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, 8 mai 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 offertes par la bibliothèque, ainsi qu'un notebook pour chaque structure différente.

1 Présentation du projet

1.1 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.

1.2 Architecture de la bibliothèque

La bibliothèque est structurée en utilisant des notebooks Jupyter, avec un notebook principal réunissant toutes les structures disponibles, ainsi qu'un notebook distinct pour chaque structure. Chaque structure aura son propre fichier Python contenant le code spécifique (les définitions de classe et les méthodes associées), importé dans le notebook correspondant pour une meilleure lisibilité et modularité.

1.3 Graphviz

Graphviz est une bibliothèque utilisée en Python pour créer et visualiser des graphes. En utilisant des modules tels que graphviz ou pygraphviz, on peut générer des représentations visuelles de graphes à partir de descriptions de graphes DOT dans nos scripts Python. Dans notre projet, nous l'utilisons spécifiquement pour créer des visualisations des structures arborescentes, ce qui facilite la compréhension et l'analyse des relations et des hiérarchies présentes dans nos données grâce à des représentations graphiques claires et informatives.

1.4 Organisation et gestion du projet

Une des particularités de ce projet est le travail au sein d'une grande équipe confrontée à une charge de développement conséquente. Pour garantir une bonne progression, il était essentiel que chaque membre de l'équipe ait une vue d'ensemble sur le projet. De ce fait, nous avons dû prendre en compte les aspects suivants afin de mener à bien le projet : l'environnement de travail, l'organisation et la communication, sans oublier l'unification qui est l'une des thématiques du projet.

Ainsi, la première étape du projet a été pour nous de préparer l'environnement de travail, déterminer l'organisation, mettre au point la communication tout en prévoyant le maintien à long terme de cette dernière. Après avoir listé les structures et fonctions à implémenter, nous nous sommes répartis les tâches au fil de l'avancement du projet.

La collaboration dans un groupe de quatre personnes a été efficace grâce à une communication régulière sur Discord. Nous avons organisé des réunions pour discuter de l'avancement, des structures en cours d'implémentation pour résoudre les problèmes rencontrés et des décisions à prendre collectivement. Parfois, nous avons travaillé en binôme sur une structure, mais généralement, chacun était capable d'implémenter une structure seul, avec une vérification croisée du code par les autres membres de l'équipe.

Chaque membre de l'équipe a travaillé sur sa structure dans un notebook désigné, en commençant par l'implémentation de la structure elle-même avant de passer à l'unification avec le reste de la bibliothèque.

L'intégration de GraphViz était un aspect clé du cahier des charges. Nous l'avons donc intégré dès le début du projet pour visualiser notre progression lors du développement des structures. Étant donné notre inexpérience avec cette bibliothèque, nous avons dû nous former avant de démarrer le projet.

L'unification des structures a été une étape importante pour assurer la cohérence de la bibliothèque. Après avoir implémenté toutes les structures, nous avons convenu d'une norme pour les noms de fonctions et de variables, ainsi que d'une architecture commune pour le code. Notamment, nous avons choisi de coder en français plutôt qu'en anglais, car la bibliothèque est destinée à être utilisée par le professeur.

Enfin, au cours du projet, nous avons rencontré quelques difficultés récurrentes. D'une part, l'unification s'est avérée être un défi à relever en raison du grand nombre de structures à unifier et de la diversité des habitudes de programmation au sein de l'équipe, notamment en ce qui concerne le nommage des fonctions et variables. D'autre part, nous avons constaté que l'utilisation de GraphViz pour visualiser les structures pendant leur implémentation a été essentielle à la compréhension des opérations à écrire.

Une autre difficulté a résidé dans la gestion des duplications des structures à chaque opération, comme spécifié par le professeur, nous devions créer une copie de la structure de données pour éviter de modifier l'original. Cela a nécessité une attention particulière lors de la gestion des

pointeurs/références, pour garantir que les opérations se déroulent correctement sans altérer les données originales.

Pour finir, l'implémentation des diverses structures arborescentes a pu se montrer complexe en raison des définitions de structure, pouvant nécessiter la manipulation de parent, d'enfants et de frères de noeuds. De plus, nous avons dû porter une attention particulière à tous les cas potentiels lors des opérations d'ajout et de suppression, notamment pour les structures Arbre 2-3-4, Arbre-B et R-Trie.

2 Présentation des structures

Parmi les structures implémentées, le Tas Min, la File Binomiale et l'Arbre Binaire de Recherche ont été repris et modifiés du projet d'AlgAv afin d'intégrer GraphViz et les attentes du cahier des charges.

Dans la description de chaque structure, nous présentons ses fonctions principales, en sélectionnant une particulièrement intéressante que nous détaillons à l'aide de pseudo-code.

2.1 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 $\left\lfloor \frac{(i-1)}{2} \right\rfloor$.

Algorithme 1 : ajoutTasMin

Entrées : tas un tas min, cle la clé à ajouter dans tas
Sorties : Le tas min après avoir ajouté clé

nouveauTas ← `TasMin()`
nouveauTas.tas ← dupliquer les éléments de tas
nouveauTas.tas ← nouveauTas.`_ajoutTasMin`(cle)

retourner nouveauTas

Algorithme 2 : _ajoutTasMin

Entrées : tas un tas min, cle la clé à ajouter dans tas
Sorties : Le tableau contenant les éléments du tas min après avoir ajouté clé

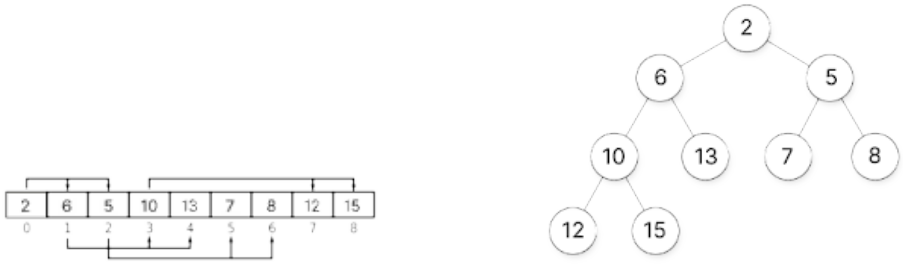
tas.tas ← ajouter cle dans le tableau contenant les éléments de tas
$i \leftarrow$ taille de tas.tas-1

tant que $i \neq 0$ et tas.tas[i] < tas.tas[tas.`_parent`(i)] **faire**
    (tas.tas[i], tas.tas[tas.`_parent`(i)]) ← (tas.tas[tas.`_parent`(i)], tas.tas[i])
    $i \leftarrow$ tas.`_parent`(i)

retourner tas.tas

Nous pouvons remarquer que tas.`_parent`(i) récupère l'indice du parent de l'élément en position i dans le tas, soit $\left\lfloor \frac{(i-1)}{2} \right\rfloor$.

- 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

2.2 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 le degré 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 :
  L'ajout d'une clé dans une File Binomiale repose sur l'algorithme de l'union de Files Binomiales. Ainsi, la File résultante de l'ajout est l'union de la File dans laquelle nous souhaitons faire l'ajout et la File contenant l'unique Arbre Binomial ayant pour unique clé la clé à ajouter.

- Suppression (extraction de la clé minimale) :
  La suppression de la clé minimale d'une File Binomiale prend en compte le fait que les éléments de la File sont des Arbres Binomiaux dont les clés sont ordonnées de façon croissante. Ainsi, chaque racine de chaque Arbre Binomial dans la File est le plus petit élément dans chaque arbre.
  Supprimer la clé minimale de la File revient donc à comparer chaque racine des arbres dans la File, retirer l'arbre contenant la clé minimale (appelé arbreMin) et enfin retourner l'union de la file sans l'arbreMin et de la file contenant les sous-arbres de l'arbreMin (c'est-à-dire que nous retirons la racine de l'arbreMin et nous créons une file contenant tous les sous-arbres de l'arbreMin).

- Union de deux files :
  L'union de deux files consiste à interclasser les files en partant des arbres binomiaux de degré minimum. Nous gardons en particulier un arbre binomial en retenue de l'appel précédent de l'algorithme de l'union.

