Sorbonne Université - PSTL

Rapport du Projet STL - Bibliothèque unifiée pour les structures de données d'AlgAv

Auteurs :
Tracy Hong
Julien Fang
Jean-Marc Zhuang
Laura Rattanavan

Encadrant :
Antoine Genitrini

M1 Informatique, 22 mars 2024

Table des matières

Introduction

Dans le cadre du projet du Master STL (PSTL), s'inscrivant de manière étroite avec le cours d'Algorithmique Avancée (AlgAv) du M1, l'objectif principal est la création d'une bibliothèque unifiée en Python afin de simplifier la manipulation et la visualisation des différentes structures de données arborescentes enseignées.

Un aspect significatif de ce projet réside dans l'intégration des fonctionnalités de visualisation. En tirant parti de GraphViz et de ses structures adaptées aux graphes dirigés (Digraph), la bibliothèque permettra une représentation visuelle claire et compréhensible des données arborescentes. Elle contribuera ainsi à renforcer la conceptualisation des structures étudiées dans le cours d'AlgAv.

Un autre élément crucial de ce projet est l'unification. Toutes les structures envisagées sont des arbres de recherche, et l'objectif est de simplifier la manipulation en permettant une utilisation cohérente. Cela se traduit par la conception de fonctions similaires, facilitant ainsi la manipulation des structures de manière intuitive et uniforme. La bibliothèque sera construite sous la forme de notebook Jupyter : un notebook regroupant toutes les structures offerte par la bibliothèque, ainsi qu'un notebook pour chaque structure différente.

1 Présentation du projet et cahier des charges

L'objectif fondamental est de créer une bibliothèque en Python qui offre une approche unifiée pour manipuler ces structures hétérogènes. Bien que chaque structure puisse avoir son implémentation spécifique, l'accent est mis sur la standardisation des fonctionnalités clés. Ainsi, peu importe la complexité ou la diversité des structures, chaque structure a les mêmes fonctionnalités : ajouter, supprimer et rechercher un élément de manière similaire.

Au sein du cours d'AlgAv, une diversité de structures de données arborescentes a été explorée. Parmi celles-ci figurent celles offertes par la bibliothèque : le Tas Min, la File Binomiale, l'Arbre Binaire de Recherche, l'AVL, le Splay-tree, l'Arbre 2-3-4, l'Arbre B, le R-Trie, le Trie Hybride, l'Arbre Digital et l'Arbre Lexicographique. Ces structures, chacune caractérisée par ses propriétés et son utilité spécifiques, sont des arbres de recherches offrant des fonctionnalités communes.

De plus, chaque implémentation est réalisée de façon persistante, signifiant qu'à chaque interaction avec une structure de données, une copie de l'original est créée. Cela s'avère pratique dans divers contextes, permettant par exemple de conserver une trace des versions antérieures suite à chaque manipulation.

2 Présentation des structures

Cette partie est incomplète et ne couvre pas encore l'intégralité des détails sur l'implémentation de chaque structure de la bibliothèque. Elle sera complétée dans le rapport final.

2.1 Arbre Binaire de Recherche (ABR)

Un Arbre Binaire de Recherche (ABR) est une structure d'arbre binaire où chaque nœud détient une clé et respecte la propriété suivante : pour chaque nœud, toutes les clés dans son sous-arbre gauche ont des valeurs strictement inférieures à la sienne, et toutes les clés dans son sous-arbre droit ont des valeurs strictement supérieures.

- Ajout :
  Lors de l'ajout d'un élément dans un Arbre Binaire de Recherche (ABR), il est essentiel de maintenir la propriété de tri de l'arbre. Dans notre implémentation de l'ABR, nous choisissons de ne pas gérer des doublons, c'est-à-dire que nous n'autorisons pas l'ajout d'un élément déjà existant dans l'arbre. Ainsi, nous commençons par le comparer à la clé de la racine. Si l'élément est inférieur à la racine, nous nous déplaçons vers le sous-arbre gauche, sinon vers le sous-arbre droit. Nous répétons ce processus récursivement jusqu'à trouver une position où insérer le nouvel élément. Il y a alors plusieurs cas :
  — Si le nœud courant est vide, cela signifie que vous avez trouvé l'endroit approprié pour insérer le nouvel élément. Nous créons un nouveau nœud avec la valeur à insérer à cet endroit.
  — Si la clé du nœud courant est égale à l'élément à insérer, cela signifie que l'élément existe déjà dans l'arbre. La réaction dépend de la politique spécifique de gestion des doublons de l'ABR : nous avons décidé de ne pas gérer les doublons, et donc d'ignorer l'ajout, car l'élément existe déjà.

- Recherche :
  Sur le même principe, la recherche tire parti de la propriété fondamentale de l'ABR pour minimiser le nombre de comparaisons nécessaires pour trouver un élément. Nous commençons à la racine et comparons l'élément recherché avec la valeur du nœud. Si l'élément est inférieur, la recherche continue dans le sous-arbre gauche ; si l'élément est supérieur, elle se poursuit dans le sous-arbre droit. Si un nœud avec la valeur recherchée est trouvé, la recherche est réussie. Sinon, si nous atteignons un sous-arbre vide, la recherche échoue.

