir si le nombre de visites de l'IP 66.16.10.23 se trouve également dans la table de hachage ? Et si vous essayez d'insérer le nombre de visites de l'IP 66.16.10.23 à la position i = 7 mais que celle-ci est déjà occupée, où devez-vous le placer ?

Lorsqu'une fonction de hachage h associe deux clés différentes k_1 et k_2 à la même position, autrement dit, quand $h(k_1) = h(k_2)$, on appelle cela une collision. Les collisions créent de la confusion quant à l'emplacement réel d'un objet dans la table de hachage, et il est donc souhaitable de les réduire autant que possible. Malheureusement, elles sont inévitables. En effet, chaque fois que le nombre de positions disponibles dans le tableau est inférieur à la taille de l'univers U, toute fonction de hachage, aussi ingénieuse soit-elle, provoquera au moins une collision.

II.4.5. Réduction et gestion des collisions

Pour limiter la fréquence de ces collisions, il est essentiel de choisir une bonne fonction de hachage, capable de répartir uniformément les clés dans le tableau, de façon que chaque position ait la même probabilité d'être utilisée. Une fonction de hachage mal conçue peut en effet provoquer des regroupements d'éléments, ralentissant fortement les opérations de recherche et d'insertion.

Il faut également mettre en place une stratégie de gestion pour éviter la perte d'information en cas de collision. Les deux approches les plus courantes sont le chaînage, où plusieurs éléments partageant la même position sont stockés ensemble dans une petite sous-liste, et l'adressage ouvert, où l'on cherche une autre case libre selon une règle déterminée.

II.4.6. Le facteur de charge

Un autre élément important est le facteur de charge d'une table de hachage. C'est le rapport entre le nombre d'éléments stockés et le nombre total de cases disponibles. Il mesure donc le taux de remplissage de la table et permet de décider quand redimensionner celle-ci pour préserver de bonnes performances et limiter les collisions.

II.4.7. Conclusion

Les tables de hachage offrent un équilibre idéal entre rapidité et flexibilité. Nous étudierons tout cela en détail au cours de l'année dans un chapitre qui sera intégralement consacré aux tables de hachage ainsi qu'à la mise en œuvre et à l'utilisation des fonctions de hachage. Nous verrons les caractéristiques nécessaires pour avoir de bonnes fonctions de hachage sur différents exemples et celles qui sont les plus utilisées.

II.5. Les dictionnaires

II.5.1. Définitions et rôle

Les structures telles que les tableaux, les listes, les files et les piles gèrent des éléments séquentiellement. Les dictionnaires organisent les données sous forme de paires clé-valeur, permettant un accès direct aux informations sans parcours complet.

II.5.2. Implémentation en Python

En Python, les dictionnaires sont implémentés à l'aide de tables de hachage.

Comme nous l'avons vu, les tables de hachage sont des structures de données qui permettent d'associer une clé à une valeur et d'y accéder très rapidement, sans avoir à parcourir toute la collection. Elles reposent sur une fonction de hachage, qui transforme chaque clé (par exemple une chaîne de caractères) en un indice numérique correspondant à une position dans un tableau. Ainsi, au lieu de rechercher une donnée séquentiellement, comme dans une liste, où il faut comparer chaque élément un à un, le dictionnaire calcule directement l'emplacement mémoire où se trouve la valeur associée à la clé. Cette approche réduit la recherche, l'insertion et la suppression à une complexité moyenne de O(1), quelle que soit la taille du dictionnaire.

II.5.3. Fonctionnement interne

Un dictionnaire fonctionne exactement comme une table de hachage : chaque clé est transformée en un indice par une fonction de hachage, cet indice correspond à une position dans un tableau, c'est à cet emplacement que Python stocke le couple clé-valeur, et toutes les opérations (insertion, recherche, suppression) se font en temps moyen O(1).

Prenons par exemple le dictionnaire `annuaire` :

```
annuaire = {
    "google.fr": "172.217.20.163",
    "lyceefrancois1.net": "213.186.33.24",
    "allocine.fr": "172.64.149.28"
}
```

Ici, les clés sont des chaînes de caractères ("google.fr", "lyceefrancois1.net", "allocine.fr") et les valeurs sont les adresses IP correspondantes.

Les informations liées à la table de hachage utilisée par le dictionnaire sont les suivantes :

Clé	**h(clé)**	**Indice dans le tableau des valeurs**	**Valeur stockée**
"google.fr"	hash("google.fr") = 42	Case 42	"172.217.20.163"
"lyceefrancois1.net"	hash("lyceefrancois1.net") = 17	Case 17	"213.186.33.24"
"allocine.fr"	hash("allocine.fr") = 88	Case 88	"172.64.149.28"

Lorsqu'on exécute annuaire["google.fr"], Python calcule hash("google.fr"), trouve que cela correspond à la case 42, et retourne la valeur associée : "172.217.20.163".

Si une autre clé produit le même indice (collision), Python applique une stratégie d'adressage ouvert pour trouver la prochaine case libre.

II.5.4. Fonctionnalités supplémentaires des dictionnaires

Les dictionnaires ne se contentent pas d'être une simple table de hachage. Ils ajoutent plusieurs fonctionnalités de haut niveau très utiles tels que la gestion automatique des collisions, un redimensionnement dynamique afin de maintenir les performances en O(1), une interface de haut niveau permettant un accès simple et lisible via des clés arbitraires comme les `str`, `int`, `tuple`, etc. ainsi que des ajustements internes apportant sécurité et robustesse.

II.5.5. Avantages et performances

Grâce à cette organisation, les dictionnaires Python offrent des performances remarquables : accéder à une valeur via sa clé se fait presque instantanément, contrairement à la recherche séquentielle dans une liste, qui nécessite un temps proportionnel au nombre d'éléments. Cela fait des dictionnaires l'une des structures les plus efficaces et les plus utilisées pour le stockage et la recherche rapide de données.

II.6. Les tas

II.6.1. Limites des tables de hachage et des dictionnaires

Les tables de hachage et les dictionnaires apportent un énorme gain de vitesse d'accès mais les éléments stockés ne sont pas organisés de manière logique. Ils sont stockés à des positions calculées par la fonction de hachage et non selon une relation (comme « plus petit que » ou « parent de »). Il n'existe donc aucune hiérarchie entre les éléments. De manière imagée, une table de hachage est plate, et non en arborescence.

Les tables de hachage et les dictionnaires n'imposent pas non plus d'ordre de priorité entre les clés. Ils permettent d'accéder rapidement à un élément via sa clé, mais pas de trier ou de parcourir les clés selon un ordre particulier (alphabétique, numérique, etc.)

II.6.2. Données hiérarchisées et priorisées dans les SGBD

Dans le domaine des SGBD, traiter efficacement des données hiérarchisées ou priorisées est essentiel, car de nombreux mécanismes internes d'un SGBD reposent sur ces notions.

En effet, les SGBD doivent organiser les données de manière logique pour faciliter la recherche rapide d'enregistrements, la navigation entre des éléments liés et l'exécution efficace des requêtes. Les arbres (comme les B-trees ou B+trees) dont nous parlerons par la suite sont largement utilisés pour cela.

D'autre part, certains mécanismes d'un SGBD reposent sur la priorisation des opérations, par exemple la gestion des requêtes concurrentes, la planification des tâches d'accès au disque, la récupération après panne (ordre des transactions à rejouer). Ces cas utilisent souvent des files de priorité ou des tas, afin de permettre de choisir rapidement la transaction ou l'opération la plus urgente ou la plus importante, et d'assurer un traitement équitable et efficace.

Dans un SGBD, traiter efficacement des données hiérarchisées permet d'accélérer la recherche et l'indexation, tandis que gérer des données priorisées permet d'optimiser l'exécution et la planification des opérations internes.

II.6.3. Le rôle des tas

Si on veut gérer des priorités pour traiter efficacement des données hiérarchisées ou priorisées, il faut une autre organisation que les tables de hachage ou les dictionnaires : les tas, basés sur la structure des arbres binaires.

Un tas est une structure de données qui permet de gérer un ensemble d'objets évolutif associés à des clés, et de retrouver rapidement l'objet ayant la plus petite clé. Par exemple, les objets peuvent représenter des fiches d'employés, dont les clés correspondent à leurs numéros d'identification ; ils peuvent aussi représenter les arêtes d'un graphe, avec des clés correspondant à leur longueur ; ou encore des événements planifiés dans le futur, chaque clé indiquant le moment où l'événement doit se produire.

II.6.4. Les opérations fondamentales des tas

Les deux opérations principales prises en charge par les tas sont l'insertion d'un objet et l'extraction du minimum (en le supprimant du tas). Par exemple, si l'on appelle l'opération d'insertion quatre fois pour ajouter des objets ayant pour clés 12, 7, 29 et 15 dans un tas initialement vide, alors l'opération d'extraction du minimum renverra l'objet dont la clé est 7 et la supprimera du tas. Les clés n'ont pas besoin d'être uniques : s'il existe plusieurs objets ayant la plus petite clé, l'opération d'extraction du minimum renverra l'un d'eux de manière arbitraire.

II.6.5. Pourquoi utiliser un tas et non une liste ?

Il serait facile de ne prendre en charge que l'opération d'insertion, en ajoutant simplement les nouveaux objets à la fin d'une liste (ce qui se fait en temps constant). Le problème, c'est que l'opération d'extraction du minimum nécessiterait alors une recherche exhaustive linéaire parmi tous les objets. De même, on pourrait ne prendre en charge que l'opération d'extraction du minimum : il suffirait de trier à l'avance l'ensemble initial de n objets selon leurs clés, puis d'extraire successivement les objets depuis le début de la liste triée, chaque extraction se faisant alors en temps constant. Mais dans ce cas, toute insertion ultérieure d'un nouvel élément exigerait un temps linéaire, puisqu'il faudrait réinsérer l'objet à la bonne position pour maintenir l'ordre.

Le véritable défi consiste donc à concevoir une structure de données qui permette d'effectuer ces deux opérations très rapidement. C'est précisément la raison d'être des tas.

II.6.6. Complexité et variantes

Dans un tas contenant n objets, les opérations d'insertion et d'extraction du minimum s'exécutent en O(log n). En prime, dans la plupart des implémentations, la constante cachée derrière la notation grand O est très faible, et la structure ne nécessite presque pas de mémoire supplémentaire. Il existe également une variante du tas qui prend en charge les opérations d'insertion et d'extraction du maximum en O(log n), où n est le nombre d'objets. Cependant, aucune implémentation de tas ne permet de gérer simultanément l'extraction du minimum et du maximum en O(log n). Il faut donc choisir l'une des deux priorités.