  Ainsi, si l'arbre en retenue est vide, nous récupérons les deux arbres de degré minimum dans les deux files à l'aide de `minDeg`. Puis nous comparons les degrés entre-eux. Nous récupérons alors le reste de la file possédant l'arbre de degré minimum et nous faisons l'union entre ce reste. La file résultante est celle obtenue en ajoutant à l'union l'arbre de degré minimum avec `ajoutMin` (cette fonction ajoute à une file son arbre de degré de minimum). Dans le cas où les arbres comparés sont de même degré k, nous calculons un arbre binomial de degré $k+1$ obtenu de l'union de ces deux arbres puis nous appliquons l'algorithme de l'union aux files privées de leur arbre de degré minimal et ce nouvel arbre comme retenue.
  Le cas où l'arbre en retenue est non vide est similaire à la description du dessus : nous comparons les degrés entre la retenue et les arbres de degré minimal des deux files et la retenue suivante est l'arbre de plus petit degré. Deux arbres de même degré k forment un arbre de degré $k+1$. Enfin, si les trois arbres sont de même degré, nous retenons l'union de deux arbres et en gardons un pour l'ajouter à la file résultante de l'union entre les restes des deux files.

Algorithme 3 : unionFile

Entrées : file1 et file2 deux Files Binomiales
Sorties : L'union de file1 et file2 une File Binomiale

nouvelleFile ← `FileBinomiale`()
nouvelleFile.arbres ← copier les arbres binomiaux de file1
resultat ← `_unionFile`(nouvelleFile, file2, `ArbreBinomial`())
nouvelleFile.arbres ← copier les arbres binomiaux de resultat

retourner nouvelleFile

Algorithme 4 : _unionFile

Entrées : file1 et file2 deux Files Binomiales, arbre un Arbre Binomial
Sorties : L'union de file1 et file2 une File Binomiale

```
si arbre.estVide() alors
    si file1.estVide() alors
        retourner file2
    si file2.estVide() alors
        retourner file1
    arbre1 = file1.minDeg()
    arbre2 = file2.minDeg()
    si arbre1.degre < arbre2.degre alors
        retourner _ajoutMin(unionFile(file1._reste(), file2), arbre1)
    si arbre2.degre < arbre1.degre alors
        retourner _ajoutMin(unionFile(file1, file2._reste()), arbre2)
    si arbre1.degre == arbre2.degre alors
        retourner _unionFile(file1._reste(), file2._reste(),
                             union_arbre(arbre1, arbre2))
sinon
    si file1.estVide() alors
        retourner unionFile(file2, arbre.transformation_file())
    si file2.estVide() alors
        retourner unionFile(file1, arbre.transformation_file()
    arbre1 = file1.minDeg()
    arbre2 = file2.minDeg()
    si arbre.degre < arbre1.degre and arbre.degre < arbre2.degre) alors
        retourner _ajoutMin(unionFile(file1, file2), arbre)
    si arbre.degre == arbre1.degre and arbre.degre == arbre2.degre) alors
        retourner _ajoutMin(_unionFile(file1._reste(), file2._reste(),
                                union_arbre(arbre1, arbre2)), arbre)
    si arbre.degre == arbre1.degre and arbre.degre < arbre2.degre) alors
        retourner _unionFile(file1._reste(), file2, union_arbre(arbre1, arbre))
    si arbre.degre == arbre2.degre and arbre.degre < arbre1.degre) alors
        retourner _unionFile(file1, file2._reste(), union_arbre(arbre2, arbre))
retourner nouvelleFile
```

2.3 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.

  Pour ce qui est de la complexité, dans le pire cas, l'ABR est de la forme d'un arbre peigne, soit similaire à une liste chaînée. Ainsi, dans le cas d'un ajout d'une clé au niveau des feuilles, nous devons parcourir les n noeuds de l'ABR. Le pire cas est donc d'une complexité en $O(n)$.

Algorithme 5 : ajoutABR

Entrées : A : un Arbre Binaire de Recherche, cle : la clé à ajouter
Sorties : L'ABR après l'ajout de cle

```
abr ← ABR()
abr.racine ← A._ajoutABR(A.racine, cle)
retourner abr
```

Algorithme 6 : _ajoutABR

Entrées : A : un Arbre Binaire de Recherche, noeud : le noeud courant qui est un Arbre Binaire, cle : la clé à ajouter
Sorties : nouveauNoeud : le noeud courant après l'ajout de cle

```
si noeud == None alors
    retourner ArbreBinaire(cle)
sinon si cle == noeud.cle alors
    retourner ArbreBinaire(noeud.cle, noeud.gauche, noeud.droite, noeud.hauteur)
sinon si cle < noeud.cle alors
    nouveauGauche ← A._ajoutABR(noeud.gauche, cle)
    nouveauNoeud ← ArbreBinaire(noeud.cle, nouveauGauche, noeud.droite)
sinon
    nouveauDroite ← A._ajoutABR(noeud.droite, cle)
    nouveauNoeud ← ArbreBinaire(noeud.cle, noeud.gauche, nouveauDroite)
retourner nouveauNoeud
```

- 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.

2.4 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.

Algorithme 7 : ajoutAVL

Entrées : un arbre AVL A, la clé à ajouter
Sorties : L'Arbre AVL A après ajout d'une clé c

racine ← racine de A
retourner A.**_ajoutAVL(**A**, racine,** c**)**

Algorithme 8 : _ajoutAVL

Entrées : un arbre AVL A, noeud, cle
Sorties : L'Arbre AVL A après ajout d'une clé

si noeud est vide **alors**
  retourner Nouvel ArbreBinaire avec la clé
sinon si cle est égal à la clé de noeud **alors**
  retourner Nouvel ArbreBinaire avec la clé de noeud, les sous-arbres gauche et droit inchangés, et la hauteur de noeud
sinon si cle est inférieur à la clé de noeud **alors**
  nouveauGauche ← A.**_ajout**(noeud.gauche, cle)
  nouveauNoeud ← Nouvel ArbreBinaire avec la clé de noeud, nouveauGauche comme sous-arbre gauche, noeud.droite comme sous-arbre droit
sinon
  nouveauDroite ← A.**_ajout**(noeud.droite, cle)
  nouveauNoeud ← Nouvel ArbreBinaire avec la clé de noeud, noeud.gauche comme sous-arbre gauche, nouveauDroite comme sous-arbre droit
nouveauNoeud.hauteur ← 1 + max(hauteur de nouveauNoeud.gauche, hauteur de nouveauNoeud.droite)
retourner A.**_equilibrage**(nouveauNoeud, cle)

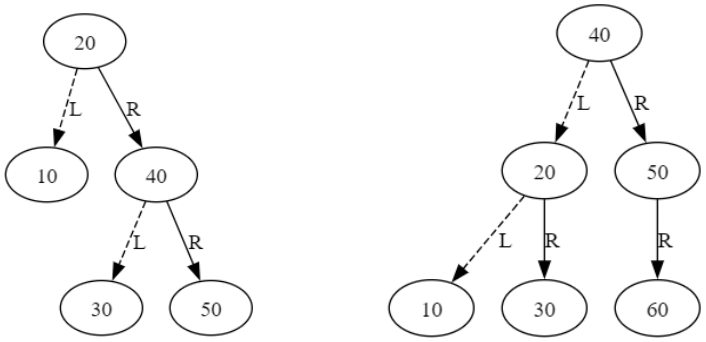


Figure 2 – Exemple d'arbre AVL avant et après ajout de la clé 60

Algorithme 9 : _equilibrage

Entrées : un arbre AVL A, noeud, cle
Sorties : Arbre équilibré après l'ajout d'une clé

```
equilibrage ← récupère le facteur d'équilibre du noeud
si equilibrage > 1 alors
    si cle < cle de noeud.gauche alors
        retourner A._rotation_droite(noeud)
    sinon
        retourner A._rotation_gauche_droite(noeud)
sinon si equilibrage < -1 alors
    si cle > cle de noeud.droite alors
        retourner A._rotation_gauche(noeud)
    sinon
        retourner A._rotation_droite_gauche(noeud)
retourner noeud
```

Une rotation est une opération qui réorganise les références des sous-arbres et met à jour les hauteurs appropriées afin de maintenir l'équilibre de l'arbre. Les pseudo-codes ci-dessous décrivent les opérations de rotation gauche et de rotation gauche-droite. Les opérations de rotation dans l'autre sens (rotation droite et rotation droite-gauche) utilisent le même code ; il suffit de remplacer les références aux nœuds gauche par droite et vice versa.

Algorithme 10 : rotation_gauche

Entrées : noeud
Sorties : Arbre après rotation gauche

```
y ← noeud.droite
T2 ← y.gauche
y.gauche ← noeud
noeud.droite ← T2
Mettre à jour les hauteurs
retourner y
```

Algorithme 11 : rotation_gauche_droite

Entrées : un arbre AVL A, noeud
Sorties : Arbre après rotation gauche-droite

```
noeud.gauche ← A._rotation_gauche(noeud.gauche)
retourner A._rotation_droite(noeud)
```

- 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.

