Fabrice Senécal

Travail Pratique #3

Présenté à   
Gabriel Trudeau-St-Hilaire

Structure De données et   
Algorithmie Avancée

Cégep Saint-Jean-sur-Richelieu

8 décembre 2023

Table des matières

[Table des figures 4](#_Toc152963555)

[Sommaire 5](#_Toc152963556)

[Fonctionnement explicite 5](#_Toc152963557)

[6](#_Toc152963558)

[La classe Algorithme\_1 7](#_Toc152963559)

[CalculeFrequence 8](#_Toc152963560)

[CompresseMessage 8](#_Toc152963561)

[FaireEncodage et FaireEncodageRecursif 9](#_Toc152963562)

[DecompresseMessage 9](#_Toc152963563)

[La classe SystemeDEncodage 11](#_Toc152963564)

[TrieQueue et TrieQueueRecursif 12](#_Toc152963565)

[TrouverPlusPetitIndex 13](#_Toc152963566)

[La classe nœud (Node) 14](#_Toc152963567)

[Algorithme de Dijkstra 15](#_Toc152963568)

[Fonctionnement explicite 15](#_Toc152963569)

# Table des figures

[Figure 1: Classes utilisées 5](file:///C:\Users\fabou\Desktop\Prog%203%20Structure\TP3\Documentation%20TP3.docx#_Toc152963575)

[Figure 2: Conversion d'une string en code binaire 5](file:///C:\Users\fabou\Desktop\Prog%203%20Structure\TP3\Documentation%20TP3.docx#_Toc152963576)

[Figure 3: Étapes de compression 6](file:///C:\Users\fabou\Desktop\Prog%203%20Structure\TP3\Documentation%20TP3.docx#_Toc152963577)

[Figure 4: Fréquence d'apparition en pourcentage de chaque caractères 8](file:///C:\Users\fabou\Desktop\Prog%203%20Structure\TP3\Documentation%20TP3.docx#_Toc152963578)

[Figure 5: Équivalent en bits de chaque caractères 9](file:///C:\Users\fabou\Desktop\Prog%203%20Structure\TP3\Documentation%20TP3.docx#_Toc152963579)

[Figure 6: exemple graphique d'un arbre binaire 11](file:///C:\Users\fabou\Desktop\Prog%203%20Structure\TP3\Documentation%20TP3.docx#_Toc152963580)

[Figure 7: exemple du trie d'une queue 12](file:///C:\Users\fabou\Desktop\Prog%203%20Structure\TP3\Documentation%20TP3.docx#_Toc152963581)

[Figure 8: exemple de l'index de position de la plus petite donnée 13](file:///C:\Users\fabou\Desktop\Prog%203%20Structure\TP3\Documentation%20TP3.docx#_Toc152963582)

[Figure 9: Exemple graphique d'un nœud 14](file:///C:\Users\fabou\Desktop\Prog%203%20Structure\TP3\Documentation%20TP3.docx#_Toc152963583)

[Figure 10: représentation en matrice d'un graphe 15](file:///C:\Users\fabou\Desktop\Prog%203%20Structure\TP3\Documentation%20TP3.docx#_Toc152963584)

[Figure 11: représentation graphique de la rangé #1 de la figure 10 16](file:///C:\Users\fabou\Desktop\Prog%203%20Structure\TP3\Documentation%20TP3.docx#_Toc152963585)

[Figure 12: méthode MaxValue 16](file:///C:\Users\fabou\Desktop\Prog%203%20Structure\TP3\Documentation%20TP3.docx#_Toc152963586)

[Figure 13: résultat du graphe représenté à la figure 10 17](file:///C:\Users\fabou\Desktop\Prog%203%20Structure\TP3\Documentation%20TP3.docx#_Toc152963587)

Algorithme 1

## Sommaire

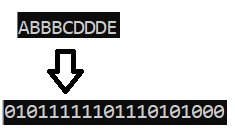
Le premier algorithme présent dans la solution est l’Algorithme 1. Celui-ci utilise les classes « [Algorithme\_1.cs](#_La_classe_Algorithme_1)», « [Node.cs](#_La_classe_nœud)» ainsi que « [SystèmeDEncodage.cs](#_La_classe_SystemeDEncodage)». En bref, L’Algorithme 1 est un algorithme de compression de chaines de caractères donc de String. Il convertit les Strings en nouvelles chaines composées uniquement de bits. Il obtempère en catégorisant dans un dictionnaire chaque caractère selon sa fréquence d’apparition en pourcentage. Les caractères les plus fréquents se voient attribuer une suite de bits courts et les caractères moins fréquents possèdent une suite de bits plus longs. Cette méthode de conversion est optimale pour réduire le poids d’un fichier dont les caractères sans compression sont initialement de 8 bits.