II.6.7. Opérations supplémentaires

Les tas peuvent également prendre en charge un certain nombre d'opérations supplémentaires, comme la recherche du minimum, l'insertion simultanée de plusieurs objets, et la suppression d'un objet.

On pourrait simuler une opération de recherche du minimum en appelant l'opération d'extraction du minimum, puis en réinsérant l'élément obtenu à l'aide de l'insertion (ce qui prendrait un temps O(log n)). Cependant, une implémentation typique d'un tas peut éviter cette approche détournée et prendre en charge l'opération de recherche du minimum directement en O(1).

De même, on pourrait construire un tas en insérant successivement n objets un par un dans un tas vide (pour un temps total O(n log n)), mais il existe une méthode plus astucieuse permettant d'ajouter n objets d'un coup en temps total O(n).

Enfin, les tas peuvent également supprimer des objets arbitraires, pas seulement celui ayant la plus petite clé, en O(log n).

Dans une application qui nécessite de calculer rapidement le minimum (ou le maximum) au sein d'un ensemble d'objets qui évolue dynamiquement, le tas est généralement la structure de données la plus adaptée.

II.6.8. Application : l'algorithme de Dijkstra

Vous avez vu l'année dernière que l'implémentation directe de l'algorithme de Dijkstra nécessite un temps d'exécution de O(mn), où m est le nombre d'arêtes et n le nombre de sommets. Cette complexité est suffisante pour traiter des graphes de taille moyenne (comportant quelques milliers de sommets et d'arêtes), mais elle devient trop lente pour les graphes très grands (avec des millions de sommets et d'arêtes). L'utilisation des tas permet une implémentation extrêmement rapide, quasiment linéaire en temps – en O((m+n)log n) – , de l'algorithme de Dijkstra.

Si cela vous intéresse, vous pouvez regarder le cours « Algorithme de Dijkstra avec les tas » et le TD associé qui expliquent comment implanter cet algorithme en utilisant les tas.

II.6.9. Implantation des tas

Prenons maintenant le temps d'étudier comment sont implantés les tas, ce qui va nous permettre d'introduire les structures arborescentes qui sont à la base des arbres binaires que nous verrons par la suite. Nous allons nous concentrer sur les deux opérations fondamentales : l'insertion et l'extraction du minimum, et sur la manière de garantir que toutes deux s'exécutent en temps logarithmique.

Il existe deux manières de représenter les objets d'un tas : sous forme d'arbre, ce qui est plus pratique pour les illustrations et les explications, ou sous forme de tableau, ce qui est plus efficace pour l'implémentation.

II.6.10. Les tas vus comme des arbres

Un tas peut être vu comme un arbre binaire enraciné, où chaque nœud possède 0, 1 ou 2 enfants, et où chaque niveau est rempli autant que possible.

Selon le nombre d'objets stockés, tous les niveaux de l'arbre peuvent être entièrement remplis ou non. Le dernier niveau peut être le seul partiellement rempli, et il est rempli de gauche à droite.

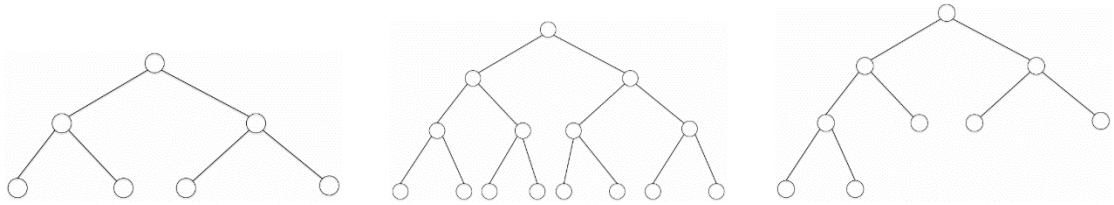


Figure 3 : Exemples d'arbres binaires avec 7, 15 et 9 nœuds.

Un tas gère des objets associés à des clés, de manière à respecter la propriété fondamentale du tas : pour chaque paire parent-enfant, la clé du parent est inférieure ou égale à celle de l'enfant. Les clés dupliquées sont autorisées. Il existe plusieurs façons d'organiser les objets tout en respectant cette propriété du tas.

Voici deux exemples valides construits sur le même ensemble de clés :

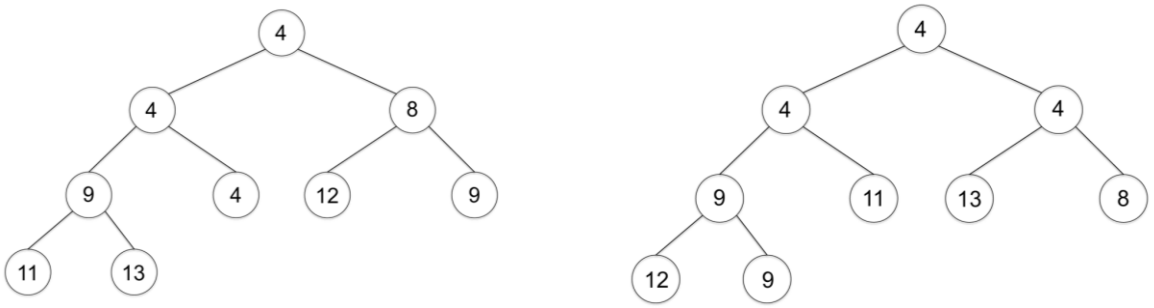


Figure 4 : Exemples de tas valides contenant neuf objets

Les deux tas ont un « 4 » à leur racine, qui est également la plus petite clé de toutes. Ce n'est pas un hasard : dans un tas, les clés ne peuvent que diminuer lorsqu'on remonte vers la racine. Ainsi, la clé du nœud racine est toujours la plus petite.

C'est une propriété essentielle puisque la raison d'être d'un tas est justement de permettre des calculs rapides du minimum.

II.6.11. Les tas vus comme des tableaux

Dans notre esprit, nous pouvons visualiser un tas sous forme d'arbre, mais dans une implémentation réelle, on l'encode sous forme de tableau, dont la taille correspond au nombre maximal d'objets que l'on prévoit de stocker. Le premier élément du tableau représente la racine de l'arbre, les deux éléments suivants correspondent au niveau suivant de l'arbre (dans le même ordre), et ainsi de suite pour les niveaux inférieurs.

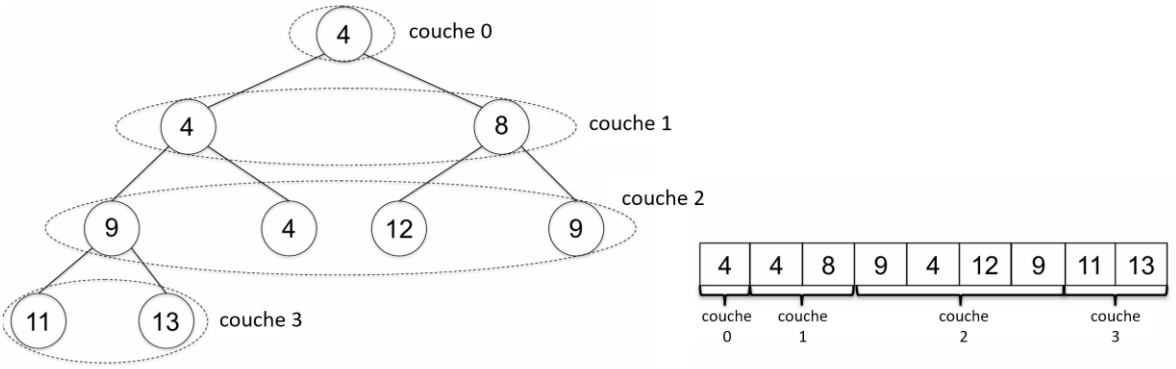


Figure 5 : Représentation en arbre (à gauche) et en tableau (à droite) d'un tas

Les relations parent-enfant dans l'arbre se traduisent très simplement dans le tableau. Si les positions du tableau sont numérotées 1, 2, …, n (où n est le nombre d'objets), alors les enfants de l'objet en position i se trouvent en positions 2i et 2i + 1 (s'ils existent) ; et inversement, pour un objet non racine en position $i \geq 2$, son parent se trouve à la position $\lfloor i/2 \rfloor$.

Soit un objet en position i dans le tableau :	
Position du parent	$\lfloor i/2 \rfloor$ (si $i \geq 2$)
Position de l'enfant à gauche	2i (si $2i \leq n$)
Position de l'enfant à droite	2i + 1 (si $2i + 1 \leq n$)

Tableau 3 : Relations parent-enfant des objets dans un tableau

Dans l'exemple donné ci-dessus :

- La racine (position 1) a pour enfants les objets en position 2 et 3,
- L'objet en position 3 (de valeur 8) a pour enfants ceux en position 6 et 7,
- Le parent du dernier objet (en position 9) est celui de la position $\lfloor 9/2 \rfloor = 4$.

Ces formules simples permettant de passer d'un enfant à son parent (et inversement) sont possibles parce que les tas reposent sur des arbres binaires complets. Ainsi, il n'est pas nécessaire de stocker explicitement la structure de l'arbre, ce qui fait du tas une structure de données à très faible coût en mémoire.

II.6.12. Implantation de l'opération d'insertion

Le défi consiste à conserver l'arbre complet tout en respectant la propriété du tas après chaque opération. Pour les deux opérations d'insertion et d'extraction du minimum, nous suivrons le même principe général :

1. Maintenir l'arbre complet de la manière la plus simple possible.
2. Corriger systématiquement les violations de la propriété du tas.