La complexité d'un arbre AVL équilibré repose sur sa hauteur, qui reste toujours en $O(\log n)$ même dans le pire des cas, où n représente le nombre de nœuds. Grâce aux rotations appliquées lors des opérations, un arbre AVL préserve son équilibre et sa hauteur. Par conséquent, tant pour l'ajout, la suppression que la recherche, la complexité est en $O(\log n)$.

2.5 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.

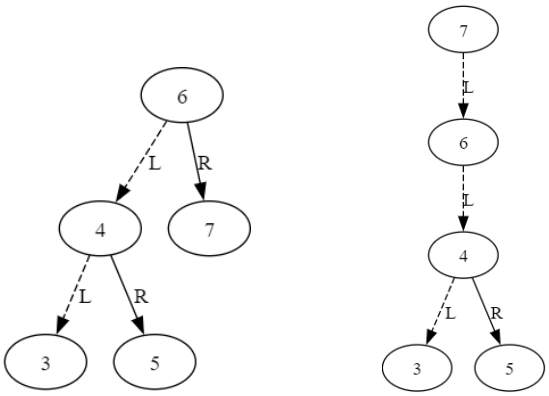


FIGURE 3 – Exemple d'arbre Splay avant et après une opération de recherche sur 7

- 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.

Algorithme 12 : recherche

Entrées : un arbre splay A, cle : entier
Sorties : (bool, ArbreSplay)

splay ← créer un nouvel arbre dupliqué de A
splay.racine ← _recherche(racine, cle)
exist ← splay.racine n'est pas None et splay.racine.cle == cle
retourner (exist, splay)

Algorithme 13 : _recherche

Entrées : un arbre splay A, noeud de l'arbre, cle : entier
Sorties : ArbreBinaire

si noeud est None ou cle == noeud.cle **alors**
  retourner noeud
gauche ← noeud.gauche
droite ← noeud.droite
nouveauNoeud ← ArbreBinaire(noeud.cle, gauche, droite)
retourner _splay(nouveauNoeud, cle)

Algorithme 14 : _rotation_gauche

Entrées : un arbre splay A, noeud
Sorties : ArbreBinaire

y ← noeud.droite
noeud.droite ← y.gauche
y.gauche ← noeud
retourner y

Algorithme 15 : splay

Entrées : un arbre splay A, noeud , cle : entier
Sorties : ArbreBinaire

```
si noeud est None ou noeud.cle == cle alors
    retourner noeud
sinon si cle < noeud.cle alors
    si noeud.gauche est None alors
        retourner noeud
    si cle < noeud.gauche.cle alors
        noeud.gauche.gauche ← splay(noeud.gauche.gauche, cle)
        noeud ← _rotation_droite(noeud)
    sinon
        noeud.gauche.droite ← splay(noeud.gauche.droite, cle)
        si noeud.gauche.droite n'est pas None alors
            noeud.gauche ← _rotation_gauche(noeud.gauche)
    si noeud.gauche n'est pas None alors
        retourner _rotation_droite(noeud)
    sinon
        retourner noeud
sinon
    si noeud.droite est None alors
        retourner noeud
    si cle < noeud.droite.cle alors
        noeud.droite.gauche ← splay(noeud.droite.gauche, cle)
        si noeud.droite.gauche n'est pas None alors
            noeud.droite ← _rotation_droite(noeud.droite)
    sinon
        noeud.droite.droite ← splay(noeud.droite.droite, cle)
        noeud ← _rotation_gauche(noeud)
    si noeud.droite n'est pas None alors
        retourner _rotation_gauche(noeud)
    sinon
        retourner noeud
```

- Suppression :
  La suppression dans un Splay Tree suit exactement le même processus qu'un arbre binaire de recherche simple. La complexité dans le pire cas des opérations sur un arbre splay est généralement de $O(n)$, où n est le nombre d'éléments dans l'arbre. Cependant, bien que chaque opération de splay puisse potentiellement avoir une complexité linéaire dans le pire cas, l'utilisation répétée et le rééquilibrage dynamique de l'arbre lors des opérations contribuent à maintenir la hauteur de l'arbre relativement basse dans des scénarios d'utilisation réalistes. Cela se traduit par une complexité en moyenne bien meilleure, généralement de $O(\log n)$ pour les opérations d'ajout, de recherche et de suppression.

2.6 Arbre 2-3-4

Un arbre 2-3-4, également connu sous le nom d'arbre B d'ordre 2, est une structure de données couramment utilisée dans diverses applications informatiques pour sa capacité à maintenir un équilibre optimal. Chaque nœud de cet arbre peut comporter entre 2 et 4 enfants, et il peut stocker plusieurs valeurs en fonction de son nombre d'enfants. Cette caractéristique garantit une répartition équilibrée des données, limitant ainsi la profondeur maximale de l'arbre et assurant des performances cohérentes pour les opérations telles que l'insertion, la suppression et la recherche.

En raison de sa nature équilibrée, l'arbre 2-3-4 offre des performances stables même lors de manipulations intensives des données. Bien que les opérations d'insertion et de suppression puissent parfois nécessiter la réorganisation des nœuds, leur complexité reste logarithmique dans le pire des cas. Cette efficacité en fait un choix populaire pour les bases de données, les systèmes de fichiers et d'autres applications où la gestion efficace des données est essentielle.

- Ajout :
  La fonction d'ajout commence par vérifier si l'élément à ajouter est déjà présent dans l'arbre. Si c'est le cas, elle renvoie une copie de l'arbre sans effectuer d'ajout, garantissant ainsi que les éléments dans l'arbre restent uniques. Pour éviter les débordements dans les nœuds, la méthode utilise l'éclatement à la descente, qui divise les nœuds remplis en deux nouveaux nœuds lorsque nécessaire. Cependant, cette stratégie est parfois excessive car elle est appelée même lorsque l'éclatement n'est pas nécessaire. Cela peut entraîner la création d'un arbre avec une hauteur supplémentaire et beaucoup de nœuds relativement vides, chaque nœud ne contenant qu'un seul élément. Cela permet de maintenir la structure et les propriétés de l'arbre 2-3-4 tout en ajoutant de nouveaux éléments. Ensuite, elle gère l'insertion de l'élément en prenant en compte les règles spécifiques de l'arbre 2-3-4 : les insertions ne peuvent se faire qu'aux feuilles.

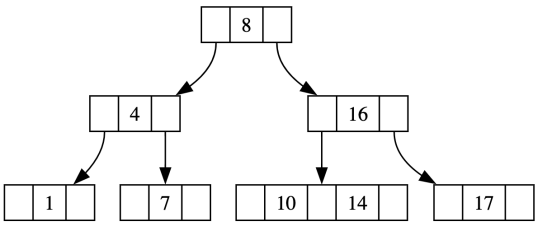


FIGURE 4 – Exemple d'Arbre 2-3-4 après éclatement lors de l'insertion de l'élément 10

- Recherche :
  La fonction de recherche suit une approche simple : elle vérifie d'abord si la clé x est présente dans le nœud courant. Si ce n'est pas le cas, elle détermine le fils dans lequel poursuivre la récursion. Cependant, si le nœud courant est une feuille et que la clé n'est pas trouvée, cela signifie que la clé recherchée n'existe pas dans l'arbre.

- Suppression :
  Quand c'est une feuille, la fonction de suppression retire l'élément spécifié, à condition que le degré du noeud soit supérieur à 2. Si ce n'est pas le cas, le noeud courant tente d'emprunter à ses voisins lorsque cela est possible. Si l'emprunt s'avère impossible, une fusion avec l'un des voisins est nécessaire pour maintenir l'équilibre de l'arbre.

Lorsque l'élément à supprimer se trouve dans un noeud interne, l'élément identifié est remplacé par la clé maximale du fils gauche, à condition qu'une clé puisse y être remontée ; sinon, la clé minimale du fils droit est utilisée. Suite à ce remplacement, une récursion est effectuée pour supprimer la clé maximale ou minimale qui a remplacé l'élément initial. Si aucune de ces conditions ne permet une action, une fusion des fils concernés devient nécessaire.

Dans les cas où l'élément à supprimer n'est pas trouvé directement, la suppression doit toujours respecter les propriétés structurelles d'un arbre 2-3-4. Si le noeud ciblé pour la récursion suivante est minimal, il doit emprunter une clé d'un voisin non minimal pour permettre la continuation de la récursion sans compromettre la structure de l'arbre.