- Suppression :
  La suppression dans un ABR est un peu plus complexe et peut se présenter en trois cas :
  — Suppression d'un nœud feuille (sans enfants) : Dans ce cas, la suppression est simple. Nous supprimons simplement le nœud, et nous mettons à jour le parent du nœud à supprimer pour qu'il ne pointe plus vers lui.
  — Suppression d'un nœud avec un seul enfant : Lorsque le nœud à supprimer a un seul enfant, nous supprimons simplement le nœud et nous relions son parent directement à son unique enfant.
  — Suppression d'un nœud avec deux enfants : C'est le cas le plus complexe. Pour supprimer un nœud avec deux enfants, nous devons trouver un successeur *inorder* (le plus petit

nœud dans son sous-arbre droit) ou un prédécesseur *inorder* (le plus grand nœud dans son sous-arbre gauche). Nous copions ensuite la valeur de ce successeur ou prédécesseur dans le nœud à supprimer, puis nous supprimons le successeur ou prédécesseur.

(exemple + un pseudo code + complexité)

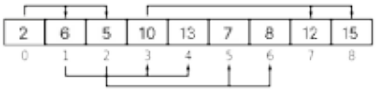
2.2 Tas Min

Un Tas Min est un arbre binaire étiqueté de façon croissante, dont toutes les feuilles sont situées au plus sur deux niveaux, les feuilles du niveau le plus bas étant positionnées le plus à gauche possible.

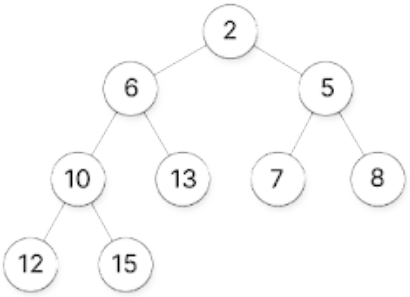
L'implémentation d'un tas min (min heap) peut se faire à l'aide d'une structure de donnée arborescente ou bien d'un tableau qui est une approche efficace et compacte qui tire parti de la structure d'arbre binaire complet du tas, et celle que nous avons choisi dans ce projet. Dans cette représentation, le premier élément du tableau (à l'indice 0) est l'élément racine de l'arbre, et pour tout noeud situé à l'indice i, ses enfants gauche et droit se trouvent respectivement aux indices $2i + 1$ et $2i + 2$. Cette méthode permet un accès rapide aux parents et aux enfants d'un nœud donné, ce qui est crucial pour les opérations d'ajout et de suppression.

- Ajout : Pour ajouter un élément au tas min, nous commençons par ajouter l'élément à la fin du tableau. Cela correspond à l'ajouter au niveau le plus bas et le plus à gauche possible de l'arbre binaire complet, tout en conservant sa structure. Puis, nous vérifions si cet élément nouvellement ajouté est inférieur à son parent. Si c'est le cas, cela viole la propriété du tas min, où chaque parent doit être plus petit que ses enfants. Pour corriger cela, nous échangeons l'élément avec son parent, puis nous répétons ce processus (nous continuons à échanger avec les parents successifs) jusqu'à ce que l'élément ne soit plus inférieur à son parent, ou jusqu'à ce qu'il atteigne la position de la racine du tas. Pour trouver l'indice du parent d'un élément donné à l'indice i, nous pouvons le récupérer grâce à la formule $\left\lfloor \frac{(i-1)}{2} \right\rfloor$.

- Suppression (extraction du minimum) : L'élément à supprimer dans un tas min est toujours la racine, puisqu'il s'agit de l'élément minimum. Nous échangeons l'élément racine avec le dernier élément du tas pour maintenir la forme complète de l'arbre binaire. Après avoir enlevé l'élément minimum (en le remplaçant par le dernier élément), cela peut violer la propriété du tas min si le nouvel élément racine est plus grand que ses enfants. Pour corriger cela, nous devons effectuer des échanges avec son enfant le plus petit jusqu'à ce que la propriété du tas soit rétablie (c'est-à-dire, jusqu'à ce qu'il soit plus petit que ses enfants ou qu'il atteigne une feuille).



(a) Par tableau



(b) Par arbre

FIGURE 1 – Deux représentations du Tas Min

(un pseudo code + complexité)

2.3 File Binomiale

Une File Binomiale est une structure de données puissante pour la gestion des files de priorité, combinant les avantages des arbres binaires et des opérations binaires. Il s'agit d'une structure de données qui combine plusieurs Arbres Binomiaux en une seule file ordonnée selon les valeurs de leurs racines. Chaque Arbre Binomial dans la File Binomiale suit la propriété des Arbres Binomiaux, c'est-à-dire qu'un Arbre Binomial de degré k a exactement 2^k nœuds.