Après l'ajout de x au tas H, celui-ci doit toujours correspondre à un arbre binaire complet (avec un nœud de plus qu'auparavant) et continuer à respecter la propriété du tas. L'opération doit s'exécuter en O(log n), où n est le nombre d'objets contenus dans le tas. Prenons l'exemple de référence ci-dessous :

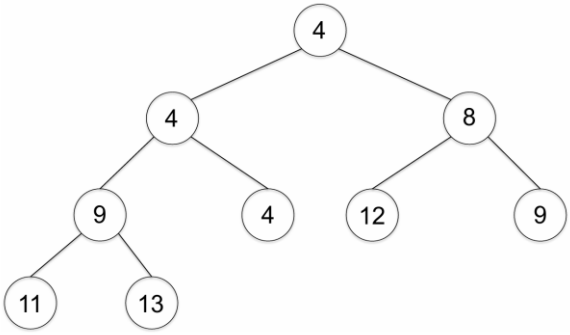


Figure 6 : Exemple de structure du tas originale avant l'opération d'insertion

Lorsqu'un nouvel objet est inséré, la manière la plus simple de garder l'arbre complet consiste à ajouter cet objet à la fin du tableau, ce qui revient à l'ajouter sur le dernier niveau de l'arbre (si ce dernier niveau est déjà plein, l'objet devient simplement le premier élément d'un nouveau niveau). Tant que l'implémentation garde en mémoire le nombre n d'objets, ce qui est facile à faire, cette étape s'effectue en temps constant O(1). Par exemple, si l'on insère un objet de clé 7 dans notre exemple précédent, on obtient :

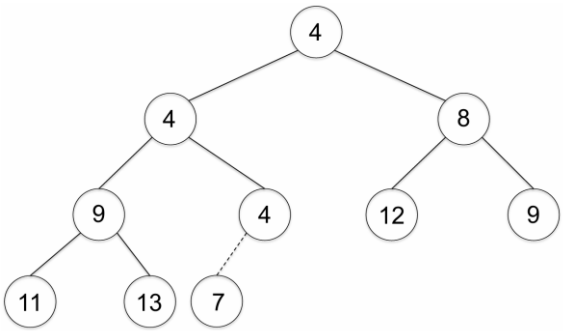


Figure 7 : Structure de l'arbre après insertion de la clé 7 : le tas reste valide

Nous avons maintenant un arbre binaire complet, et il n'y a qu'un seul endroit où la propriété du tas pourrait être violée : dans la nouvelle relation parent-enfant (entre le 4 et le 7). Dans l'exemple précédent, nous avons de la chance : cette nouvelle paire ne viole pas la propriété du tas. Et si nous insérons ensuite un objet de clé 10, nous avons encore une fois de la chance car le tas obtenu est immédiatement valide.

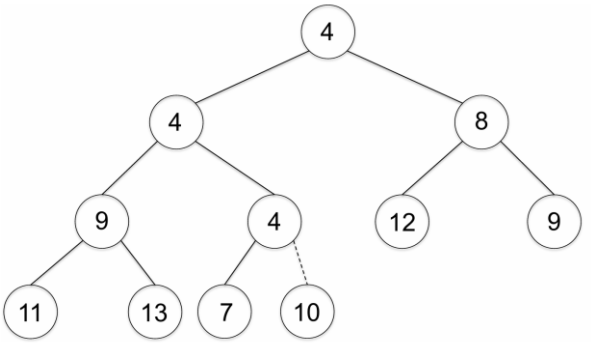


Figure 8 : Structure de l'arbre après insertion de la clé 10 : le tas reste encore valide

Mais supposons que nous insérions maintenant un objet ayant pour clé 5. Après l'avoir ajouté à la fin du tableau (et donc sur le dernier niveau de l'arbre), notre structure devient la suivante :

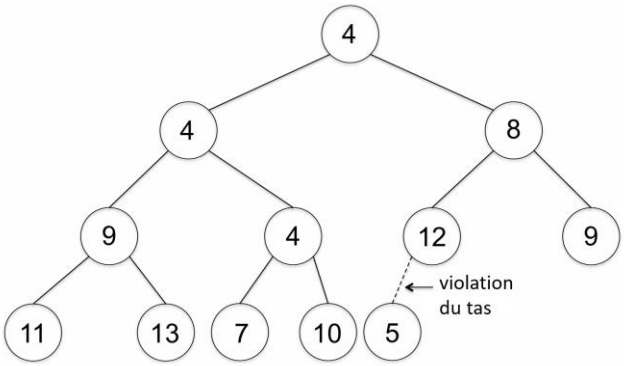


Figure 9: Structure de l'arbre après insertion de la clé 5 : le tas devient invalide

Nous avons maintenant un problème : la nouvelle paire parent-enfant 12-5 viole la propriété du tas.
Nous pouvons alors corriger localement cette violation en échangeant les deux nœuds concernés, mais le tas reste alors encore invalide au niveau de la paire parent-enfant 8-5 :

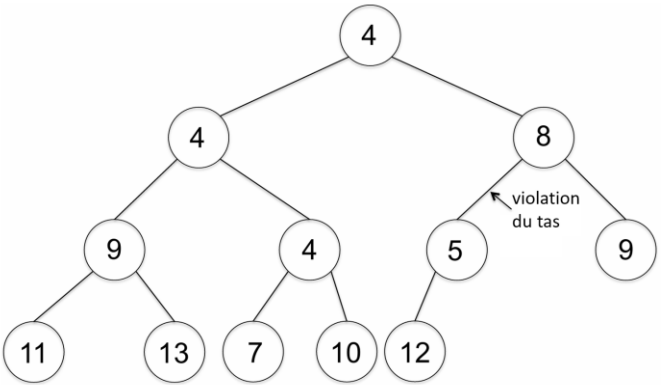


Figure 10: Après correction locale du tas, il reste invalide

Nous répétons donc l'opération et échangeons à nouveau les nœuds de la paire 8-5 qui viole la propriété du tas, ce qui nous permet d'obtenir finalement un tas valide. La propriété du tas étant désormais rétablie, l'opération d'insertion est terminée :

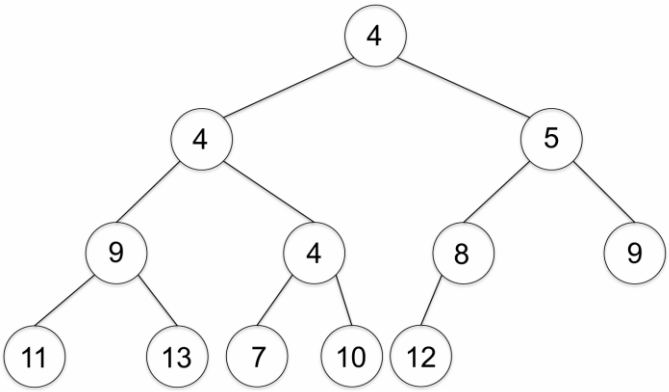


Figure 11: Après plusieurs corrections, le tas est maintenant valide

De manière générale, l'opération d'insertion consiste à ajouter le nouvel objet à la fin du tas, puis à échanger de manière répétée les nœuds de la paire parent-enfant qui viole la propriété du tas. À tout moment, il ne peut y avoir qu'une seule paire fautive : celle où le nouvel objet est l'enfant. Chaque échange fait remonter cette paire d'un niveau dans l'arbre. Ce processus ne peut pas durer indéfiniment : si le nouvel objet atteint la racine, il n'a plus de parent, et il ne peut donc plus exister aucune violation de la propriété du tas.
Puisqu'un tas est un arbre binaire complet, il comporte environ $\log_2(n)$ niveaux, où n est le nombre d'objets qu'il contient. Le nombre maximal d'échanges effectués lors d'une insertion est donc égal au nombre de niveaux, et chaque échange ne demande qu'une quantité constante de travail. On en conclut que le temps d'exécution dans le pire des cas de l'opération d'insertion est bien de O(log n).

II.6.13. Implantation de l'opération d'extraction du minimum

Rappelons que l'opération d'extraction du minimum consiste à supprimer et renvoyer l'objet ayant la plus petite clé d'un tas.
La racine du tas est toujours cet objet minimal et le véritable défi consiste ensuite à rétablir les propriétés du tas : l'arbre doit rester binaire complet, et la propriété du tas doit de nouveau être respectée après la suppression.
Pour cela, on conserve l'arbre complet de la manière la plus simple possible. Comme pour l'insertion, mais en sens inverse, on sait que le dernier nœud du tas doit être déplacé ailleurs. Puisque nous extrayons la racine, on remplace l'ancien nœud racine par le dernier nœud du tas.

Par exemple, en partant du tas suivant :

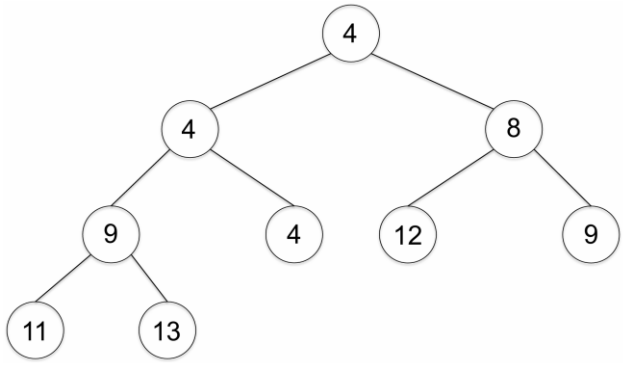


Figure 12: Exemple de structure du tas originale avant l'opération d'extraction du minimum

On obtiendrait le résultat suivant :

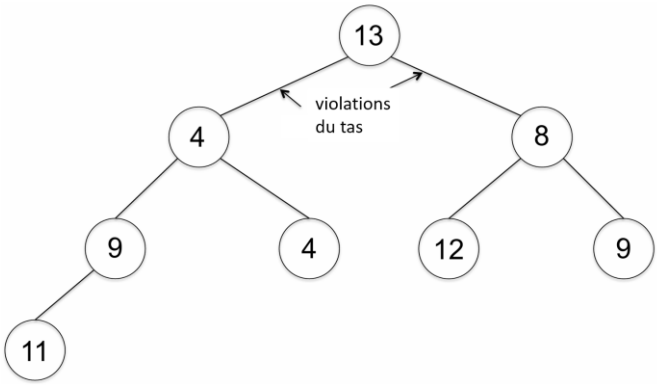


Figure 13: Après extraction du minimum, la structure du tas est devenue invalide

La bonne nouvelle, c'est que la propriété d'arbre binaire complet est rétablie. La mauvaise nouvelle, c'est que la promotion soudaine de l'objet ayant la clé 13 a créé deux violations de la propriété du tas : entre la paire 13-4, et entre 13-8.