Figure : Classes utilisées

Figure : Conversion d'un string en code binaire

## Fonctionnement explicite

En premier lieu, la chaine de caractère original est déclarée, dans la suite des explications celle-ci sera référée à « string original » pour éviter toute confusion. Ensuite, une instance de la classe « [Algorithme\_1](#_La_classe_Algorithme_1)» est déclarée pour accéder aux méthodes de compression publiques. Une fois la classe instanciée, un nouveau string est initialisé. Cette nouvelle chaine de caractères contiendra l’équivalent en bits du string original, donc le texte compressé. Le string compressé est initialisé avec l’instance de l’ « Algorithme\_1 » auquel on fait appel à la méthode « CompressionComplete » en passant en paramètre le string original. Ainsi, le string détient désormais l’équivalent compressé du string original. La [décompression](#_DecompresseMessage) du message fonctionne de la même façon. Néanmoins, la même instance de l’ « Algorithme\_1 » doit être utilisée pour la compression et la décompression. En effet, il s’agit d’une des deux limitations de l’algorithme. Une même instance ne pourrait pas être utilisée pour traiter plusieurs différents strings à compresser, car pour ce faire, l’algorithme devrait créer un nouveau système d’encodage. De ce fait, il perdrait son système précédent nécessaire pour la décompression. La deuxième limitation est que l’algorithme ne peut pas traiter le caractère étoile « \* ». Il est attribué par défaut au nœud parent et est utilisé pour détecter si le [nœud](#_La_classe_nœud) est une feuille.

3



1

## 

2



Figure : Étapes de compression

3

## La classe Algorithme\_1

« Algorithme\_1 » est la classe centralisant chacune des autres classes utilisées par le programme. Dans le fichier principal du projet, elle est l’unique classe devant être instanciée pour compresser et décompresser une chaine de caractères. Elle contient quatre variables privées. Deux d’entre elles sont des dictionnaires donc des structures de données clés et valeurs. Le premier dictionnaire est celui des fréquences en pourcentage. Il utilise un caractère comme clé et un chiffre double pour la valeur. Il sert à entreposer le pourcentage de fréquence de chaque caractère présent dans le string original. Le deuxième dictionnaire utilise un caractère comme clé et une chaine de caractères comme valeur. Celui-ci contient la définition de chaque caractère et leurs équivalents en bits comme un dictionnaire contient les définitions des mots. Les deux autres variables sont des instances de classes. La première est une instance de la classe [nœud](#_La_classe_nœud), puis la deuxième est une instance de la classe " [SystèmeDEncodage](#_La_classe_SystemeDEncodage)". « Algorithme\_1 » contient six méthodes dont deux sont publiques et quatre d’entre elles sont privées. La première méthode est « CompressionComplete ». Elle utilise un paramètre qui est un string représentant le string original devant être compressé. Cette méthode est utilisée pour réunir les autres méthodes et données de la classe nécessaire pour la compression de données. Premièrement, « CompressionComplete » fait appel à une nouvelle méthode qui est « [CalculeFrequence](#_CalculeFrequence) » tout en lui passant en paramètre le string original. Ainsi le dictionnaire des pourcentages est désormais initialisé avec un référencement des fréquences de chaque caractère dans le string. De ce fait, la variable globale « root » peut-être initialiser avec la méthode « ConstruireArbre ». En effet, car étant une méthode de l’instance « [SystemeDEncodage](#_La_classe_SystemeDEncodage) » cette fonction nécessite d’avoir le dictionnaire des pourcentages pour construire l’arbre. Finalement, après avoir initialisé la variable globale « root », « CompressionComplete » peut enfin appeler la méthode « [CompresseMessage](#_CompresseMessage)». Puisque la méthode « [CompresseMessage](#_CompresseMessage) » s’occupe de la compression, elle a comme valeur de retour le string original désormais compressé. Ainsi, la méthode « CompressionComplete » retourne le résultat retourné par « [CompresseMessage](#_CompresseMessage) ». La deuxième méthode publique présente dans l’ « Algorithme\_1 » est la méthode « [DecompresseMessage](#_DecompresseMessage)» qui s’occupe de reconvertir le string compressé en son message initial.