Une File Binomiale maintient les Arbres Binomiaux dans un ordre spécifique en fonction du degré de l'arbre. Plus précisément, la File Binomiale est une liste d'Arbres Binomiaux, où chaque arbre est d'un degré différent. La File Binomiale garantit que la racine de chaque arbre est plus petite que celle des arbres de degré supérieur.

- Ajout :
- Suppression (extraction du minimum) :
- Union de deux files :

(explication des fonctions + exemple + un pseudo code + complexité)

Cette structure est en cours d'implémentation.

2.4 Arbre 2-3-4

L'Arbre 2-3-4 est une structure de données utilisée dans de nombreuses applications informatiques en raison de ses performances équilibrées et efficaces, par exemple dans les bases de données et les systèmes de fichiers pour stocker et organiser efficacement les données.

Chaque nœud de cet arbre peut contenir 2, 3 ou 4 enfants, et peut également stocker une ou plusieurs valeurs en fonction du nombre d'enfants qu'il possède. Sa caractéristique principale est son équilibre constant, garantissant ainsi une profondeur maximale limitée, ce qui maintient des performances stables pour les opérations d'insertion, de suppression et de recherche.

De plus, malgré la nécessité occasionnelle de réorganiser les nœuds lors des opérations d'insertion et de suppression, ces opérations conservent une complexité logarithmique dans le pire des cas.

- Ajout :

Tout d'abord, la fonction vérifie si l'élément x à ajouter est déjà présent dans l'arbre en appelant la méthode `EstDans`. Si c'est le cas, elle renvoie une copie de l'arbre actuel à l'aide de la méthode `Dupplique`, en spécifiant le parent le cas échéant. Ensuite, si le degré de l'arbre est égal à 4, cela signifie que l'arbre doit être éclaté avant d'ajouter l'élément. La méthode `EcR` est appelée pour effectuer cet éclatement. Si l'arbre est une feuille, la fonction crée une copie de l'arbre avec l'élément inséré de manière triée.

Dans le cas où l'arbre n'est pas une feuille, un nouvel arbre A est créé avec un nœud contenant les mêmes valeurs que l'arbre actuel. Ensuite, la position où l'élément x doit être inséré est déterminée à l'aide de la fonction `bisect_left`.

Ensuite, chaque sous-arbre de l'arbre actuel est parcouru. Pour chaque sous-arbre, une copie est créée sauf pour celui qui va être modifié. La méthode `AjoutSimple` est appelée récursivement sur le sous-arbre concerné. De plus si un éclatement a eu lieu, x est soit insérer dans le sous-arbre gauche (x est inférieur à l'élément du milieu) soit dans le sous-arbre droit (dans le cas contraire).

— $x <$ l'élément du milieu, cela indique que l'élément doit être inséré dans le sous-arbre gauche sans modifier le sous-arbre droit. Par conséquent, le sous-arbre droit est inséré en premier, car il ne nécessite aucune modification. Après l'insertion de x, le nouveau sous-arbre contenant cet élément doit être ajouté avant le sous-arbre droit existant. C'est pourquoi la méthode `insert` est utilisée pour insérer le nouveau sous-arbre à l'indice approprié dans la liste des sous-arbres de A. Cette approche assure que la propriété de tri de l'arbre est maintenue, car les sous-arbres sont organisés dans l'ordre correct en fonction de leurs contenus.

— $x >$ l'élément médian, cela signifie que l'élément doit être inséré dans le sous-arbre droit sans modifier le sous-arbre gauche. Ainsi, le sous-arbre gauche reste inchangé et est ajouté à l'arbre A. Lors de l'insertion de x, la méthode `append` est utilisée pour ajouter le sous-arbre modifié à la liste des sous-arbres de A. Cette approche garantit que la propriété de tri de l'arbre est toujours respectée, car les sous-arbres sont maintenus dans l'ordre correct en fonction de leurs contenus.

- Recherche :

Tout d'abord, si l'arbre est une feuille et que l'élément n'est pas présent, la méthode renvoie `None`, indiquant que l'élément recherché n'est pas présent dans l'arbre.

Ensuite, si l'élément x est présent dans le nœud courant, cela signifie que l'élément recherché est trouvé dans le nœud actuel, et donc l'arbre courant est retourné.

Dans le cas récursif, l'algorithme détermine dans quel sous-arbre poursuivre la recherche en comparant x aux clés du nœud courant.

- Suppression :

La suppression ne marche pas encore parfaitement, tous les cas où le nœud courant est une feuille (un commentaire a été rajouté dans le code) ont l'air de fonctionner comme prévu. Ensuite pour ce qui est du cas récursif, il existe 2 cas :

— x est présent dans le noeud, soit la clé la plus grande (max) du sous-arbre gauche est récupéré soit la clé la plus petit (min) du sous-arbre droit est récupéré, la clé récupéré remplace la clé à supprimer et un appel récursif est effectué sur min ou max.

— x n'est pas présent, l'appel récursif est effectué sur le bon sous-arbre.