Pour résoudre ce problème, il suffit de remplacer le nœud racine par le plus petit de ses deux enfants (échanger 4 et 13) :

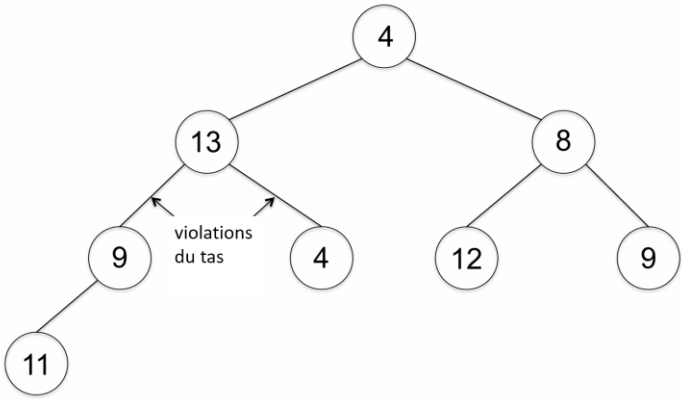


Figure 14 : Après remplacement du nœud racine par le plus petit de ses deux enfants, le tas reste invalide

Il n'y a désormais plus aucune violation de la propriété du tas au niveau de la racine : le nouveau nœud racine est plus petit à la fois que le nœud qu'il a remplacé (c'est précisément la raison pour laquelle nous l'avons échangé) et que son autre enfant (puisque nous avons échangé avec le plus petit des deux). Les violations du tas se sont simplement déplacées vers le bas, impliquant maintenant l'objet ayant la clé 13 et ses nouveaux enfants. Nous répétons donc l'opération : on échange le 13 avec son enfant le plus petit (4) :

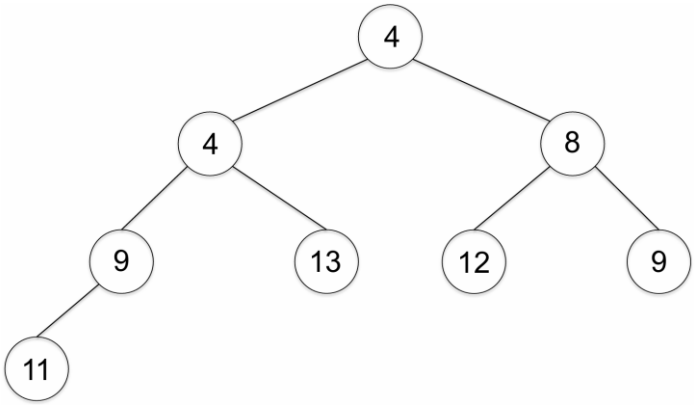


Figure 15: La structure du tas est maintenant valide

La propriété du tas est enfin rétablie, et l'extraction est donc terminée.

De manière générale, l'opération d'extraction du minimum consiste à déplacer le dernier objet du tas vers la racine (en écrasant l'ancienne), puis à échanger cet objet à plusieurs reprises avec son enfant le plus petit. À tout moment, il ne peut y avoir au maximum que deux paires parent-enfant fautives : celles où l'ancien dernier objet est devenu le parent. Chaque échange fait descendre cet objet d'un niveau dans l'arbre. Ce processus ne peut pas durer indéfiniment, il s'arrête dès que le nouvel objet atteint le dernier niveau, ou plus tôt si la propriété du tas est déjà respectée.

Le nombre d'échanges effectués est au plus égal au nombre de niveaux de l'arbre, et chaque échange ne demande qu'une quantité constante de travail. Comme un tas comporte environ $\log_2(n)$ niveaux, on en déduit que le temps d'exécution dans le pire des cas de l'opération d'extraction du minimum est de O(log n), où n représente le nombre d'objets contenus dans le tas.

II.7. Les arbres de recherche binaires

Le tas est une première approche des structures arborescentes. Pour mieux organiser et rechercher les données, explorons maintenant la structure de l'arbre de recherche binaire.

Un arbre de recherche, comme un tas, est une structure de données permettant de stocker un ensemble évolutif d'objets associés à des clés (et éventuellement à d'autres informations). Il maintient un ordre total entre les objets stockés et peut prendre en charge un ensemble d'opérations plus riche qu'un tas, mais cela se fait au prix d'une utilisation mémoire plus importante et, pour certaines opérations, de temps d'exécution un peu plus longs.

II.7.1. Opérations supportées

Un arbre de recherche permet de réaliser l'ensemble des opérations suivantes :

- Recherche : pour une clé k, renvoie un pointeur vers l'objet de la structure ayant cette clé, ou indique qu'aucun objet correspondant n'existe.
- Min / Max : renvoie un pointeur vers l'objet ayant la plus petite (respectivement, la plus grande) clé.
- Prédécesseur / Successeur : étant donné un pointeur vers un objet de la structure, renvoie un pointeur vers l'objet ayant la clé immédiatement inférieure (respectivement, immédiatement supérieure). Si l'objet donné a déjà la clé minimale (ou maximale), la structure renvoie « aucun ».
- Sortie triée : affiche ou renvoie les objets de la structure un par un, dans l'ordre croissant de leurs clés.
- Sélection : étant donné un nombre i compris entre 1 et le nombre total d'objets, renvoie un pointeur vers l'objet ayant la i^e plus petite clé.
- Rang : étant donnée une clé k, renvoie le nombre d'objets dans la structure dont la clé est inférieure ou égale à k.
- Insertion : étant donné un nouvel objet x, ajouter x à la structure de données.
- Suppression : pour une clé k, supprimer de la structure de données un objet de clé k, s'il existe.

Les arbres de recherche permettent d'effectuer l'ensemble de ces opérations en O(log n), y compris les insertions et les suppressions (sauf l'opération de sortie triée qui est réalisée en O(n)).

Un point important à souligner : les temps d'exécution précédents sont obtenus grâce à un arbre de recherche équilibré, qui est une version plus sophistiquée de l'arbre binaire de recherche standard décrit dans ce paragraphe. Ces performances ne sont pas garanties dans le cas d'un arbre de recherche non équilibré.

II.7.2. Quand utiliser un arbre de recherche équilibré ?

Dans une application qui doit maintenir une représentation ordonnée d'un ensemble d'objets qui évolue dynamiquement, un arbre de recherche équilibré (ou une structure de données dérivée de ce type) est généralement le meilleur choix. Cependant, si vous devez simplement maintenir une représentation ordonnée d'un ensemble statique (sans insertions ni suppressions), il est préférable de choisir un tableau trié plutôt qu'un arbre de recherche équilibré car ce dernier serait disproportionné.

Si votre ensemble de données est dynamique, mais que vous avez seulement besoin d'effectuer des opérations rapides de minimum (ou de maximum), préférez un tas à un arbre de recherche équilibré. Ces structures de données plus simples offrent moins de fonctionnalités qu'un arbre de recherche équilibré, mais ce qu'elles font, elles le font mieux : plus rapidement (d'un facteur constant ou logarithmique) et avec moins d'espace mémoire (d'un facteur constant). Les tables de hachage offrent encore moins de fonctionnalités, mais ce qu'elles font, elles le font encore mieux (en temps constant), dans la plupart des cas pratiques.

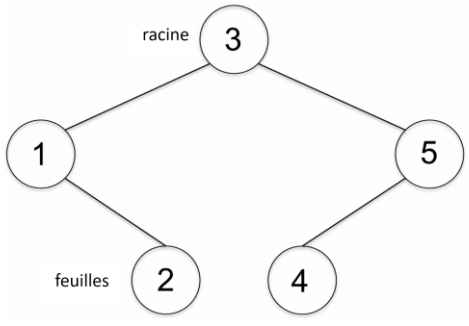
II.7.3. Propriétés des arbres de recherche

Dans un arbre binaire de recherche, chaque nœud correspond à un objet (associé à une clé) et possède trois pointeurs : un pointeur vers son parent, un pointeur vers son enfant gauche et un pointeur vers son enfant droit. Chacun de ces pointeurs peut être nul, indiquant l'absence de parent ou d'enfant. Le sous-arbre gauche d'un nœud x est constitué des nœuds accessibles depuis x via son pointeur enfant gauche et de même pour le sous-arbre droit.

La propriété caractéristique d'un arbre de recherche est la suivante. Elle doit être assurée pour tous les nœuds de l'arbre, et pas uniquement à la racine :

1. Pour chaque objet x, tous les objets du sous-arbre gauche de x ont des clés plus petites que celle de x.
2. Pour chaque objet x, tous les objets du sous-arbre droit de x ont des clés plus grandes que celle de x.

Voici par exemple un arbre de recherche contenant des objets dont les clés sont {1, 2, 3, 4, 5}, ainsi qu'un tableau indiquant la destination des trois pointeurs (parent, enfant gauche, enfant droit) pour chaque nœud. Les feuilles sont les nœuds situés en bas de l'arbre, c'est-à-dire ceux qui n'ont aucun enfant :



Noeud	Parent	Enfant gauche	Enfant droit
1	3	null	2
2	1	null	null
3	null	1	5
4	5	null	null
5	3	4	null

Figure 16: Exemple d'un arbre de recherche et ses pointeurs parents et enfants correspondants

II.7.4. Différences entre les tas et les arbres de recherche

Les arbres binaires de recherche et les tas diffèrent à plusieurs égards. Les tas peuvent être considérés comme des arbres, mais ils sont implémentés sous forme de tableaux, sans pointeurs explicites entre les objets. Un arbre de recherche, en revanche, stocke explicitement trois pointeurs par objet (parent, enfant gauche et enfant droit), ce qui entraîne une consommation d'espace plus élevée (d'un facteur constant). Les tas n'ont pas besoin de pointeurs explicites, car ils correspondent toujours à des arbres binaires complets, tandis que les arbres binaires de recherche peuvent avoir une structure arbitraire.

Les arbres de recherche ont un objectif différent de celui des tas. C'est pourquoi leurs propriétés caractéristiques sont différentes. Les tas sont optimisés pour les calculs rapides de minimum : la propriété caractéristique du tas, selon laquelle la clé d'un enfant est toujours supérieure à celle de son parent, rend très facile la recherche de l'objet ayant la clé minimale (c'est toujours la racine).
Les arbres de recherche, eux, sont optimisés pour la recherche (comme leur nom l'indique). Leur propriété est donc définie dans ce but : par exemple, si vous recherchez un objet dont la clé est 23 dans un arbre de recherche et que la clé de la racine est 17, vous savez immédiatement que l'objet recherché se trouve dans le sous-arbre droit, et vous pouvez ignorer tout le sous-arbre gauche. Ce mécanisme rappelle celui de la recherche dichotomique, ce qui est logique pour une structure de données dont la raison d'être est de simuler un tableau trié qui évolue dynamiquement.