### CalculeFrequence

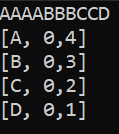
« CalculeFrequence » est une méthode privée de l’ « [Algorithme\_1](#_La_classe_Algorithme_1)». Elle a comme unique but d’initialiser le dictionnaire contenant la fréquence d’apparition en pourcentage des caractères. La méthode prend en paramètre un string qui est le string original. « CalculeFrequence » débute en initialisant un nouveau dictionnaire unique à la méthode qui contiendra les fréquences. Ensuite, le programme rentre dans une boucle « foreach » qui passe au travers de chaque caractère du string passé en paramètre. Pour chaque caractère, le programme doit regarder si le dictionnaire des fréquences contient la lettre. Pour ce faire, la condition utilise la méthode « ContainsKey ». Il s’agit d’une méthode de la classe « Dictionary » qui retourne vrai si la clé est trouvée. Dans notre cas, la clé est le caractère. Si la clé est trouvée, la valeur liée à la clé est incrémentée de 1. Sinon, le caractère est ajouté avec une valeur de 1. Une fois chaque caractère passé dans la boucle, le résultat est un dictionnaire contenant chaque caractère et le nombre de fois qu’elles apparaissent. Finalement, une autre boucle "foreach" est utilisée pour cette fois si passer au travers de chaque paire, clé et valeur, présent dans le dictionnaire. Dans cette deuxième boucle, le programme doit traiter le dictionnaire des pourcentages qui est une des variables globales. Dans la boucle, deux données sont ajoutées aux dictionnaires, la clé, puis la valeur. La valeur est la fréquence divisée par la somme de toutes les valeurs présente dans dictionnaire de la première boucle. Ainsi le résultat final est que le dictionnaire des pourcentages est désormais initialisé avec le pourcentage d’apparition de chaque caractère présent dans le string original.

Figure : Fréquence d'apparition en pourcentage de chaque caractère

### CompresseMessage

« CompresseMessage » est la fonction privée faisant appel aux deux autres méthodes privées présentes dans la classe « Algorithme\_1 ». En premier lieu, la méthode observe si le compte du dictionnaire de code est de zéro. Si tel est le cas, elle fait appel à la méthode « [FaireEncodage](#_FaireEncodage_et_FaireEncodageRecur)». Ainsi le deuxième dictionnaire dans les variables globales est initialisé. Pour rappel, ce deuxième dictionnaire contient l’équivalent en bits de chaque caractère utilisé dans le string original. « CompresseMessage » déclare ensuite un string vide. La contenue de ce string est définie dans la boucle « for » suivant sa déclaration. La boucle s’incrémente tant que la donnée « i » est plus petite que le nombre de caractères présent dans le string original passé en paramètre. Pour chaque tour de boucle, le string vide se voit concaténer l’équivalent en bits du caractère du string original, présent dans le dictionnaire. En d’autres termes, le programme va chercher dans le dictionnaire la lettre du string original à la position « i », pour retourner la valeur. Cette valeur étant une suite de bits est concaténée au string vide déclaré avant la boucle. Une fois la boucle terminée, le message est complètement compressé et se fait retourner par la méthode.

### FaireEncodage et FaireEncodageRecursif

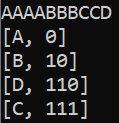
« FaireEncodage » et « FaireEncodageRecursif » sont deux méthodes « void » étroitement liées qui ont pour but d’initialiser le deuxième dictionnaire utilisé par la classe. Celui-ci contient tous les caractères et leur équivalent en bits selon le string original et l’arbre binaire fait en dépend par « [SystemeDEncodage](#_La_classe_SystemeDEncodage) ». « FaireEncodage » est une méthode « one-liner » utilisée simplement pour encadrer l’appel de la méthode « FaireEncodageRecursif ». L’appel est fait en passant le [nœud](#_La_classe_nœud) « root » précédemment initialiser et un string vide qui représente le caractère une fois encodé. « FaireEncodageRecursif » est une méthode récursive assez simple. Sa condition de sortie observe si le caractère du nœud est différent du caractère étoile « \* » assigné aux nœuds parents. Ainsi, le programme peut détecter s’il est en train de traiter une feuille. Si le caractère est bel et bien une feuille, celui-ci et le string, passé en paramètre, sont tous deux ajoutés au dictionnaire. Sinon, la méthode est rappelée deux fois de suite. Le premier rappel se fait en envoyant en paramètre l’enfant situé à gauche de la racine, puis en concaténant le chiffre 0 au string représentant le caractère. Le deuxième se fait en envoyant l’enfant à droite et en concaténant le chiffre 1 au string. Ainsi si le nœud est une feuille le programme ne fera pas un rappel de la fonction et celle-ci se terminera. L’importance d’avoir les deux rappels est pour être certain d’aller chercher chaque nœud à gauche et à droite dans l’arbre binaire. Par exemple, lorsque le programme fini de traiter une feuille a gauche, il retourne dans l’appel de fonction précédant et peut désormais traiter la feuille située à droite de du parent. La méthode récursive permet donc d’initialiser le dictionnaire des caractères compressé en assignant une suite de bits plus longue aux caractères plus profonds. L’utilité d’assigner le bit 1 ou 0 sera utile lors de la décompression.