Algorithme 16 : supprime

Entrées : noeud : noeud courant, x : élément à supprimer
Sorties : l'arbre après suppression de x

```
si estFeuille(noeud) alors
    si contient(noeud, x) alors
        si degre(noeud) > 2 alors
            retirer(noeud, x)
        sinon
            VG ← voisinGauche(noeud)
            VD ← voisinDroit(noeud)
            si estVide(VG) alors
                si estVide(VD) alors
                    supprimer l'arbre car le seul élément dans l'arbre est x
                sinon
                    si degre(VD) > 2 alors
                        emprunt(noeud, VD, cleMinimale(VD))
                    sinon
                        fusion(noeud, VD)
            sinon
                si degre(VG) > 2 alors
                    emprunt(noeud, VG, cleMaximale(VG))
                sinon
                    si estVide(VD) alors
                        fusion(VG, noeud)
                    sinon
                        si degre(VD) > 2 alors
                            emprunt(noeud, VD, cleMinimale(VD))
                        sinon
                            fusion(noeud, VG)
    sinon
        Ne rien changer car x n'est pas présent dans l'arbre
```

```
si not estFeuille(noeud) alors
    si contient(noeud, x) alors
        indice ← index(contenu(noeud), x)
        FG ← sousArbres(noeud)[indice]
        FD ← sousArbres(noeud)[indice + 1]
        si not estArbreMinimal(FG) alors
            contenu(noeud)[indice] ← cleMaximale(FG)
            supprime(sousArbres(noeud)[indice], cleMaximale(FG))
        sinon
            si not estArbreMinimal(FD) alors
                contenu(noeud)[indice] ← cleMinimale(FD)
                supprime(sousArbres(noeud)[indice + 1], cleMinimale(FD))
            sinon
                fusion(FG, FD)
                supprime(sousArbres(noeud)[indice], x)
    sinon
        indice ← index(contenu(noeud), x)
        recursionSuivante ← sousArbres(noeud)[indice]
        si estArbreMinimal(recursionSuivante) et not estFeuille(recursionSuivante) alors
            indiceArbreNonMinimal ← indexNonMinimal(contenu(noeud))
            si indiceArbreNonMinimal > indice alors
                cle ← cleMinimale(indiceArbreNonMinimal)
            sinon
                cle ← cleMaximale(indiceArbreNonMinimal)
            emprunt(recursionSuivante, sousArbres(noeud)[indiceArbreNonMinmal], cle)
        supprime(recursionSuivante, x)
```

Dans notre présentation du pseudo-code, nous avons choisi de nous concentrer sur les éléments essentiels pour conserver la clarté et la concision, étant donné que le code source complet est assez volumineux. Ainsi, certaines opérations complexes, telles que la récursion par copie, ont été omises pour simplifier la lecture. De même, la fonction fusion a été abrégée en éliminant les paramètres initialement requis pour spécifier l'indice de l'élément gauche qui devait servir d'élément médian après la fusion

Dans ce pseudo-code, VG et VD désignent respectivement les voisins gauche et droit du noeud, et FG et FD représentent les sous-arbres gauche et droit. Ces derniers correspondent aux structures fils situées de part et d'autre de l'élément trouvé. Cette représentation aide à comprendre visuellement comment les opérations d'emprunt et de fusion sont appliquées en fonction de la position de l'élément concerné

Quand le noeud est une feuille, les opérations principales comme emprunt et fusion ont une complexité de $O(1)$. Les autres opérations primitives utilisées dans ce contexte sont également en $O(1)$. Par conséquent, dans le cas où le noeud est une feuille, l'ensemble des opérations effectuées est en $O(1)$.

Par contre si le noeud n'est pas une feuille, il existe 2 cas, soit x est présent dans le noeud, soit il n'est pas présent :

— Lorsque x est présent dans un noeud interne, l'opération estArbreMinimal est la plus coûteuse et s'exécute en $O(n)$, étant donné que qu'elle parcours tout l'arbre passé en paramètre. Les fonctions cleMinimale et cleMaximale, quant à elles, s'exécutent en $O(\log n)$ car il suffit de trouver la clé la plus la plus profonde à gauche et à droite respectivement. La combinaison de ces deux types d'opérations donne une complexité totale de $O(n + \log n)$, qui est simplifiée en $O(n)$. À noter que cette partie du code n'est appelé qu'une seule fois quand x est dans un noeud interne par la suite x sera remplacé par la clé minimale ou maximale qui se trouve aux feuilles.

— Ensuite pour le cas où x n'est pas présent dans le noeud, le pire cas serait que le prochain noeud pour la récursion soit minimal et aura une complexité en $O(n)$ comme en haut mais ce cas n'arrive qu'une fois, puisqu'après l'emprunt, cette condition sera toujours fausse. Ainsi il serait raisonnable de supposer que la seule opération appelé dans cette condition est estArbreMinimal qui est en $O(n)$

La fonction supprime est invoquée $\log n$ fois, ce qui suggérerait initialement une complexité de $O(n \log n)$. Toutefois, la méthode estArbreMinimal est appelée à chaque niveau de profondeur du noeud, à l'exception des cas où le noeud courant est une feuille. Étant donné que l'arbre est réduit à chaque appel récursif, on pourrait envisager que la complexité de la méthode supprime que nous avons implémentée est de $O(\log n^2)$. Cette complexité étant moins efficace que la complexité optimale de $O(\log n)$, attendue pour les algorithmes de suppression dans un arbre 2-3-4.

2.7 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.

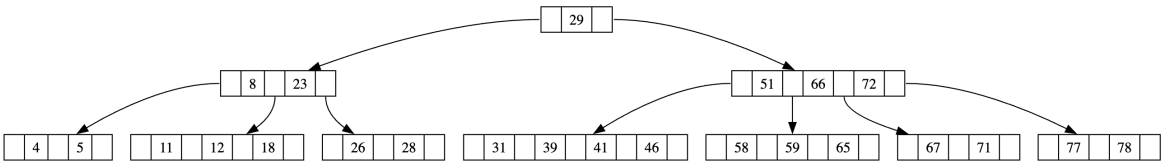


FIGURE 5 – Exemple d'Arbre-B d'ordre 2 avec GraphViz

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, comme nous pouvons l'observer sur l'image ci-dessous. Cette position d'insertion est alors l'indice de l'enfant dans lequel appliquer l'algorithme de l'ajout.

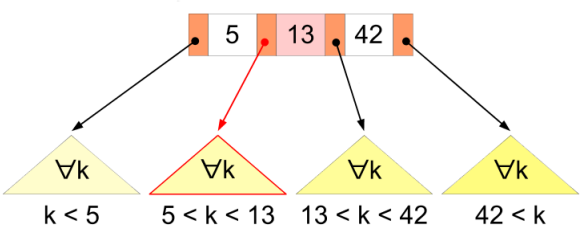


FIGURE 6 – Exemple d'Arbre-B généralisant les valeurs des clés dans les enfants d'un noeud

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 clés 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.

- 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.

- Suppression :
  Tout comme l'ajout, la suppression d'une clé s'effectue au niveau des feuilles et implique une vérification des propriétés de l'Arbre-B, ainsi que des opérations supplémentaires pour respecter ces propriétés. Pour la suppression, il s'agit du nombre de clés minimales autorisé, c'est-à-dire 1 pour la racine et m pour un nœud non racine.

  Lors de la suppression d'une clé, nous distinguons deux cas : soit la suppression s'effectue au niveau d'une feuille, soit elle a lieu au niveau d'un nœud interne.
  Si nous avons détecté la clé à supprimer, le premier cas est trivial à traiter tandis que le second cas nécessite plus d'éléments à prendre en considération. Tout d'abord, nous devons récupérer une clé de remplacement pour prendre la place de celle à supprimer. Pour respecter l'ordre des clés dans l'arbre, nous choisissons la clé prédécesseur de la clé à supprimer, soit la plus grande clé du sous-arbre gauche. Puis, après avoir échangé ces deux clés, nous appliquons l'algorithme de suppression sur la clé prédécesseur et le sous-arbre gauche.
  Sinon, la clé ne se trouve pas dans la page courante. Si la page courante est une feuille, la clé à supprimer ne se trouve pas dans l'arbre. Sinon, nous continuons à chercher la clé à supprimer dans la page suivante parmi les enfants de la page courante (que nous trouvons comme pour l'ajout).