Nous n'avons pas encore réussi à implémenter le cas où un des sous-arbres est de taille minimale, c'est-à-dire que nous devons emprunter une clé au sous-arbre d'à côté. Prenons le cas où le sous-arbre gauche est minimal, la manière dont nous voulions résoudre ce problème était de récupérer l'élément min du sous-arbre droit, appeler la méthode de suppression sur min pour l'effacer et la clé min remplace la clé courante, et ensuite nous faisons un ajout de la clé courante pour l'ajouter dans le sous-arbre gauche. Idem dans le cas où l'arbre droit est minimal. Et pour finir si les 2 sont minimaux une fusion est effectuée.

2.5 Arbre-B

Les Arbres-B sont des arbres de recherche avec des propriétés additionnelles et qui sont fortement utilisés dans les domaines manipulant des bases de données et des systèmes de gestion de fichiers. Un Arbre-B d'ordre $m > 0$ est défini tel que :

1. Les nœuds contiennent k clés, avec k tel que $m \leq k \leq 2m$.
2. La racine peut contenir entre 1 et $2m$ clés.
3. Toutes les feuilles sont au même niveau.

En pratique, les Arbres-B utilisent de la mémoire externe pour stocker l'information, par exemple sur un disque ou des pages web, à l'exception de la racine qui est en mémoire principale. Or, le temps d'accès à la mémoire secondaire est 10^5 fois supérieur au temps d'accès à la mémoire principale. Ainsi, une telle structure de données est intéressante lorsque m est assez grand afin que les nœuds stockent davantage de clés pour un arbre de hauteur plus faible, ce qui permet de réduire le nombre d'appels de recherche en mémoire externe.

Les opérations d'ajout, de recherche et suppression d'un Arbre-B sont similaires à un Arbre 2-3-4. Nous avons choisi de représenter les nœuds par des `PageB` qui contiennent k clés sous forme de liste, une liste de ses $k + 1$ enfants, ainsi que son parent afin de faciliter l'opération de l'éclatement que nous détaillerons par la suite. Nous pouvons remarquer qu'il serait possible par la suite de complexifier cette `PageB` en l'associant réellement de la mémoire externe afin de stocker les clés. Cependant, dans le cadre de ce projet, nous n'implémenterons pas une telle structure ; le choix étant librement délégué à notre professeur par la suite dans le cadre de son cours d'AlgAv.

- Ajout :

Une première caractéristique à noter est que l'ajout d'une clé se réalise au niveau des feuilles que nous appelons page externe, en référence à un nœud externe. Les clés étant triées, nous devons donc dans un premier temps vérifier si la clé est dans la page courante (l'arbre ne contient pas de doublon de clés dans cette représentation), ou bien s'il s'agit d'une page externe : nous pouvons alors effectuer l'ajout. Sinon, nous devons trouver parmi les enfants de la page courante quelle est

la page suivante dans laquelle nous pourrions ajouter la clé. La page suivante est déterminée en regardant la position d'insertion de la clé si elle était ajoutée à la page courante.

Un point crucial de l'ajout dans un Arbre-B est que nous devons respecter la limite sur le nombre de clés possible dans une page. La limite étant au plus $2k$ clés dans une page, un nombre pair, nous devons donc effectuer des éclatement en remontée, et non en descente comme dans l'Arbre 2-3-4 vu précédemment.

Pour savoir si nous devons éclater la page courante, nous vérifions d'abord si la page déborde, c'est-à-dire si la page comporte plus de clé que permis dans une page. L'éclatement en remontée d'une page consiste à diviser les clés de la page en deux puis remonter la clé médiane (le débordement implique un nombre impair de clés) dans la page parent. Les enfants de la page éclatée sont alors partagés entre les deux nouvelles pages.

Un cas particulier d'une telle opération est le cas de la page racine. Si la page éclatée est la racine, nous devons alors créer une nouvelle page racine contenant comme clé unique la clé médiane. Il n'y a donc que dans ce cas particulier que la hauteur de l'arbre augmente.

- Suppression :

La suppression pour un Arbre-B est en cours d'implémentation.

- Recherche :

L'algorithme de recherche est relativement simple : si l'arbre est non vide et que la clé recherchée n'est pas dans la page courante, nous cherchons dans la page suivante qui pourrait contenir la clé.

2.6 Arbre AVL

L'Arbre AVL (Adelson-Velsky et Landis) est une forme particulière d'ABR qui garantit un équilibrage automatique. Dans un Arbre AVL, la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit de chaque nœud, appelée facteur d'équilibre, est maintenue dans l'ensemble $\{1, 0, 1\}$, c'est-à-dire que les hauteurs des deux sous-arbres d'un même nœud diffèrent au plus de un, et cette propriété est conservée dynamiquement au moment de l'insertion ou de la délétion d'un nouveau nœud. Cela assure que l'arbre reste équilibré et prévient ainsi la dégénérescence de la structure, ce qui pourrait conduire à des temps d'accès inefficaces. Lorsqu'une opération d'ajout ou de suppression est effectuée, l'Arbre AVL est ajusté de manière à maintenir son équilibre. Ces ajustements peuvent impliquer des rotations simples ou doubles selon la situation spécifique. L'équilibrage constant garantit que la hauteur de l'arbre reste logarithmique par rapport au nombre de nœuds.