Figure : Équivalent en bits de chaque caractère

### DecompresseMessage

Cette méthode publique retourne une chaine de caractères ayant la possibilité d’être nuls. Tout d’abord elle prend en paramètre un string qui est le message à décompresser. Dans le code, vérifie si la variable « root » a été initialisée. Si ce n’est pas le cas, elle retourne un nul. Cette vérification est nécessaire, car si la racine est nulle c’est qu’il n’y a pas d’arbre binaire pour permettre à la méthode décompressée de se diriger. De plus, cela veut aussi dire qu’aucun message n’a été compressé en premier lieu. Après la vérification, un nouveau [nœud](#_La_classe_nœud) est déclaré avec le nœud racine. Celui-ci est nommé nœud actuel. Un « StringBuilder » est aussi utilisé pour construire optimalement le string. Ensuite, une boucle « foreach » est utilisée pour passer au travers de chaque bit présent dans le message compressé. Pour chaque tour de boucle, si le bit est égal à zéro le nœud actuel est remplacé par le nœud enfant à gauche. Si le bit est égal à un c’est l’enfant positionné à droite qui remplace le nœud actuel. Avant de terminer le tour de boucle, le programme finit par vérifier si le caractère est une feuille. Si tel est le cas, le caractère présent dans le nœud est ajouté au « StringBuilder », puis le nœud actuel se voit réattribuer la référence du nœud racine. Ainsi, en traversant chaque bit du message compressé, le nœud peut savoir s’il doit se diriger à droite ou à gauche dans l’arbre binaire. De plus, lorsqu’il rencontre une feuille, la racine réattribuée au nœud actuel permet de recommencer la recherche du caractère selon les bits restants. Enfin, le « StringBuilder » est renvoyé par la méthode en lui appliquant une conversion en string.

## La classe SystemeDEncodage

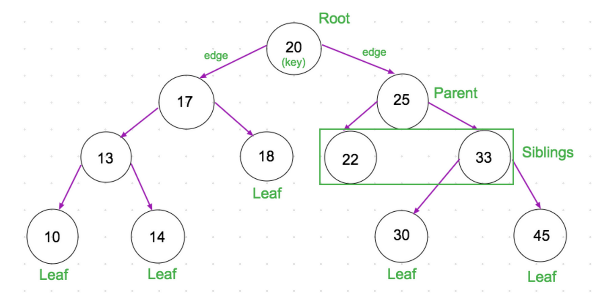
La classe « SystèmeDEncodage » ne contient pas de constructeur. En effet, car elle n’emploie aucune variable globale ou de propriété pour son fonctionnement. Cette classe contient quatre méthodes pour effectuer son unique tâche qui est la construction de l’arbre binaire. La première méthode est l’unique méthode publique de la classe, la méthode « ConstruireArbre ». Elle prend en paramètre le dictionnaire des pourcentages au préalablement déclaré. La méthode commence par créer une nouvelle instance d’un dictionnaire avec le même contenu que celui des pourcentages en utilisant la méthode « OrderBy ». Cette méthode renvoie un dictionnaire ordonné selon les paramètres mis en place. Dans notre cas la méthode « OrderBy » une fois appliquer a pourcentage renvoie une copie don les valeurs ont été mises en ordre croissant. Ensuite, une variable de type queue appelé « queueOrdonne » est déclarée pour contenir des objets [nœuds](#_La_classe_nœud). Une boucle « foreach » est alors utilisée pour parcourir toutes données du dictionnaire ordonnées afin de les enfiler dans la queue. Une fois la queue remplie, le programme entre dans une boucle « while ». La condition de sortie est que le compte des nœuds présent dans la queue est plus grand que 1, le programme doit continuer de boucler. Dans la boucle « while », deux variables de type « var » appelées respectivement « gauche » et « droite » sont déclarées. Ces dernières sont initialisées avec la méthode « Dequeue » appliquée sur la queue ordonnée renvoyant donc les deux premiers nœuds présents tout en les retirant de celle-ci. Les deux variables sont maintenant de type [nœud](#_La_classe_nœud) et la queue possède désormais deux nœuds en moins. Par après, un nouveau [nœud](#_La_classe_nœud) parent est déclaré. Il a l’adition des pourcentages de la fréquence d’apparition des nœuds gauche et droite précédemment déclarée et le caractère étoile « \* » lui est automatiquement assigné. De ce fait, un nœud doté du caractère étoile peut être identifié comme une branche. Le nœud parent est inséré dans la queue qui est ensuite triée avec la deuxième méthode « [TrieQueue](#_TrieQueue_et_TrieQueueRecursif) » . Le cycle de la boucle « while » continue en retirant les deux prochains membres de la queue qui sont constamment les plus petits (en termes de pourcentage) pour ensuite être lié a un parent qui est inséré dans la queue. Cette boucle permet donc de créer un arbre binaire des feuilles vers la racine. Le but de créer un arbre binaire de cette façon est pour obtenir un système qui permettra un encodage binaire ou chaque feuille est un caractère. Les caractères les plus fréquents seront situés dans un nœud plus près de la racine, car avec le [tri](#_TrieQueue_et_TrieQueueRecursif) de la queue, les nœuds ayant des caractères plus fréquents seront insérés en dernier. Ce système est important, car en circulant dans l’arbre, les prochaines méthodes pourront attribuer une suite de bits courts aux feuilles étant plus proche de la racine. Enfin, une fois sorti de la boucle la méthode « ConstruireArbre » retourne la racine de l’arbre binaire qui contient désormais un pourcentage de 100% une fois toutes les fréquences additionnées.