II.7.5. Hauteur d'un arbre de recherche

Il peut exister de nombreux arbres de recherche différents pour un même ensemble de clés. Voici, par exemple, un deuxième arbre de recherche contenant des objets dont les clés sont {1, 2, 3, 4, 5}. Les deux conditions de la propriété de l'arbre de recherche sont respectées, la seconde de manière triviale (puisqu'il n'existe aucun sous-arbre droit non vide) :

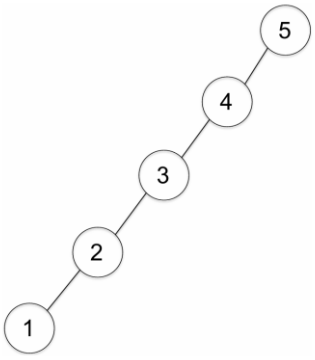


Figure 17 : Exemple d'un arbre de recherche avec les sous-arbres droits vides

La hauteur d'un arbre est définie comme la longueur du plus long chemin reliant sa racine à une feuille. Différents arbres de recherche contenant le même ensemble d'objets peuvent avoir des hauteurs différentes, comme dans nos deux premiers exemples (dont les hauteurs sont respectivement 2 et 4).

De manière générale, un arbre binaire de recherche contenant n objets peut avoir une hauteur comprise entre $\log_2 n$ (dans le meilleur des cas, pour un arbre parfaitement équilibré) et n-1 (dans le pire des cas, pour une chaîne).

II.7.6. Implantation de l'opération de recherche

L'opération de recherche consiste, pour une clé k, à renvoyer un pointeur vers l'objet de la structure de données ayant cette clé, ou indiquer qu'aucun objet correspondant n'existe. La propriété de l'arbre de recherche indique exactement où chercher un objet de clé k. Si k est inférieure (respectivement supérieure) à la clé de la racine, alors l'objet recherché doit se trouver dans le sous-arbre gauche (respectivement droit). Pour effectuer la recherche, il suffit de suivre ce principe simple : commencer à la racine, puis descendre successivement à gauche ou à droite selon la comparaison, jusqu'à trouver l'objet souhaité (recherche réussie) ou atteindre un pointeur nul (recherche infructueuse).

Le temps d'exécution est proportionnel au nombre de pointeurs suivis, ce qui correspond au maximum à la hauteur de l'arbre de recherche (ou à cette hauteur + 1, si l'on compte le dernier pointeur nul dans le cas d'une recherche infructueuse).

II.7.7. Implantation de l'opération de recherche du minimum et du maximum

L'opération consiste à renvoyer un pointeur vers l'objet de la structure de données ayant la plus petite (respectivement la plus grande) clé.

La propriété de l'arbre de recherche rend l'implémentation de ces opérations particulièrement simple. Les clés du sous-arbre gauche de la racine ne peuvent être que plus petites que la clé de la racine, et celles du sous-arbre droit ne peuvent être que plus grandes. Ainsi, si le sous-arbre gauche est vide, la racine doit être le minimum. Sinon, le minimum du sous-arbre gauche est également le minimum de tout l'arbre. Cela suggère de suivre le pointeur enfant gauche de la racine et de répéter ce processus jusqu'à atteindre la feuille la plus à gauche.

En suivant de manière répétée les pointeurs vers les enfants gauches, on atteint l'objet ayant la clé minimale (et inversement pour la clé maximale).

Le temps d'exécution est proportionnel au nombre de pointeurs suivis, ce qui correspond à O(hauteur).

II.7.8. Implantation des opérations de recherche du prédécesseur et du successeur

L'opération de recherche du prédécesseur consiste , étant donné un pointeur vers un objet dans la structure de données, à renvoyer un pointeur vers l'objet ayant la clé immédiatement inférieure (si l'objet possède déjà la clé minimale, renvoyer « aucun »). L'opération de recherche du successeur est analogue.

Étant donné un objet x, où pourrait se trouver son prédécesseur ? Certainement pas dans le sous-arbre droit de x, puisque, selon la propriété de l'arbre de recherche, toutes les clés de ce sous-arbre sont supérieures à celle de x.

Dans l'exemple ci-dessous, les prédécesseur des clés 3 ou 5 peuvent se trouver soit dans le sous-arbre gauche, et ceux des clés 2 ou 4 peuvent être un ancêtre plus haut dans l'arbre :

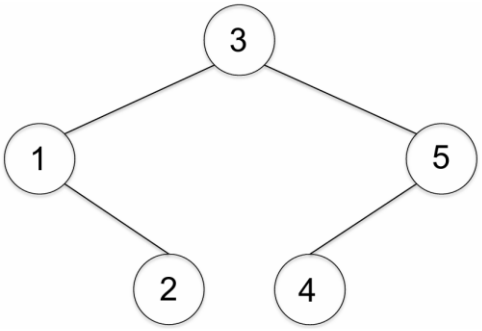


Figure 18 : Illustration des deux possibilités pour trouver le prédécesseur

Le schéma général est le suivant : si le sous-arbre gauche d'un objet x n'est pas vide, alors le maximum de ce sous-arbre est le prédécesseur de x. Sinon, le prédécesseur de x est le plus proche ancêtre de x dont la clé est inférieure à celle de x.

Autrement dit, si l'on remonte les pointeurs parent à partir de x, le prédécesseur est le premier nœud rencontré après avoir pris un « virage à gauche ». Dans notre exemple, si l'on remonte depuis le nœud de clé 4, on prend d'abord un virage à droite (vers un nœud de clé plus grande, 5), puis un virage à gauche, ce qui nous amène au bon prédécesseur (3).

Si x n'a pas de sous-arbre gauche ni aucun virage à gauche au-dessus de lui, alors il s'agit du minimum de l'arbre de recherche et il n'a pas de prédécesseur (comme le nœud de clé 1 dans l'exemple).

Le temps d'exécution est proportionnel au nombre de pointeurs suivis, ce qui, dans tous les cas, correspond à O(hauteur).

II.7.9. Implantation de l'opération de sortie triée

L'opération de sortie triée consiste à renvoyer les objets de la structure de données un par un, dans l'ordre croissant de leurs clés.

Une manière simple (mais peu efficace) d'implémenter cette opération consisterait à utiliser d'abord l'opération de recherche du minimum pour trouver l'objet ayant la clé minimale, puis à appeler plusieurs fois la fonction de recherche du successeur pour afficher ensuite tous les autres objets dans l'ordre.

Une méthode plus efficace consiste à traiter récursivement le sous-arbre gauche de la racine, puis la racine elle-même et enfin le sous-arbre droit.

Dans notre exemple précédent :

- On visite d'abord le sous-arbre gauche,
- On arrive sur 1, qui n'a pas de sous-arbre à gauche, donc on affiche [1],
- On visite le sous-arbre droit,
- On arrive sur 2, qui n'a pas de sous-arbre à gauche ni droit, donc on affiche [2]

- À ce stade, on a affiché [1,2]
- On retourne à la racine, et on affiche la valeur [3],
- On passe au sous-arbre droit,
- On arrive sur 5, on prend le sous-arbre gauche,
- On arrive sur 4, qui n'a pas de sous-arbre gauche, ni droit donc on affiche [4],
- On retourne sur 5, qui n'a pas de sous-arbre droit, donc on affiche [5]
- On trouve finalement [1,2,3,4,5]

Pour un arbre contenant n objets, l'opération effectue n appels récursifs (un pour chaque nœud) et réalise une quantité constante de travail à chaque appel, soit un temps d'exécution total de O(n).

II.7.10. Implantation de l'opération d'insertion

Aucune des opérations abordées jusqu'à présent ne modifie l'arbre de recherche donné ; elles ne risquent donc pas d'altérer la propriété essentielle de l'arbre de recherche.

Les deux opérations d'insertion et de suppression modifient, quant à elles, la structure de l'arbre et doivent donc veiller à préserver ses propriétés caractéristiques.

L'opération d'insertion s'appuie sur l'opération de recherche. Une recherche infructueuse d'un objet ayant la clé k permet de déterminer l'endroit exact où cet objet aurait dû se trouver. C'est à cet emplacement qu'il faut insérer le nouvel objet portant la clé k, en remplaçant le pointeur nul rencontré à la fin de la recherche.

Dans notre exemple, la position correcte pour un nouvel objet de clé 6 correspond donc à l'endroit où la recherche infructueuse s'est arrêtée :

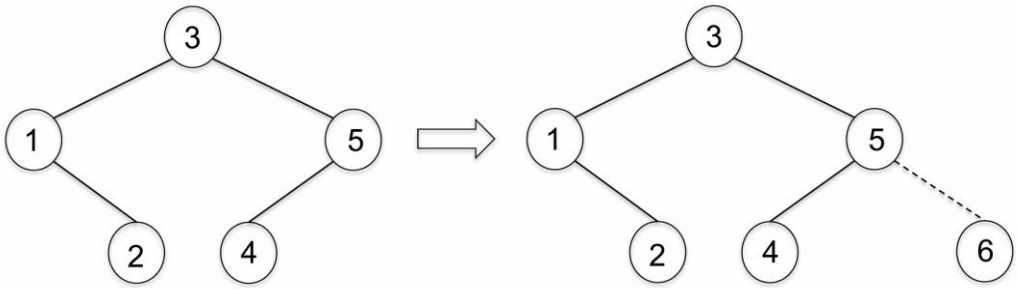


Figure 19 : Illustration de l'opération d'insertion

Dans le cas où il existe déjà un objet ayant la clé k dans l'arbre, si on souhaite éviter les doublons de clés, on peut simplement ignorer l'insertion. Sinon, la recherche continue à suivre le pointeur enfant gauche de l'objet existant ayant la clé k, et se poursuit jusqu'à rencontrer un pointeur nul, où le nouvel objet pourra alors être inséré.

L'opération préserve la propriété de l'arbre de recherche, car elle place le nouvel objet exactement à l'endroit où il devait se trouver. Son temps d'exécution est identique à celui de l'opération de recherche, c'est-à-dire O(hauteur).

II.7.11. Implantation de l'opération de suppression

Le principal défi consiste à réparer l'arbre après la suppression d'un nœud, de manière à restaurer la propriété de l'arbre de recherche.