  Enfin, à la fin d'une suppression, nous devons vérifier les propriétés de l'arbre : après avoir retiré une clé d'un noeud, il est possible de se trouver face à un déficit de clés. Pour combler un tel déficit, deux méthodes sont possibles : l'emprunt de clé ou la fusion avec une autre page.
  L'emprunt de clé est possible si un des frère existe et si cet emprunt n'entraîne pas lui-même de déficit de clés (une page n'a pas de frère gauche ou droite s'il s'agit du noeud à l'extrémité gauche ou droite de l'arbre). Suite à un emprunt, la page courante doit également emprunter un enfant à son frère puisqu'un noeud à k clés possède $k + 1$ enfants. Nous remarquons qu'une page ne peut pas directement récupérer la clé de son frère car cela fausserait l'ordre des clés dans l'arbre. Pour cela, son parent échange la clé empruntée avec une de ses clés. Nous pouvons voir cela comme le frère remonte une clé au parent et le parent descend une clé dans la page courante.

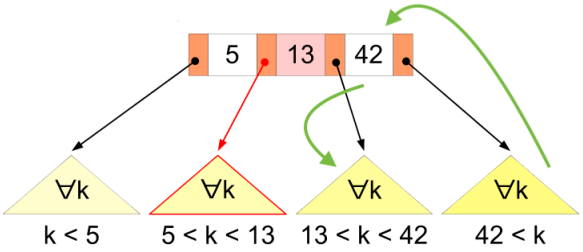


FIGURE 7 – Exemple d'Arbre-B montrant les échanges de clés (flèches vertes) entre le parent, le frère (4e enfant de son parent) et la page courante (3e enfant) lors d'un emprunt de clé

D'après la figure ci-dessus, nous pouvons voir que la page courante souhaite emprunter une clé à son frère droit. Son frère remonte une clé à son parent (pour respecter l'ordre des clés, il échange sa plus petite clé, inversement la plus grande clé s'il s'agissait du frère gauche) et le parent descend

une clé dans la page courante (ici la clé 42).

Sinon, si la page courante ne peut pas emprunter de clé, elle doit alors fusionner avec un de ses frères (son frère gauche s'il existe, le droit dans le cas échéant). Un point à noter est que lors d'une fusion, afin de respecter l'ordre des clés dans l'arbre, le parent doit céder une clé qui servira de clé médiane au nouveau noeud résultant de la fusion. Ce point permet également au parent de respecter k clés et $k+1$ enfants (après fusion, il possède donc k enfants pour $k-1$ clés).

Un autre cas à considérer est le cas de la racine. La racine peut se trouver sans clé suite à une fusion de ses enfants si elle ne possédait qu'une unique clé avant la fusion. Dans ce cas, le nouvel enfant résultant de la fusion devient la nouvelle racine : il s'agit du cas de la diminution de la hauteur de l'arbre.

Pour finir, nous remarquons que suite à une fusion, le parent peut lui-même être atteint d'un déficit de clé puisqu'il doit céder une clé. Ainsi, dans le pire cas, nous supprimons une clé au niveau des feuilles et chaque noeud possède le nombre minimal de clés possible, ce qui implique après la suppression une remontée jusqu'à la racine pour compenser le déficit par fusion. Donc la complexité de cet algorithme est en $O(\log n)$, soit la hauteur de l'arbre.

Algorithme 17 : suppressionBtree

Entrées : A un Arbre-B, cle la clé à supprimer
Sorties : L'Arbre-B A après suppression de la clé cle

si A est non vide **alors**
  $(A.\text{racine}, \_) \leftarrow A.$`_suppressionBtree`$(A.\text{racine}, \text{None}, \text{cle}, -1)$
retourner A

Algorithme 18 : _suppressionBtree

Entrées : A : un Arbre-B, page : la page courante, parent : le parent de page, cle : la clé à supprimer, indicePage : l'indice de la page courante parmi les enfants de son parent

Sorties : Le couple (nouvellePage, nouvelIndicePage) correspondant à la page courante après suppression de cle et son nouvel indice parmi les enfants de son parent

```
si page._recherche_page(cle) alors
    si page est la racine et ne contient qu'une clé alors
        retourner (None,−1)
    sinon
        nouvellePage ← PageB(page.cles)
        nouvellePage.parent ← page.parent
        si page._est_page_externe() alors
            nouvellePage.cles ← retirer cle des clés de nouvellePage
        sinon
            indice ← indice de cle parmi les clés de nouvellePage
            pagePrecedente ← page.enfants[indice]
            nouvellePage.enfants ← ajouter tous les enfants de page et les dupliquer
                                   sauf pagePrecedente
            cleRemplacement ← pagePrecedente._cle_predecesseur_dans_enfant()
            nouvellePage.cles[indice] ← cleRemplacement
            (pagePrecedente, nouvelIndice) ← A._suppressionBtree(pagePrecedente,
                                             cleRemplacement, nouvellePage, indice)
            nouvellePage ← pagePrecedente.parent
            nouvellePage.enfants[nouvelIndice] ← pagePrecedente
sinon
    si page._est_page_externe() alors
        retourner (page, indicePage)
    sinon
        nouvellePage ← PageB(page.cles)
        nouvellePage.parent ← page.parent
        indice ← page._indice_page_suivante(cle)
        pageSuivante ← page.enfants[indice]
        nouvellePage.enfants ← ajouter tous les enfants de page et les dupliquer
                               sauf pageSuivante
        (pageSuivante, nouvelIndice) ← A._suppressionBtree(pageSuivante, cle,
                                       nouvellePage, indice)
        nouvellePage ← pageSuivante.parent
        nouvellePage.enfants[nouvelIndice] ← pageSuivante
nouvelIndicePage ← indicePage
si nouvellePage est la racine, ne contient aucune clé et possède un unique enfant alors
    A.racine ← nouvellePage.enfants[0]
    retourner (nouvellePage.enfants[0], -1)
si nouvellePage._est_page_en_deficit(A.ordre) alors
    (nouvellePage, nouvelIndicePage) ← nouvellePage._compensation_deficit(
                                       A.ordre, nouvelIndicePage)
retourner (nouvellePage, nouvelIndicePage)
```

Algorithme 19 : _compensation_deficit

Entrées : page : une page d'un Arbre-B, nbMinCles : le nombre minimal de clés possible dans page, indicePage : l'indice de page parmi les enfants de son parent

Sorties : Le couple (page,nouvelIndice) après compensation du déficit de clés

```
nouvelIndice ← indicePage
frereGauche ← page._frere_gauche(nouvelIndice)
frereDroit ← page._frere_droit(nouvelIndice)

si frereGauche est non vide et la suppression n'entraîne pas de déficit alors
    page._emprunter_cle_frere(frereGauche, nouvelIndice-1, estGauche=True)

sinon si frereDroit est non vide et la suppression n'entraîne pas de déficit alors
    page._emprunter_cle_frere(frereDroit, nouvelIndice, estGauche=False)

sinon
    si frereGauche est non vide alors
        page ← page._fusion_page(frereGauche, nouvelIndice−1, estGauche=True)
        nouvelIndice ← nouvelIndice−1
    sinon si frereDroit est non vide alors
        page ← page._fusion_page(frereDroit, nouvelIndice, estGauche=False)

retourner (page, nouvelIndice)
```

Algorithme 20 : _emprunter_cle_frere

Entrées : page : une page d'un Arbre-B, frere : le frère à qui emprunter une clé, indiceCle : l'indice de la clé du parent à récupérer, estGauche : un booléen indiquant s'il s'agit du frère gauche ou droit

Sorties : frere cède une clé au parent et page récupère un enfant de frere et une clé du parent

```
cleParent ← récupérer la clé à l'indice indiceCle parmi les clés du parent
si estGauche alors
    cleEmpruntee ← récupérer la plus grande clé de frere
    page.cles ← ajouter cleParent comme plus petite clé de page

    si page n'est pas une feuille alors
        enfantFrere ← récupérer le dernier enfant de frere
        enfantFrere.parent ← page
        page.enfants ← ajouter enfantFrere comme premier enfant de page
sinon
    cleEmpruntee ← récupérer la plus petite clé de frere
    page.cles ← ajouter cleParent comme plus grande clé de page

    si page n'est pas une feuille alors
        enfantFrere ← récupérer le premier enfant de frere
        enfantFrere.parent ← page
        page.enfants ← ajouter enfantFrere comme dernier enfant de page
page.parent.cles ← ajouter la clé cleEmpruntee à l'indice indiceCle
```