- Ajout : L'ajout dans un AVL commence de la même manière que dans un arbre binaire de recherche ordinaire. Cependant, après l'insertion d'un nœud, l'arbre peut devenir déséquilibré. Il faut alors rééquilibrer l'arbre en effectuant des rotations pour s'assurer qu'il conserve sa propriété d'équilibre. Il existe deux types de rotations : les rotations simples et les rotations

doubles. Ces rotations ajustent la structure de l'arbre tout en maintenant l'ordre de recherche.

- Recherche : La recherche dans un AVL suit le même principe que dans un arbre binaire de recherche classique. Nous commençons par comparer la valeur recherchée à la clé du nœud courant, puis nous nous déplaçons vers le sous-arbre gauche ou droit en fonction de la comparaison. Comme les AVL sont équilibrés, la recherche est généralement plus rapide que dans un arbre binaire de recherche non équilibré, car la hauteur maximale de l'arbre est maintenue à un niveau minimal.

- Suppression : La suppression dans un AVL commence également comme dans un arbre binaire de recherche ordinaire. Après avoir supprimé un nœud, l'arbre peut devenir déséquilibré. Comme pour l'ajout, des rotations peuvent être nécessaires pour rééquilibrer l'arbre. Ces rotations sont effectuées pour maintenir l'équilibre de l'arbre tout en préservant la propriété de recherche. Il peut y avoir plusieurs cas de déséquilibre à gérer lors de la suppression, et les rotations appropriées doivent être appliquées en fonction de la structure de l'arbre.

(exemple + un pseudo code + complexite)

2.7 Arbre Auto-adaptatif (Splay-tree)

Le Splay-tree est une structure d'ABR auto-ajustable, conçue pour optimiser l'accès aux éléments fréquemment consultés. Ce type d'arbre se distingue par sa caractéristique d'auto-ajustement, où les nœuds visités récemment sont déplacés vers la racine de l'arbre. Lors des opérations de recherche, d'insertion ou de suppression, la technique de « *splaying* » induit à une série de rotations pour amener le nœud cible à la racine. Cette approche a pour objectif de rapprocher les nœuds les plus souvent consultés vers le haut de l'arbre, améliorant ainsi les performances d'accès pour ces éléments spécifiques. Contrairement à des structures d'arbres comme l'AVL, le Splay-tree ne maintient pas une hauteur équilibrée de manière stricte.

- Ajout : L'ajout dans un Splay Tree est similaire à celui d'un arbre binaire de recherche ordinaire. Après l'insertion d'un nœud, le Splay Tree effectue une opération appelée « *splay* » pour réorganiser l'arbre. Le *splay* consiste à réorganiser l'arbre de telle manière que le nœud inséré devienne la racine de l'arbre. Cela peut nécessiter une série de rotations et d'inversions pour rééquilibrer l'arbre tout en conservant sa structure de recherche.

- Recherche : Lors de la recherche d'un élément dans un Splay Tree, nous effectuons également l'opération de *splay*. Après avoir trouvé l'élément recherché, on le *splay* pour le faire remonter à la racine de l'arbre. Cela signifie que les nœuds les plus souvent recherchés sont déplacés vers le haut de l'arbre, ce qui améliore les performances des futures recherches sur ces nœuds.

- Suppression : La suppression dans un Splay Tree suit exactement le même processus qu'un arbre binaire de recherche simple.

(exemple + un pseudo code + complexite)

2.8 Arbre Digital (DST)

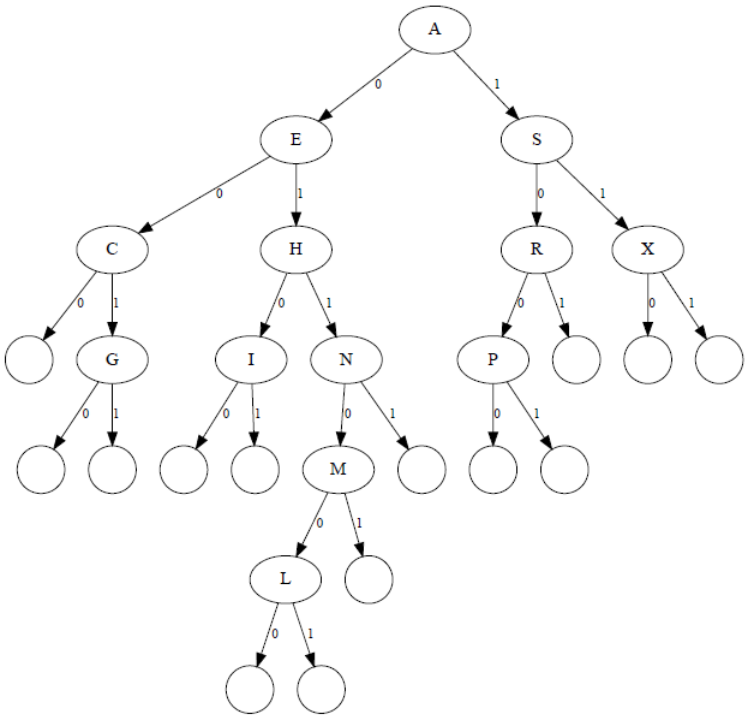


FIGURE 2 – Exemple d'arbre digital avec GraphViz

- Ajout :

L'ajout d'un caractère fait appel à une fonction auxiliaire `_ajout`. Cette méthode utilise la récursivité pour insérer un caractère et se base sur son encodage binaire. La méthode `_ajout`, parcours l'arbre binaire en comparant son caractère à insérer. Elle traite les cas suivants :

- Si l'arbre est vide, elle renvoie un arbre contenant le caractère à ajouter.
- Si le caractère est déjà présente, elle retourne le nœud courant.