La première étape consiste à utiliser l'opération de recherche pour localiser un objet x de clé k (s'il n'existe aucun objet avec cette clé, alors l'opération de suppression n'a rien à faire). Il existe trois cas possibles, selon que x possède 0, 1 ou 2 enfants. Si x est une feuille, il peut être supprimé sans conséquence. Par exemple, si l'on supprime le nœud de clé 2 dans notre arbre de recherche :

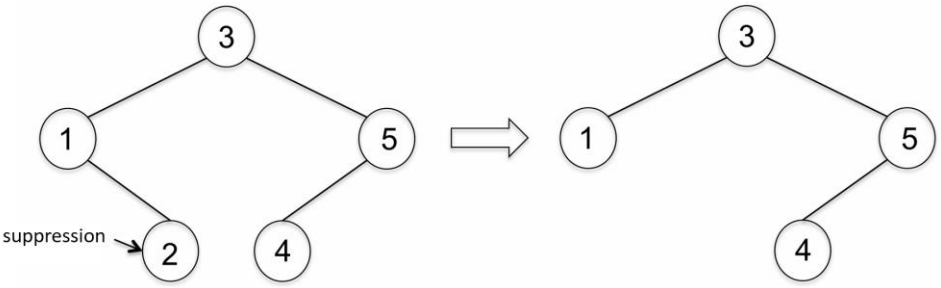


Figure 20 : Exemple d'une opération de suppression sur une feuille sans conséquence

Pour chaque nœud restant y, les nœuds de ses sous-arbres restent inchangés, sauf que le nœud x a éventuellement été supprimé ; ainsi, la propriété de l'arbre de recherche demeure valide.

Lorsque x possède un seul enfant y, on peut simplement le retirer. La suppression de x laisse alors y sans parent, et l'ancien parent de x, noté z, sans un de ses enfants. La solution naturelle consiste à faire occuper à y la position qu'occupait x, c'est-à-dire à relier y comme nouvel enfant de z. Par exemple, si l'on supprime le nœud de clé 5 dans notre arbre de recherche. La propriété de l'arbre de recherche demeure valide.

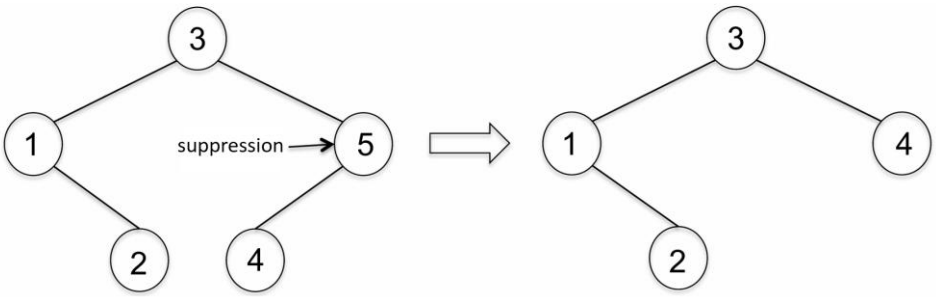


Figure 21: Exemple de suppression sur un nœud n'ayant qu'un seul enfant

Le cas difficile survient lorsque x possède deux enfants. La suppression de x laisse alors deux nœuds sans parent, et il n'est pas évident de savoir comment les rattacher correctement. Dans notre exemple, il n'est pas clair comment réparer l'arbre après avoir supprimé sa racine.

L'astuce consiste à réduire le cas difficile à l'un des cas simples. Pour cela, on commence par utiliser l'opération de recherche du prédécesseur afin de trouver le prédécesseur y de x. Puisque x possède deux enfants, son prédécesseur se trouve dans son sous-arbre gauche (non vide) et correspond à l'objet ayant la clé maximale dans ce sous-arbre. Comme ce maximum est obtenu en suivant les pointeurs enfants droits aussi loin que possible, le nœud y ne peut pas avoir d'enfant droit ; il peut ou non avoir un enfant gauche.

L'idée un peu audacieuse est d'échanger x et y. Dans notre exemple, où le nœud racine joue le rôle de x :

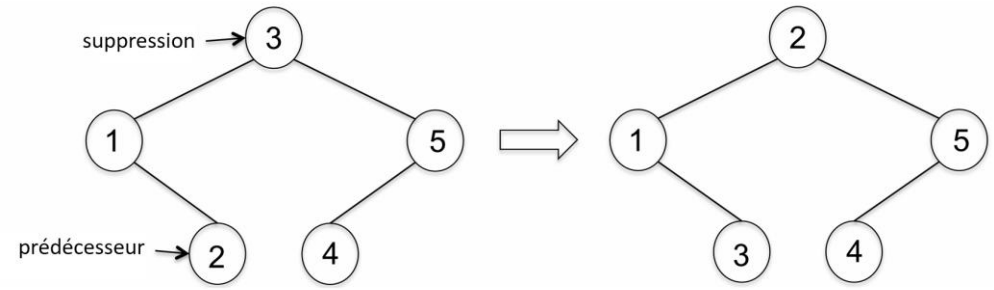


Figure 22 : Échange d'un nœud avec son prédécesseur avant sa suppression

Cette idée peut sembler mauvaise au premier abord, car elle viole temporairement la propriété de l'arbre de recherche (par exemple, le nœud de clé 3 se retrouve dans le sous-arbre gauche d'un nœud de clé 2). Mais chaque violation de la propriété concerne uniquement le nœud x, que nous allons de toute façon supprimer. Puisque x occupe maintenant la position précédemment occupée par y, il n'a plus d'enfant droit. La suppression de x dans cette nouvelle position se ramène donc à l'un des deux cas simples : si x n'a pas d'enfant gauche, on le supprime directement ; s'il en a un, on le retire comme précédemment. Dans les deux cas, une fois x supprimé, la propriété de l'arbre de recherche est restaurée.

Revenons à notre exemple :

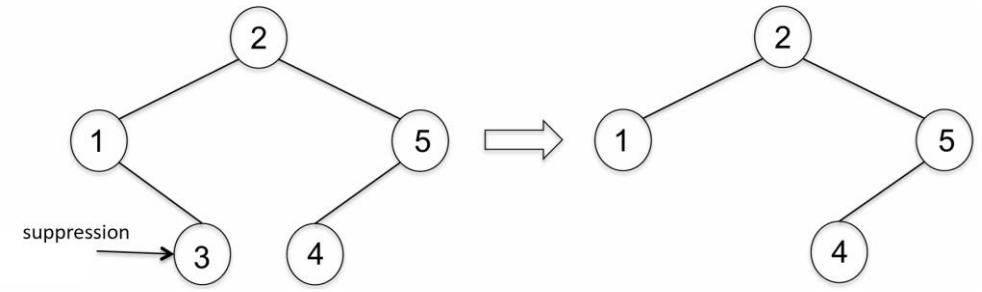


Figure 23 : Dans notre exemple, après l'échange, le nœud à supprimer est définitivement enlevé

L'opération effectue une quantité constante de travail, en plus d'une opération de recherche et d'une opération de recherche de prédécesseur ; elle s'exécute donc en O(hauteur).

II.7.12. Implantation de l'opération de sélection

L'opération de sélection consiste, étant donné un nombre i compris entre 1 et le nombre total d'objets, à renvoyer un pointeur vers l'objet de la structure de données ayant la i^e plus petite clé.

Pour que l'opération Select s'exécute rapidement, on peut enrichir (augmenter) la structure de l'arbre de recherche, en faisant en sorte que chaque nœud conserve des informations sur la structure de l'arbre et pas seulement sur l'objet qu'il contient. Les arbres de recherche peuvent être augmentés de plusieurs manières ; dans notre cas, on ajoute à chaque nœud x un entier size(x) qui indique le nombre total de nœuds dans le sous-arbre enraciné en x (y compris x lui-même). Dans notre exemple :

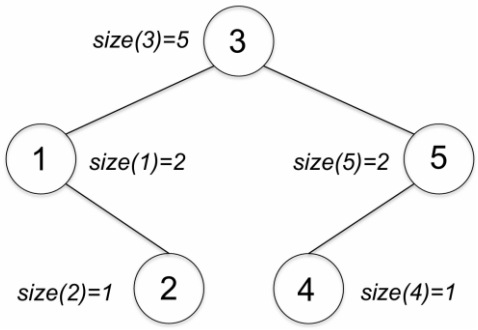


Figure 24 : Arbre de recherche augmenté

Supposons que le nœud x d'un arbre de recherche ait pour enfants y et z. Alors chaque nœud du sous-arbre enraciné en x est soit x lui-même, soit un nœud du sous-arbre gauche de x, soit un nœud du sous-arbre droit de x. Nous avons donc la relation suivante : size(x) = size(y) + size(z) + 1

Imaginons maintenant que l'on cherche l'objet ayant la 17^e plus petite clé (i = 17) dans un arbre de recherche contenant 100 objets.

En partant de la racine, on peut calculer en temps constant la taille des sous-arbres gauche et droit. Grâce à la propriété de l'arbre de recherche, on sait que toutes les clés du sous-arbre gauche sont plus petites que celles de la racine et du sous-arbre droit. Si le sous-arbre gauche contient 25 nœuds, alors ces 25 nœuds représentent les 25 plus petites clés de l'arbre, et la 17^e plus petite clé se trouve donc dans ce sous-arbre gauche.

Si, au contraire, il ne contient que 12 nœuds, cela signifie que le sous-arbre droit contient les 87 clés restantes, et que la 17^e plus petite clé est en réalité la 4^e plus petite clé de ce sous-arbre droit (puisque 17 - 12 - 1 = 4).

Dans les deux cas, on peut alors appeler récursivement l'opération de sélection pour localiser l'objet recherché.

Puisque chaque nœud de l'arbre de recherche stocke la taille de son sous-arbre, chaque appel récursif n'effectue qu'une quantité constante de travail. De plus, chaque appel récursif descend d'un niveau supplémentaire dans l'arbre, de sorte que la quantité totale de travail est de O(hauteur).