Algorithme 21 : _fusion_page

Entrées : page : une page d'un Arbre-B, frere : le frère à fusionner avec page, indiceCle : l'indice de la clé du parent à récupérer, estGauche : booléen indiquant s'il s'agit du frère gauche de page

Sorties : fusion la page résultante de la fusion entre page et frere

fusion ← `PageB()`
fusion.parent ← page.parent

si estGauche **alors**
    cleParent ← fusion.parent.cles.pop(indiceCle)
    fusion.cles ← frere.cles + [cleParent] + page.cles
    fusion.enfants ← frere.enfants + page.enfants
sinon
    cleParent ← récupérer la clé à l'indice indiceCle parmi les clés du parent de fusion
    fusion.cles ← page.cles + [cleParent] + frere.cles
    fusion.enfants ← page.enfants + frere.enfants

pour chaque enfant dans fusion.enfants **faire**
    enfant.parent ← fusion

fusion.parent.enfants ← retirer les enfants page et frere
fusion.parent.enfants ← ajouter le nouvel enfant fusion à l'indice indiceCle
retourner fusion

2.8 Arbre Digital (DST)

L'arbre digital est une variante de l'arbre binaire de recherche où chaque noeud contient une clé. Sa recherche s'effectue en fonction des bits. Cet arbre est facile à construire mais sa structure dépend de l'ordre d'insertion puisqu'une clé ne peut pas être préfixe d'une autre clé.

La structure choisie pour un noeud est la suivante :

```
class ArbreBinaire:
    def __init__(self,cle , gauche = None, droite = None):
        self.cle = cle
        self.gauche = gauche
        self.droite = droite
```

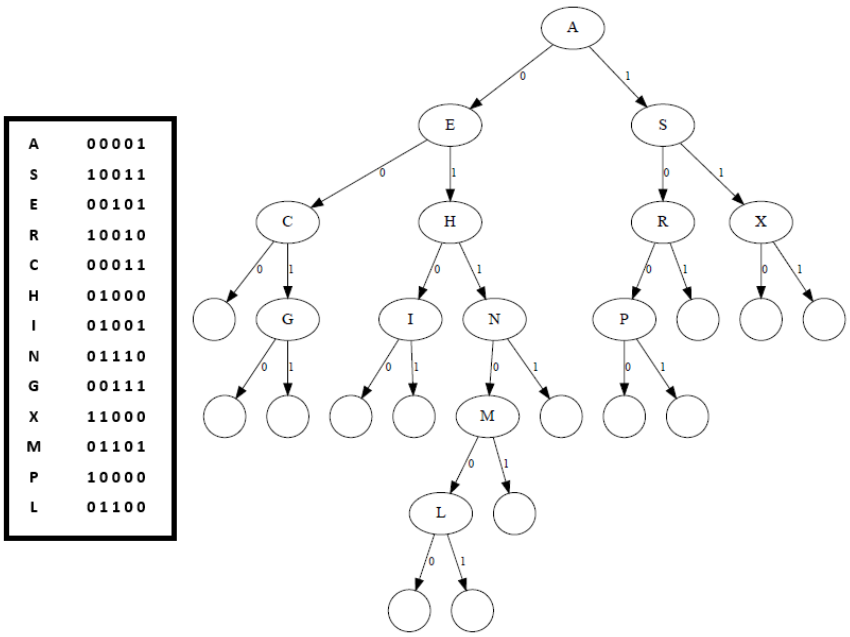


FIGURE 8 – Exemple d'arbre digital avec GraphViz

- 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` parcourt 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 présent, 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 ne 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 $O(n)$.

- Suppression :
  La suppression consiste à retirer un caractère c de l'arbre s'il existe dans l'arbre en nous basant sur le fait que tout caractère possède un encodage en binaire. Nous utilisons alors la méthode `car` pour déterminer le parcours de l'arbre (si le i-ème indice du caractère de

l'encodage en binaire du c est 0, nous appliquons la suppression dans le sous-arbre gauche). Ainsi, si le noeud courant correspond au caractère à supprimer, nous vérifions si l'un des fils en vide et si c'est le cas, nous remplaçons le noeud courant par le fils non vide. Sinon, nous remplaçons le noeud courant par le nœud le plus à gauche (le plus petit) du sous-arbre droit. Puis nous faisons un appel sur le fils droit pour supprimer la clé du noeud de remplacement. Sinon, nous continuons à chercher le caractère à supprimer et nous déterminons le fils sur lequel appliquer la suppression à l'aide de `car`.

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

Algorithme 22 : suppressionDST

Entrées : A un Arbre Digital, c le caractère à supprimer
Sorties : L'Arbre Digital A après suppression du caractère c

retourner A._suppressionDST(A.racine, c, 0)

Algorithme 23 : _suppressionDST

Entrées : noeud : le noeud sous-arbre où supprimer c, c : le caractère à supprimer, i
Sorties : L'Arbre Digital après la suppression du caractère c

```
si noeud == None alors
    retourner None
si noeud.cle == c alors
    si nous sommes sur la racine avec aucun fils alors
        retourner None
    si nous sommes sur la racine avec le fils gauche vide alors
        retourner duplique(noeud.droite)
    si nous sommes sur la racine avec le fils droite vide alors
        retourner duplique(noeud.gauche)
    si nous sommes sur le noeud avec les deux fils vides alors
        retourner None
    si nous sommes sur le noeud avec le fils gauche vide alors
        retourner duplique(noeud.droite)
    si nous sommes sur le noeud avec le fils droite vide alors
        retourner duplique(noeud.gauche)
    sinon
        noeud_min ← noeud_min(noeud.droite)
        nouveau_G ← duplique(noeud.gauche)
        retourner ArbreBinaire(noeud_min.cle, nouveau_G,
                               _suppressionDST(noeud.droite, noeud_min.cle, i+1))
si car(c,i) == 0 alors
    nouveau_D ← duplique(noeud.droite)
    retourner ArbreBinaire(noeud.cle, _suppressionDST(noeud.gauche, c , i+1),
                           nouveau_D)
sinon
    nouveau_G ← duplique(noeud.gauche)
    retourner ArbreBinaire(noeud.cle, nouveau_G, _suppressionDST(noeud.gauche, c ,
                           i + 1))
```

2.9 Arbre Lexicographique (Trie binaire)

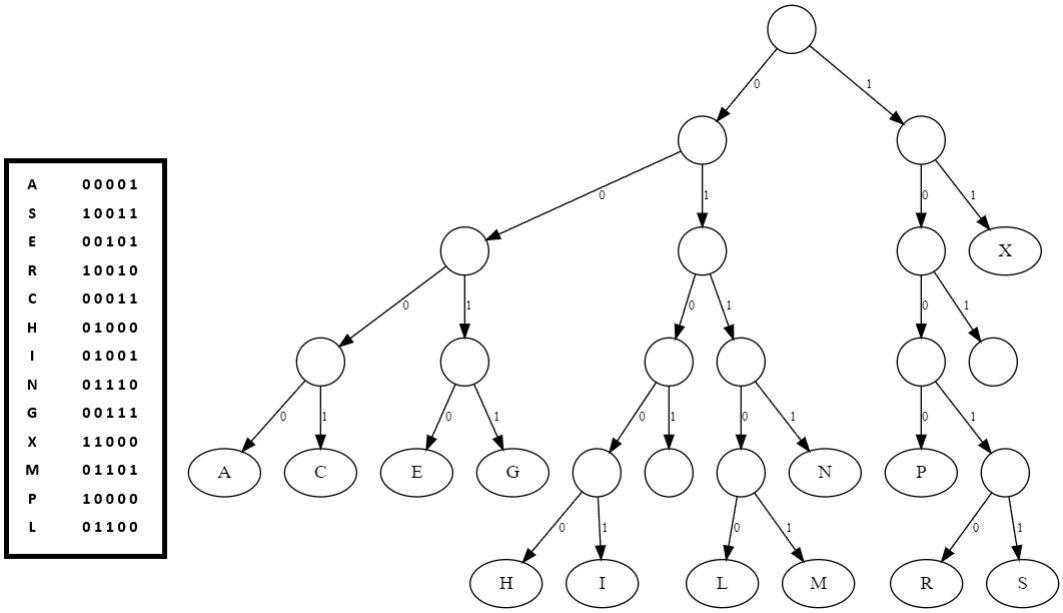


FIGURE 9 – Exemple d'un arbre lexicographique avec GraphViz

Un Trie binaire est une structure d'arbre où chaque noeud interne agit come un point de décision. À une profondeur donnée d et avec un chemin w depuis la racine, un noeud interne appartient à l'ensemble des chaînes binaires 0,1*. Son sous-arbre gauche contient toutes les clés commençant par w0, tandis que sons sous-arbre droit contient toutes les clés qui commencent par w1. Les feuilles contiennent les clés.