Ensuite, elle détermine si le caractère doit être inséré à droite ou à gauche grâce à la méthode `car`. Durant son appel récursif dans le sous arbre approprié, la variable i augmente pour l'encodage binaire. Enfin, elle renvoie l'arbre courant si le caractère n'a pas été trouvé.

La complexité de cette méthode est en $O(\log n)$, avec n le nombre de nœuds. Elle parcourt l'arbre en suivant le chemin approprié pour l'insertion. Mais dans le cas où l'arbre est un arbre peigne droit/gauche, la complexité serait en $O(n)$ puisque le parcours se fait que d'un coté et donc

nous passerions par les n nœuds.

- car :

Cette méthode parcourt un dictionnaire qui associe chaque caractère à son encodage binaire sous forme de chaîne de caractères. Elle vérifie si la clé correspond au caractère c et si l'indice i est valide, afin de renvoyer le i-ème caractère de l'encodage sous forme d'entier. Sinon, elle renvoie `None`. La complexité est en $O(n)$.

- Recherche :

La recherche fait appel à une fonction auxiliaire `_recherche`. Cette méthode utilise la récursivité pour chercher un caractère tout en se basant sur son encodage binaire. La méthode `_recherche` commence par vérifier si le nœud courant est vide ou non puis son caractère. Ensuite, elle parcourt à droite ou à gauche en fonction de la méthode `car`. Enfin, elle renvoie `True` ou `False` si elle trouve le caractère recherché.

La complexité de cette méthode est en $O(\log n)$, avec n le nombre de nœuds. La recherche parcourt l'arbre en suivant un chemin unique. Cependant, si l'arbre est un arbre peigne, alors la complexité est en $O(n)$, puisqu'il faut parcourir les n nœuds.

- Suppression :

La méthode `supprime` appelle une méthode auxiliaire `_supprime` afin de retirer un caractère de l'arbre s'il existe. La méthode auxiliaire commence par vérifier si le nœud courant est vide. Ensuite, si le caractère recherché correspond à la clé du nœud, plusieurs cas sont traités :

— Si nous sommes sur la racine :
  – Si elle n'a pas de sous-arbre, alors elle renvoie `None`.
  – Si le nœud à supprimer est la racine, et qu'il a un seul sous-arbre, elle retourne une copie du sous-arbre existant.

— Pour les autres nœuds qui se trouvent hors de la racine :
  – Nous nous occupons d'abord des cas de base. Si les fils droit et gauche du nœud courant sont vide, alors nous renvoyons `None`.
  – Si l'un des deux fils est vide, alors nous renvoyons une copie du fils qui n'est pas vide.
  – Sinon, nous traitons le cas de la suppression, en utilisant une méthode `nœud_min`, pour récupérer le nœud le plus à gauche (le plus petit) du sous arbre droit. Nous le remplaçons par le nœud à supprimer et nous faisons un appel sur le nœud (le plus petit).

Puis, nous traitons le parcours dans l'arbre grâce à `car` afin d'avoir le parcours approprié.

La complexité de cette méthode est généralement logarithmique, en $O(\log n)$, avec n le nombre de nœuds. Elle parcourt l'arbre en suivant un chemin déterministe. Cependant, la complexité est en $O(n)$ si notre arbre est un arbre peigne.