II.8. Les arbres de recherche binaires équilibrés

II.8.1. Ajouter de la complexité pour obtenir de meilleures performances

Le temps d'exécution de chaque opération sur un arbre binaire de recherche (sauf l'opération de sortie triée) est proportionnel à la hauteur de l'arbre, laquelle peut varier : du meilleur cas, environ $\log_2(n)$ (pour un arbre parfaitement équilibré), au pire cas, soit n − 1 (pour un arbre dégénéré en chaîne), où n est le nombre d'objets dans l'arbre. Des arbres de recherche très déséquilibrés peuvent réellement apparaître, par exemple lorsque les objets sont insérés dans un ordre déjà trié ou inversement trié :

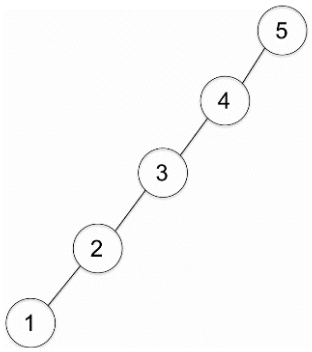


Figure 25 : Arbre binaire non équilibré

La différence entre un temps d'exécution logarithmique et linéaire est considérable et il vaut donc la peine de complexifier légèrement les opérations d'insertion et de suppression, tout en restant en O(hauteur), mais avec un facteur constant un peu plus grand afin de garantir que la hauteur de l'arbre reste toujours O(log n).

Un arbre binaire de recherche équilibré maintient donc sa hauteur minimale en réorganisant ses nœuds lors des insertions et suppressions. Plusieurs types d'arbres de recherche équilibrés garantissent une hauteur en O(log n), par exemple les AVL et les arbres rouge-noir. Ces structures inspirent directement les B-trees et B+trees, utilisés dans les SGBD pour gérer les index et accélérer les requêtes (ex. jointures, recherches).

Cependant, les détails d'implémentation peuvent devenir assez complexes. Nous allons donc ici survoler les idées les plus répandues dans la mise en œuvre des arbres de recherche équilibrés.

II.8.2. Principe de rééquilibrage d'un arbre à l'aide des rotations

Toutes les implémentations les plus courantes d'arbres de recherche équilibrés utilisent des rotations, une opération en temps constant qui permet d'effectuer un rééquilibrage local limité tout en préservant la propriété de l'arbre de recherche.

Par exemple, on peut imaginer transformer une chaîne de cinq objets (très déséquilibrée) en un arbre de recherche plus équilibré, en effectuant une composition de deux opérations locales de rééquilibrage.

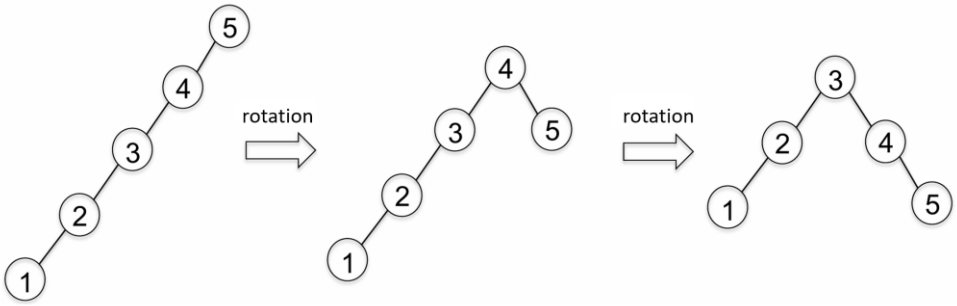


Figure 26 : Rééquilibrage d'un arbre à l'aide de rotations

Une rotation prend une paire parent-enfant et inverse leur relation. Une rotation à droite s'applique lorsque le nœud enfant y est le fils gauche de son parent x (donc y possède une clé plus petite que x). Après la rotation, x devient le fils droit de y. Inversement, lorsqu'un nœud y est le fils droit de x, une rotation à gauche transforme x en fils gauche de y.

La propriété de l'arbre de recherche détermine les détails restants de la rotation. Prenons par exemple (voir ci-dessous) une rotation à gauche, où y est le fils droit de x. Cette propriété implique que la clé de x est inférieure à celle de y, toutes les clés du sous-arbre gauche de x sont inférieures à celles de x (et donc à celles de y), toutes les clés du sous-arbre droit de y sont supérieures à celles de y (et donc à celles de x), enfin, toutes les clés du sous-arbre gauche de y se trouvent entre les clés de x et de y.

Après la rotation, y hérite de l'ancien parent de x et x devient le fils gauche de y. Il n'existe qu'une seule manière cohérente de réassembler ces sous-arbres tout en préservant la propriété de l'arbre de recherche. Il suffit donc de suivre cette logique naturelle.

Il existe trois emplacements disponibles pour rattacher les sous-arbres A, B et C : le pointeur d'enfant droit de y, ainsi que les deux pointeurs d'enfants de x. La propriété de l'arbre de recherche nous oblige à placer le plus petit sous-arbre (A) comme fils gauche de x et le plus grand sous-arbre (C) comme fils droit de y. Il reste alors un seul emplacement pour le sous-arbre B : le pointeur d'enfant droit de x. Heureusement, la propriété de l'arbre de recherche est respectée : toutes les clés du sous-arbre B sont comprises entre celles de x et de y, ce qui fait que ce sous-arbre se retrouve à la fois dans le sous-arbre gauche de y (là où il doit être) et dans le sous-arbre droit de x, exactement comme prévu.

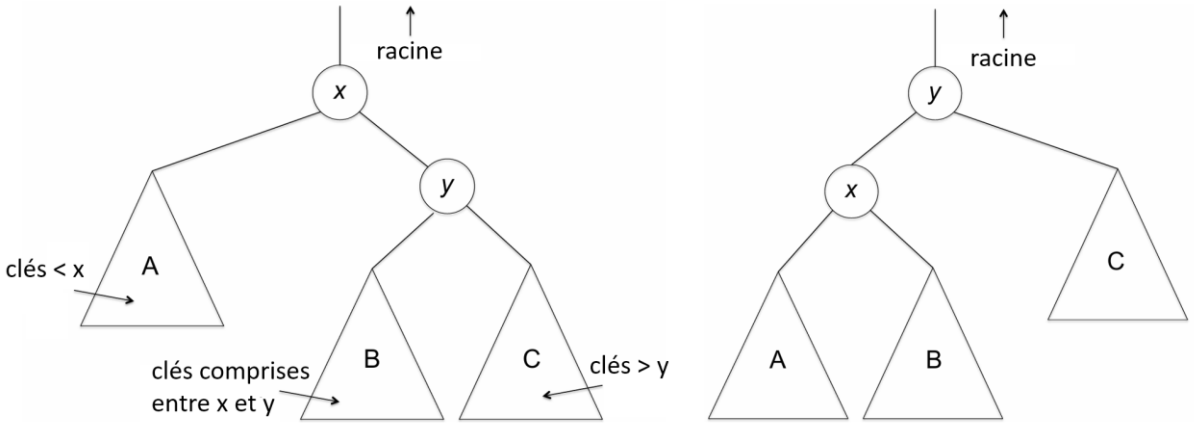


Figure 27 : Exemple d'une rotation à gauche : avant la rotation (à gauche) et après la rotation (à droite)

Une rotation à droite n'est rien d'autre qu'une rotation à gauche effectuée en sens inverse :

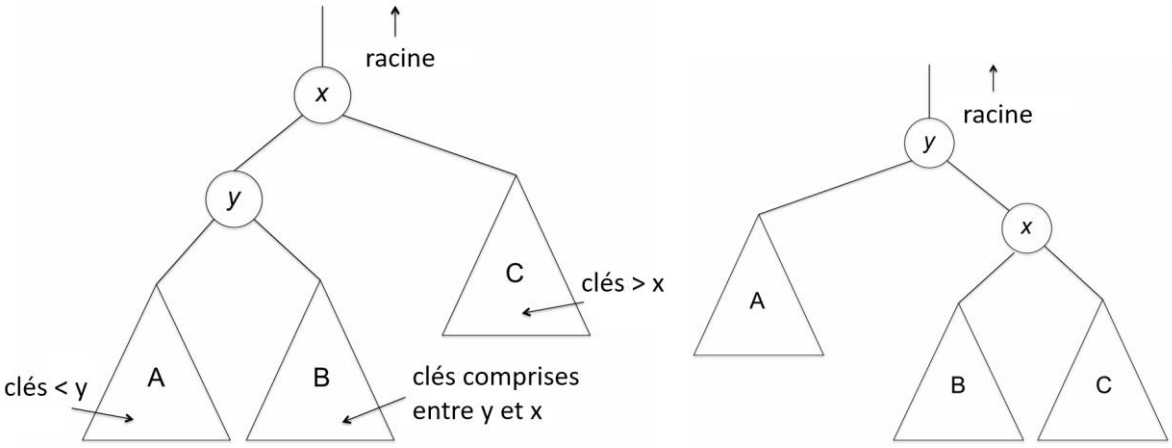


Figure 28 : Exemple d'une rotation à droite : avant la rotation (à gauche) et après la rotation (à droite)

Comme une rotation ne fait que réarranger quelques pointeurs, elle peut être implémentée avec un nombre constant d'opérations. De plus, par construction, elle préserve la propriété de l'arbre de recherche.

Les opérations qui modifient l'arbre de recherche, c'est-à-dire les opérations d'insertion et de suppression, sont précisément celles qui doivent utiliser des rotations.

Sans ces rotations, une telle opération pourrait déséquilibrer légèrement l'arbre. Cependant, comme une seule insertion ou suppression ne peut provoquer qu'un déséquilibre local limité, il est raisonnable de penser qu'un petit nombre de rotations, constant ou logarithmique, suffit à corriger tout déséquilibre nouvellement créé.

C'est exactement ce que font les arbres de recherche équilibrés mentionnés précédemment. Le travail supplémentaire induit par ces rotations ajoute un surcoût de O(log n) aux opérations d'insertion et de suppression, tout en maintenant leur complexité globale à O(log n).

II.8.3. Les arbres de recherche équilibrés AVL

Les arbres AVL (du nom d'Adelson-Velsky et Landis, leurs inventeurs en 1962) sont les premiers arbres binaires de recherche auto-équilibrés. Le principe d'équilibrage est de faire en sorte que pour chaque nœud, la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit (appelée facteur d'équilibre) soit au plus égale à 1.

Dès qu'une insertion ou une suppression rompt cette condition, l'arbre est rééquilibré à l'aide de rotations simples ou doubles.

L'arbre reste très bien équilibré, ce qui assure une hauteur minimale. Les opérations de recherche sont donc très rapides, toujours en O(log n).

L'inconvénient est que les opérations d'insertion et de suppression nécessitent parfois plusieurs rotations, ce qui les rend un peu plus coûteuses que dans d'autres arbres équilibrés.