La structure choisie pour un noeud :

```
class ArbreBinaire:
    def __init__(self,cle = None , gauche = None, droite = None):
        self.cle = cle
        self.gauche = gauche
        self.droite = droite
```

- Ajout :
  La méthode `ajout` commence par appeler `_ajout`, qui effectue l'insertion récursivement. Si le noeud courant est `None`, alors un noeud 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 noeud.
  - Sinon, nous faisons appel à la méthode `split` en lui donnant les informations nécessaires : le caractère à ajouter, la clé du noeud 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 sa méthode auxiliaire qui fera une suppression à la remontée.
  La méthode **__suppression** suit un parcours récursif à travers l'arbre pour localiser le nœud correspondant au caractère à supprimer. Une fois le nœud identifié, plusieurs scénarios sont envisagés : si le nœud n'a pas d'enfants, il est retiré de l'arbre. En revanche, si le nœud a un seul enfant, cet enfant est remonté pour remplacer le nœud supprimé, assurant ainsi la préservation de la structure de l'arbre. Une approche de suppression à la remonté est comprise afin de mettre à jour l'arbre lors de la suppression du caractère.
  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 $O(n)$.

Algorithme 24 : suppression

Entrées : A : un arbre lexicographique, c : caractères à supprimer
Sorties : supprime le caractère c de l'arbre lexicographique
retourner A._suppression(racine,c,0)

Algorithme 25 : _suppression

Entrées : noeud : noeud courant , c : caractère à supprimer, i : indice du code binaire du caractère à supprimer

Sorties : Renvoie l'arbre lexicographique après la suppression du caractère c

```
res ← noeud
si noeud == None alors
    retourner None
si noeud.cle == c alors
    si le noeud n'a pas de fils gauche et droit alors
        retourner None
    si le noeud n'a pas de fils gauche alors
        nouveau_DD ← duplique(noeud.droite.droite)
        nouveau_DG ← duplique(noeud.droite.gauche)
        res ← ArbreBinaire(noeud.droite.cle, nouveau_DG, nouveau_DD)
    si le noeud n'a pas de fils droite alors
        nouveau_GG ← duplique(noeud.gauche.gauche)
        nouveau_GD ← duplique(noeud.gauche.droite)
        res ← ArbreBinaire(noeud.gauche.cle, nouveau_GG, nouveau_GD)
si car(c,i) == 0 alors
    nouveau_D ← duplique(noeud.droite)
    res ← ArbreBinaire(noeud.cle, _suppression(noeud.gauche, c , i+1), nouveau_D)
sinon si car(c,i) == 1 alors
    nouveau_G ← duplique(noeud.gauche)
    res ← ArbreBinaire(noeud.cle, nouveau_G, _suppression(noeud.gauche, c , i+1))
si nous sommes sur un noeud vide avec aucun enfant alors
    retourner None
si nous sommes sur un noeud avec un enfant gauche et contenant un enfant alors
    retourner res.gauche
si nous sommes sur un noeud avec un enfant droite et contenant un enfant alors
    retourner res.droite
retourner res
```

2.10 R-Trie

Un R-Trie, ou Trie d'arité R, est une variante de la structure de données Trie où chaque noeud peut avoir jusqu'à R enfants. Contrairement au Trie Binaire où chaque nœud a exactement deux enfants, dans un R-Trie, chaque noeud peut avoir un nombre variable d'enfants jusqu'à R.
Dans un R-Trie, les clés sont stockées le long des chemins de la racine jusqu'aux feuilles, et chaque noeud représente un caractère dans la clé. Les chemins sont organisés de manière à ce que les préfixes communs soient partagés entre les clés.

La structure choisie pour un noeud :

```
class rtrie:
    def __init__(self, c , v = None , enfant = [None]*26 ):
```

```
        self.c = c
        self.v = v
        self.enfants = enfant
```

c : caractère, v : valeur, enfant : liste des enfants

Et la racine est une liste de 26 éléments représentant les 26 lettres de l'alphabet.

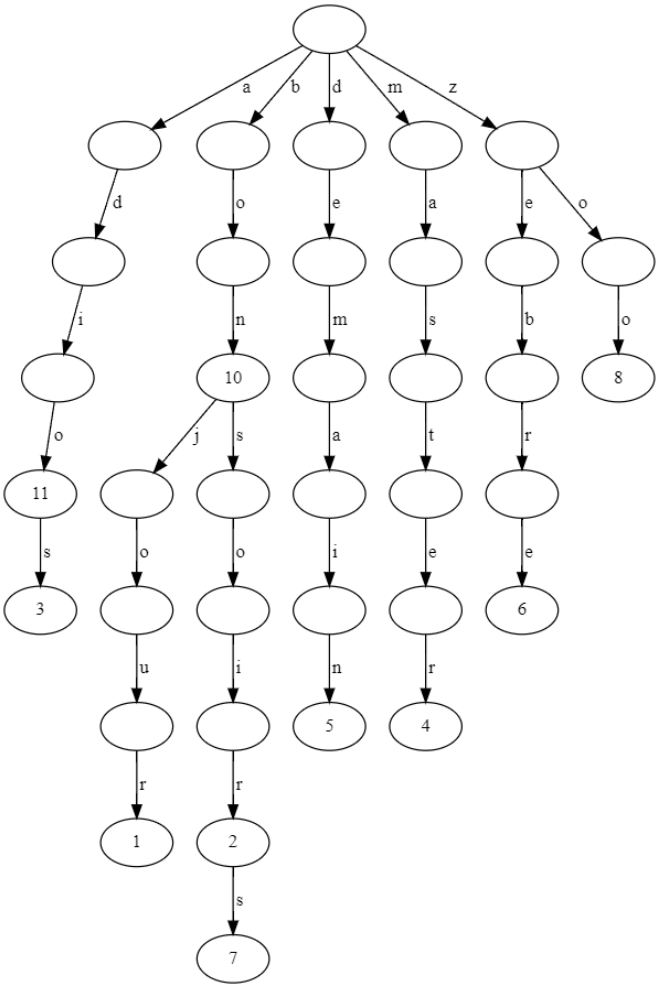


FIGURE 10 – Exemple d'un R-Trie avec GraphViz

- Ajout :
  Cette méthode commence par récupérer la première lettre du mot à ajouter afin de déterminer dans quel nœud (**enfant**), que nous nommerons **p**, nous allons ajouter le mot. Ensuite la fonction **__ajout** est appelée pour ajouter la suite du mot dans le noeud **p**.
  La méthode **__ajout** prend le mot a supprimer, le noeud courant et une valeur de terminaison. Elle traite les cas suivant :
  - Si le noeud courant est vide, nous renvoyons le R-trie contenant le mot.

– Si la longueur du mot est égale à 1, nous ajoutons la valeur de terminaison sur le noeud courant et nous le renvoyons.

Enfin, nous récupérons le prochain caractère à ajouter (**suivant**) et sa position (**psuiv**) afin de renvoyer un R-Trie en lui indiquant : la première lettre du mot, la valeur de terminaison, la position **psuiv**, la liste des enfants du noeud courant excepté l'enfant à la position **psuiv** et un appel récursif **_ajout** avec le reste du mot, le **psuiv**-ème sous-arbre du noeud courant et la valeur de terminaison.

La complexité de l'ajout d'un R-Trie dépend de la longueur du mot à ajouter.
Dans le pire des cas, où chaque caractère du mot nécessite un nouveau nœud distinct, la complexité de l'ajout est linéaire par rapport à la longueur du mot. Cela signifie que la complexité est proportionnelle à la longueur du mot à ajouter, donc $O(n)$, où n est la longueur du mot.

- Suppression :
  Dans un arbre R-Trie, la suppression est gérée en examinant d'abord si le nœud est vide. Ensuite, lorsque nous atteignons le noeud correspondant au dernier caractère du mot à supprimer, la suppression se fait en plusieurs étapes. Si le nœud contient une valeur et n'a pas d'enfants, la valeur est simplement marquée comme nulle, équivalant à la suppression du mot. Si le nœud a d'autres enfants ou valeurs associées, il est réorganisé pour conserver la structure de l'arbre compacte et efficace. La suppression par la remontée permet de maintenir l'intégrité de la structure de l'arbre tout en supprimant les noeuds vides.