2.9 Arbre Lexicographique (Trie binaire)

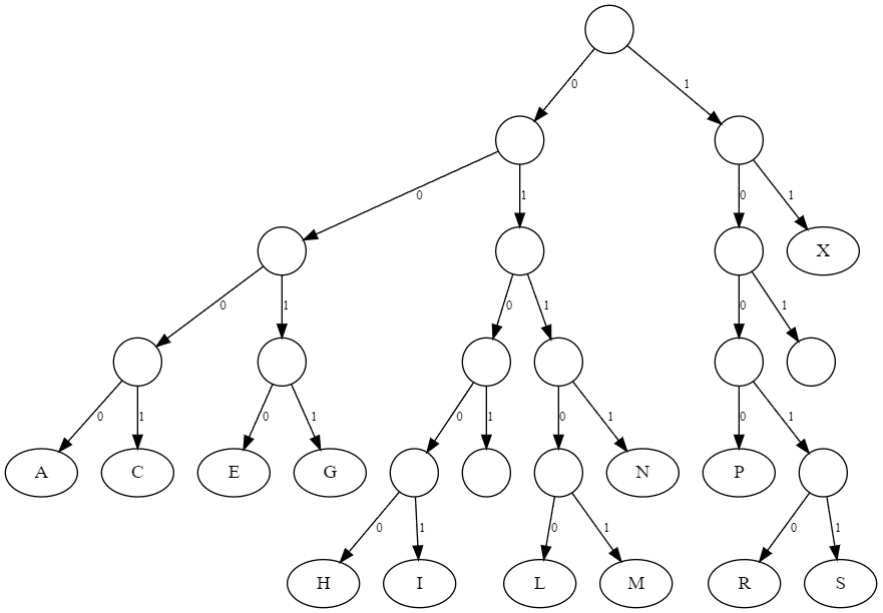


FIGURE 3 – Exemple d'un arbre lexicographique avec GraphViz

- Ajout :

La méthode `ajout` commence par appeler `_ajout`, qui effectue l'insertion récursivement. Si le nœud courant est `None`, alors un nœud contenant le caractère est créé. Ensuite, nous traitons le cas d'une feuille :

- Si la feuille contient la lettre que nous souhaitons ajouter, alors nous renvoyons simplement le nœud.
- Sinon, nous faisons appel à la méthode `split` en lui donnant les informations nécessaires : le caractère à ajouter, la clé du nœud courant et un indice i pour l'encodage.

Enfin, nous déterminons le parcours à gauche ou à droite à l'aide de la méthode `car`.

La complexité est en $O(\log n)$ et est due au fait que nous parcourons l'arbre en suivant un chemin déterministe. Mais la complexité pourrait être en $O(n)$ si l'arbre est un arbre peigne.

- split :

Cette méthode permet de diviser un nœud lorsque deux caractères avec le même préfixe sont ajoutés dans l'arbre. En entrée, nous avons $c1$ et $c2$ correspondant tous deux à un caractère, ainsi que l'indice i, pour l'encodage. La méthode compare les bits à l'indice i de $c1$ et $c2$ et traite plusieurs cas :

- Si la valeur de leur encodage sont identiques, alors nous faisons un appel récursif à `split` vers la droite ou la gauche en fonction de l'encodage.
- Si leur valeur d'encodage à l'indice i sont différentes, alors deux nœuds sont créés pour chacun des deux caractères.

La complexité est en $O(\log n)$ car la méthode parcourt le long d'une branche de l'arbre lors de l'insertion.

- Suppression :

La méthode utilise `_supprime`, sa méthode auxiliaire. La méthode auxiliaire fera une suppression à la remontée. Tout d'abord elle commence par les cas de base :

— Si le nœud est vide, elle renvoie `None`

— Si nous sommes sur le nœud contenant le caractère à supprimer :
  – Si le nœud ne contient aucun fils, nous renvoyons `None`.
  – S'il n'a pas de fils gauche, nous créons un nœud pour le sous-arbre droit que nous mettons dans une variable `res` qui sera utile pour la remontée, sans oublier de dupliquer les fils du sous arbre droit.
  – S'il n'a pas de fils droit, alors nous effectuons les mêmes actions que le cas du dessus.

Ensuite, nous parcourons l'arbre pour atteindre le caractère recherché. La méthode `car` permet de savoir dans quel sous-arbre continuer le parcours tout en incrémentant utile pour l'encodage. Durant le parcours, nous avons affecté nos résultats à la variable `res`.

Enfin, la remontée est gérée par `res` qui contient un nœud, pour s'assurer que l'arbre reste cohérent après la suppression. Plusieurs cas sont vérifiés à l'aide de `res` :

— Si le nœud `res` n'a pas d'enfant, alors nous renvoyons `None`.

— Si le nœud `res` a un seul enfant dans son fils gauche et pas de fils droit, alors nous renvoyons le fils gauche, afin de le remonter.

— Si le nœud `res` a un seul enfant dans son fils droit et pas de fils gauche, alors nous renvoyons le fils droit, afin de le remonter.

Finalement, nous retournons `res` qui est un nœud.

La complexité est en $O(\log n)$, où n est le nombre de nœuds dans l'arbre. La méthode parcourt l'arbre selon le mot à supprimer puis elle effectue une remontée pour la suppression. Dans le pire cas, celui d'un arbre peigne, la complexité est en $O(n)$.

- Recherche :

La recherche fait appel à une fonction auxiliaire `_recherche`. Cette méthode commence par vérifier les cas de base :

— Si le nœud courant vaut None, on renvoie `False`

— Si le nœud courant contient le caractère, on renvoie `True`

Puis, elle parcourt l'arbre par le sous arbre droit ou gauche en fonction de son encodage grâce a la variable i.

La complexité de cette méthode est en $O(\log n)$, avec n le nombre de nœud. Elle parcourt l'arbre en suivant un chemin déterministe. Cependant, si l'arbre est un arbre peigne, alors la complexité est en $O(n)$, puisqu'elle passe par tous les nœuds.

2.10 R-Trie

Cette structure est en cours d'implémentation.