II.8.4. Les arbres de recherche équilibrés Rouge-Noir

Les arbres Rouge-Noir sont une autre forme d'arbre binaire de recherche auto-équilibré, plus souple que les AVL.

Chaque nœud est coloré en rouge ou noir, et l'arbre doit respecter certaines règles de couleur garantissant un équilibre global :

1. Chaque nœud est soit rouge, soit noir.
2. La racine est toujours noire.
3. Les feuilles nulles (pointeurs vides) sont considérées comme noires.
4. Un nœud rouge ne peut pas avoir de fils rouges.
5. Pour tout nœud, tous les chemins de ce nœud à ses feuilles contiennent le même nombre de nœuds noirs.

Ces règles assurent que la hauteur noire (le nombre de nœuds noirs sur un chemin) reste équilibrée, garantissant une hauteur maximale de $2\log_2(n+1)$.

Les opérations d'insertion et de suppression sont plus simples que dans un arbre AVL (moins de rotations nécessaires). L'arbre reste raisonnablement équilibré avec une complexité en O(log n).

Cependant, l'arbre est moins strictement équilibré qu'un AVL, donc la recherche est parfois légèrement plus lente (même si toujours logarithmique). Les arbres AVL et Rouge-Noir visent le même objectif : maintenir un arbre de recherche binaire équilibré afin de garantir des opérations logarithmiques.

Les AVL privilégient la recherche rapide grâce à un équilibre plus strict. Les Rouge-Noir privilégient la rapidité des mises à jour (insertions/suppressions) grâce à une tolérance d'équilibre plus souple.

II.9. Les arbres de recherche équilibrés B-trees et B+trees

II.9.1. Définition générale

Les B-trees (ou arbres B) sont des arbres de recherche équilibrés à plusieurs branches (contrairement aux arbres binaires, qui n'ont que deux enfants par nœud). Un B-tree peut avoir des dizaines, voire des centaines d'enfants par nœud, ce qui le rend particulièrement adapté aux systèmes de stockage sur disque.

Chaque nœud d'un B-tree contient plusieurs clés et pointeurs vers ses sous-arbres, et les clés sont triées à l'intérieur du nœud.

Les B-trees étant équilibrés, toutes les feuilles se trouvent au même niveau, ce qui garantit une hauteur logarithmique en $O(\log_m n)$ (avec m = nombre d'enfants max).

II.9.2. Propriété caractéristique d'un B-tree

Un B-tree généralise la logique d'un arbre binaire de recherche à plusieurs clés par nœud.

Un B-tree d'ordre m vérifie que :

1. Chaque nœud (sauf la racine) contient entre ⌈m/2⌉ et m clés triées dans l'ordre croissant,
2. La structure d'ordre des clés respecte celle des arbres de recherche binaires,
3. Chaque nœud interne a entre ⌈m/2⌉ et m+1 enfants,
4. Les clés à l'intérieur d'un nœud séparent les intervalles de valeurs stockés dans les sous-arbres,
5. La racine a au moins 2 enfants (sauf si l'arbre ne contient qu'un nœud).

L'exemple ci-dessous est un B-tree d'ordre m=4. Un nœud interne (index) peut donc contenir jusqu'à 4 enfants (et donc jusqu'à 3 clés). Chaque clé sépare les intervalles des valeurs stockées dans les sous-arbres :

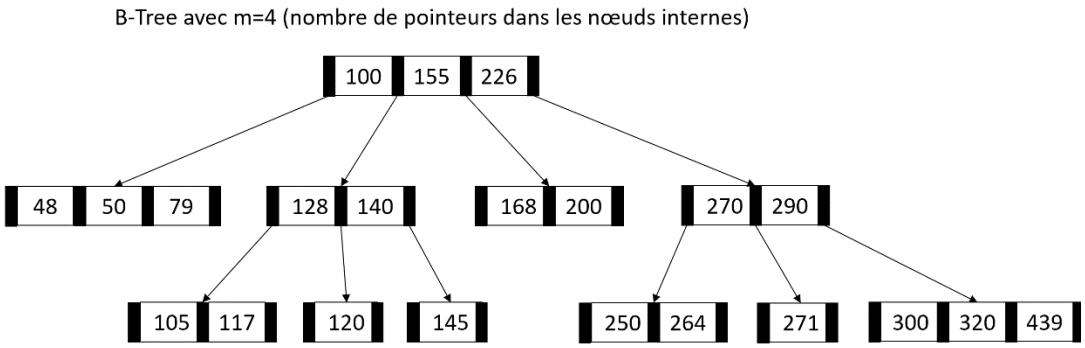


Figure 29 : Exemple d'un B-tree d'ordre 4

Grâce à ces règles, l'arbre reste équilibré en permanence, et ses opérations de recherche, insertion et suppression s'effectuent en $O(\log_m n)$.

II.9.3. Les B+trees

Les B+trees sont une amélioration des B-trees. Ils en reprennent la structure générale, mais avec deux différences majeures :

1. Toutes les données (valeurs associées aux clés) sont stockées dans les feuilles uniquement. Les nœuds internes ne contiennent que des clés servant d'index.
2. Tous les nœuds ont la même taille logique, c'est-à-dire qu'ils occupent le même espace mémoire (cette taille correspond généralement à une page disque).
3. Les feuilles sont chaînées entre elles (comme une liste doublement chaînée), ce qui permet un balayage séquentiel rapide des données (par exemple pour les tris ou les parcours d'index).

Dans l'exemple ci-dessous, le B-tree est d'ordre m=4 ; un nœud interne (index) peut donc contenir jusqu'à 4 enfants (et donc jusqu'à 3 clés). La capacité est L=5 donc chaque feuille peut contenir jusqu'à 5 valeurs.

La racine contient deux clés d'index : 12 et 44. Ces clés découpent l'espace de valeurs en trois intervalles : < 12, [12 ; 44[et ≥ 44. Chaque feuille contient jusqu'à 5 valeurs triées. Les feuilles sont chainées (elles pointent vers la suivante) pour permettre un parcours séquentiel. Le parcours séquentiel des feuilles donne 1,2,3,4,6,8,… soit un ensemble trié de valeurs.

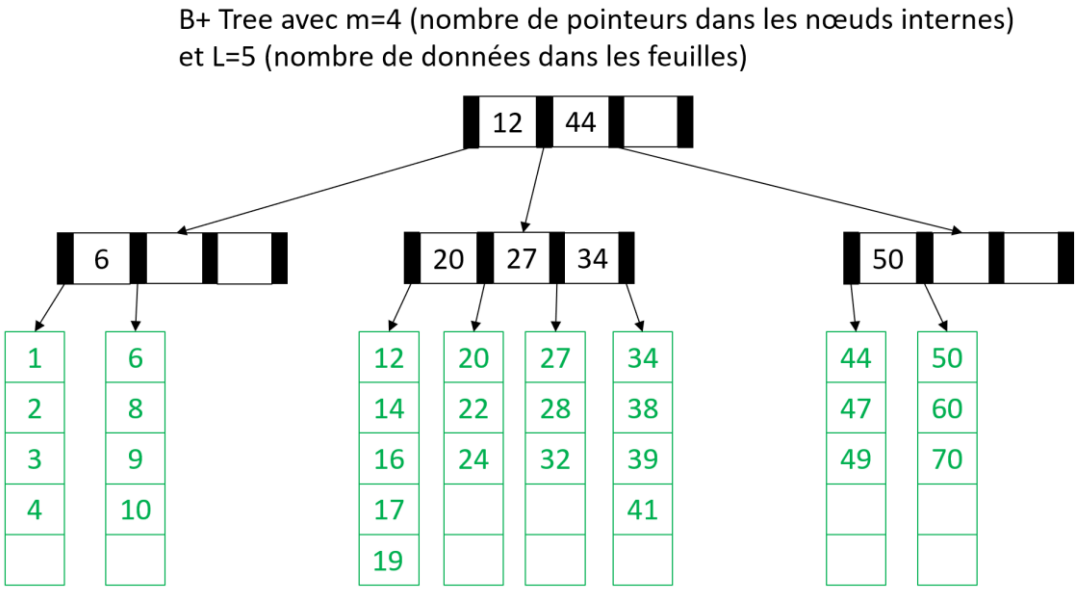


Figure 30 : Exemple d'un B+tree

Les B+trees sont également équilibrés, et ont la même hauteur logarithmique que les B-trees.

II.9.4. Comparaison avec les arbres AVL et Rouge-Noir

Caractéristique	AVL / Rouge-Noir	B-tree / B+tree
Type d'arbre	Binaire (2 enfants max)	Multi-aire (jusqu'à m enfants)
Équilibrage	Oui (hauteur ≈ $\log_2 n$)	Oui (hauteur ≈ $\log_m n$, beaucoup plus petit)
Stockage	En mémoire vive	Sur disque ou SSD
Données	1 clé par nœud	Plusieurs clés par nœud
Objectif	Accès rapide en mémoire (petite structure)	Réduction du nombre d'accès disque
Parcours séquentiel	Lent (récursif)	Rapide (feuilles chaînées dans les B+trees)
Taille typique	Quelques milliers de nœuds	Quelques centaines de nœuds (chaque nœud peut stocker des centaines de clés)

II.9.5. Utilisation des B-trees et des B+trees dans les SGBD

Les Systèmes de Gestion de Bases de Données (SGBD) gèrent des volumes de données massifs, souvent stockés sur disque. Or, les accès disques sont extrêmement coûteux en temps (comparés aux accès mémoire).

Les B-trees et B+trees ont été conçus pour minimiser le nombre d'accès aux disques :

- chaque nœud correspond à une page disque (souvent 4 à 8 Ko),
- une recherche ne nécessite que quelques lectures de pages (grâce à la faible hauteur de l'arbre),
- les insertions et suppressions modifient peu de nœuds,
- les B+trees permettent un parcours séquentiel efficace des données triées (utile pour les index et les tris d'enregistrements).

Par exemple, si chaque nœud contient 100 clés, un B+tree de 3 niveaux peut indexer environ 1 million d'enregistrements, avec seulement 3 lectures disque pour trouver une donnée.

Les arbres binaires équilibrés comme les AVL ou les Rouge-Noir, eux, ont des nœuds très petits (seulement deux enfants). Cela rend leur hauteur beaucoup plus grande et donc il faut beaucoup plus d'accès disque, ce qui est inefficace pour les bases de données.