  La complexité de la suppression dépend de la longueur du mot à supprimer. Dans le pire des cas, où le mot à supprimer est le seul dans l'arbre ou le dernier de son chemin, la complexité serait proportionnelle à la longueur du mot. Cependant, en moyenne, la complexité serait moindre car elle dépend de la structure de l'arbre et de la distribution des mots. En général, la complexité de la suppression dans un R-Trie est $O(n)$.

Algorithme 26 : suppressionR-Trie

Entrées : A : un arbre R-Trie, mot : mot à supprimer
Sorties : supprime le mot de l'arbre R-Trie
p ← **prem(mot)**
racine[p] ← **A._suppressionR-Trie(mot,racine[p])**
retourner racine

Algorithme 27 : _suppressionR-Trie

Entrées : mot : mot à supprimer, noeud : noeud courant
Sorties : supprime le mot de l'arbre R-Trie
res ← noeud
si res == None **alors**
  retourner None
si len(mot) == 1 **alors**
  retourner res
p ← **prem(mot)**
si len(mot) == 1 **alors**
  si res.v ≠ None and mot[0] == res.c **alors**
    res.v ← None
  sinon
    res.enfants[p] ← **_suppressionR-Trie(reste(mot), res.enfants[p])**
si len(mot) > 1 **alors**
  suivant ← **reste(mot)[0]**
  psuiv ← **prem(suivant)**
  res ← **R_Trie(mot[0], res.v, psuiv, EnfantSauf(noeud, psuiv),**
    _suppressionR-Trie(reste(mot), SousArbre(noeud, psuiv)))
si mot[0] == res.c and contientNone(res.enfants) and res.v ≠ None **alors**
  retourner res
si mot[0] == res.c and contientNone(res.enfants) **alors**
  retourner None
retourner res

2.11 Trie Hybride

Un trie hybride est une variante de trie où chaque noeud peut avoir jusqu'à trois enfants : un enfant gauche (Inf), un enfant du milieu (Eq), et un enfant droit (Sup). Chaque noeud contenir une valeur associée à un caractère.
L'organisation des nœuds dans un trie hybride permet une recherche efficace de mots ou préfixes de mots dans l'ensemble de données. Les caractères des mots sont stockés le long des chemins de la racine jusqu'aux feuilles de l'arbre, et chaque nœud du milieu représente un caractère dans le mot.

La structure choisie pour un noeud :

```
class TrieH:
    def __init__(self , c , inf, eq , sup, v=None):
        self.c = c
        self.inf = inf
        self.eq = eq
        self.sup = sup
        self.v = v
```

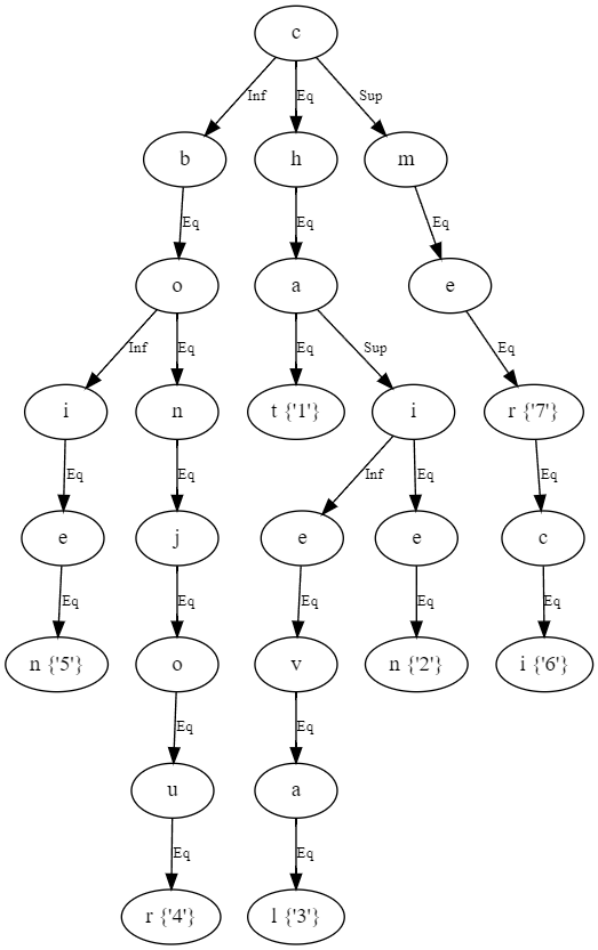


FIGURE 11 – Exemple d'un trie hybride avec GraphViz

- 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 $O(\log n)$. 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.

- Suppression :
  La suppression dans un trie hybride commence par vérifier si le noeud est vide. Ensuite, elle identifie le premier caractère du mot à supprimer et compare sa valeur avec celle du noeud courant. Si le caractère est inférieur, la suppression est effectuée dans le sous-arbre gauche. Si le caractère est supérieur, la suppression est effectuée dans le sous-arbre droit. Si le caractère est égal, la suppression continue dans le sous-arbre central.
  Lorsque nous atteignons le noeud correspondant au dernier caractère du mot à supprimer, une approche de suppression par la remontée est utilisée pour maintenir la structure de l'arbre, en vérifiant plusieurs cas. Elle gère divers scénarios, notamment la suppression au niveau de la racine ou en dehors de celle-ci.

  La complexité de la suppression dépend de la longueur du mot à supprimer. Dans le pire des cas, où l'arbre est équilibré et que le mot à supprimer est le seul dans l'arbre ou le dernier de son chemin, la complexité serait proportionnelle à la longueur du mot. En général, la complexité est similaire à celle de la recherche, c'est-à-dire en $O(n)$, où n est la longueur maximale des mots dans l'arbre.

Algorithme 28 : suppressionTrieHybride

```
Entrées : mot : mot à supprimer, noeud : noeud courant
Sorties : supprime le mot de l'arbre hybride
res ← noeud
si noeud == None alors
    retourner None
p ← prem(mot)
si  p < noeud.c alors
    n_Eq ← duplique(noeud.eq)
    n_Sup ← duplique(noeud.sup)
    res ← TrieH(noeud.c, suppression(mot, noeud.inf), n_Eq, n_Sup, noeud.v)
si p > noeud.c alors
    n_Inf ← duplique(noeud.inf)
    n_Eq ← duplique(noeud.eq)
    res ←
      TrieH(noeud.c, n_Inf, n_Eq, suppressionTrieHybride(mot, noeud.sup), noeud.v)
si p == noeud.c alors
    si lg(mot) == 1 and noeud.v != None alors
        noeud.v ← None
    sinon
        n_Inf ← duplique(noeud.inf)
        n_Sup ← duplique(noeud.sup)
        res ← TrieH(noeud.c, n_Inf, suppressionTrieHybride(mot, noeud.eq),
                    n_Sup, noeud.v)
si nous sommes sur la racine sans aucun fils alors
    retourner None
si nous sommes sur la bonne lettre avec aucun fils ni de valeur de terminaison alors
    retourner None
si nous sommes sur un noeud sans fils ni de valeur de terminaison alors
    retourner None
si nous sommes sur un noeud avec un fils Eq et Sup alors
    retourner Fusion(res.inf, res.sup)
si nous sommes sur un noeud avec seulement le fils Inf alors
    retourner res.inf
si nous sommes sur un noeud avec seulement le fils Sup alors
    retourner res.sup
retourner res
```

La methode **Fusion**, permet de fusionner deux noeuds pour qu'on ait au moins le fils Eq.

Conclusion

Ce projet collaboratif a été une expérience enrichissante, nous permettant d'appliquer les concepts théoriques appris dans le cadre du cours d'Algorithmique Avancée (AlgAv) du M1. L'objectif était de créer une bibliothèque unifiée en Python pour manipuler et visualiser diverses structures de données arborescentes. Nous avons pu mettre en œuvre toutes les structures arborescentes étudiées. Chaque structure a été développée dans des fichiers Python séparés pour une meilleure organisation. L'utilisation des notebooks Jupyter a permis d'illustrer de manière interactive comment chaque structure est implémentée et fonctionne. L'intégration de fonctionnalités de visualisation via GraphViz a permis une représentation graphique claire et didactique des données, renforçant ainsi notre compréhension des structures étudiées.

Ce projet a été une occasion précieuse d'approfondir nos connaissances pratiques en algorithmique et en programmation, tout en développant des compétences collaboratives et de communication au sein de l'équipe.

Références

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

[2] R. Sedgewick, 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.