2.11 Trie Hybride

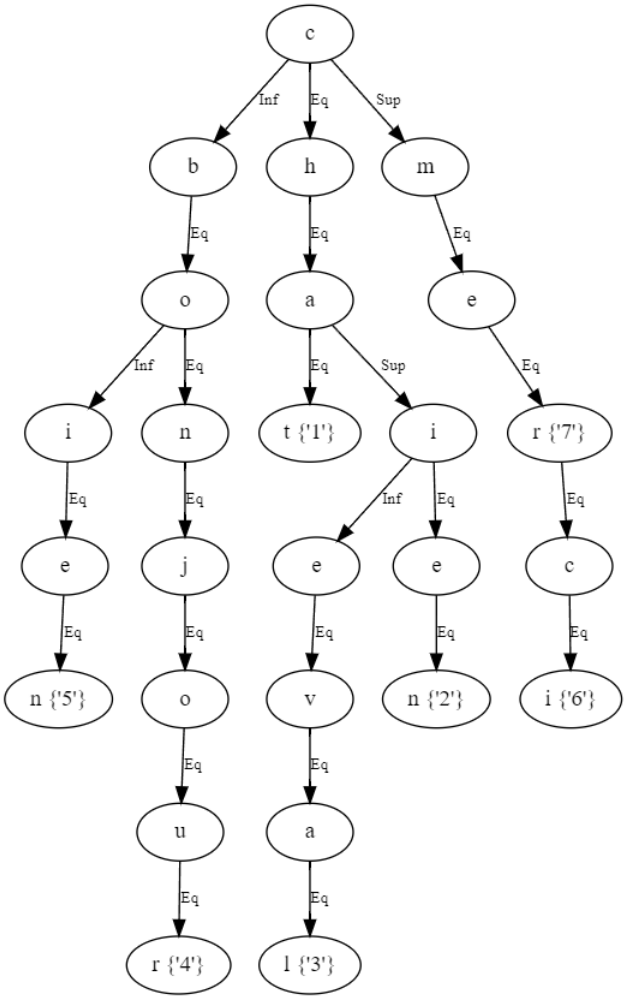


FIGURE 4 – Exemple d'un trie hybride avec GraphViz

- Ajout :

La méthode `ajout` est une fonction récursive et commence par les cas de base :

— Si l'arbre est vide :
  – Si le mot contient un seul caractère, nous renvoyons un nouveau nœud contenant le caractère et sa valeur pour indiquer la terminaison du mot.
  – Dans le cas où le mot contient plus d'un caractère, nous retournons un nouveau nœud contenant le premier caractère du mot, et le reste du mot est ajouté dans le sous arbre *eq* par un appel récursif à `ajout`.
— Si notre arbre n'est pas vide et le mot a une longueur de 1, alors nous renvoyons un nouveau nœud contenant le caractère ainsi que sa valeur. Puis nous effectuons une duplication à chaque sous-arbre de l'arbre pour le nouveau nœud.

Ensuite, si le mot contient plus d'un caractère, nous récupérons le premier caractère du mot dans une variable p, et plusieurs cas sont vérifiés :

— Si p vaut `None`, nous renvoyons simplement l'arbre.
— Si p est inférieur au caractère du nœud de l'arbre courant, le mot est inséré dans le sous-arbre `inf` du nœud courant.
— Si p est supérieur au caractère du nœud de l'arbre courant, le mot est inséré dans le sous arbre `sup` du nœud courant.
— Sinon, cela signifie que que mot est égal au caractère du nœud. La méthode est alors appelée récursivement sur le reste du mot dans le sous-arbre `eq` du nœud courant.

Dans le cas général où l'arbre est bien équilibré, la complexité est en $O(\log n)$, avec n le nombre de nœud dans l'arbre. Le parcours est déterministe donc il parcourt dans un sous-arbre à chaque fois. Mais la complexité peut être en $O(n)$ dans le cas où tous les mots se trouvent dans un seul sous-arbre de l'arbre.

3 Tâches réalisées et à terminer

La majorité des structures offertes par la bibliothèque ont été implémentées, à l'exception de l'Arbre 2-3-4, de l'Arbre-B et du R-Trie qui sont en partie implémentés. Par ailleurs, parmi toutes les structures, seule la File Binomiale ne peut pas encore être visualisée avec GraphViz et l'Arbre-B ne manipule pas encore des copies d'arbres.

Nous pouvons également remarquer que le Tas Min et la File Binomiale ont été reprises du projet d'AlgAv où nous avions eu l'occasion de les implémenter.

Au niveau de l'unification de la bibliothèque, certaines parties doivent être traduites de l'anglais au français pour une meilleure cohérence.

Conclusion

Références

[1] A. Genitrini, *Algorithmique Avancée* - UE `MU4IN500`, 2023-2024.

[2] R. Sedgewic, K. Wayne, *Algorithms 4th Edition*, Addison-Wesley, 2011.

[3] T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction à l'algorithmique*, Dunod, 2002

[4] N. Delestre, G. Del Mondo, *Les arbres-B* - INSA Rouen Normandie.