Figure : exemple graphique d'un arbre binaire

### TrieQueue et TrieQueueRecursif

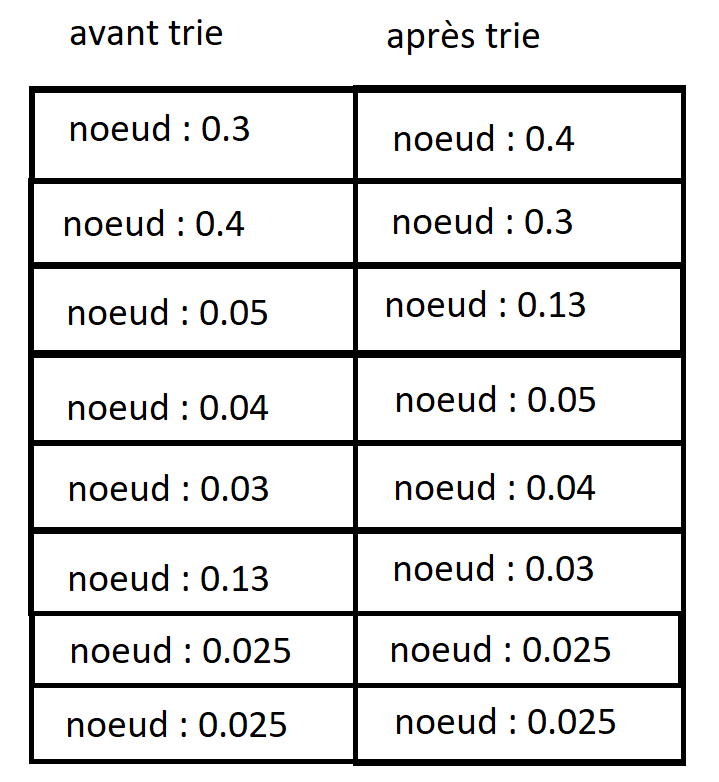
La méthode « TrieQueue » est de type « void » et prend en paramètre une queue de [nœuds](#_La_classe_nœud). D’abord, la classe instancie une nouvelle liste de type [nœud](#_La_classe_nœud) à ordonner à partir des données de la queue. Ensuite, une nouvelle liste nommée liste ordonnée est instanciée à partir de la méthode « TrieQueueRecursif » qui est appelée en véhiculant en paramètre la liste de queues à ordonner. « TrieQueueRecursif » est une méthode récursive donc qui se rappelle elle-même jusqu’à ce qu’une condition de sortie soit remplie. Dans son cas, la méthode commence par analyser si le compte de la liste à ordonner passé en paramètre est plus petit que 1. Si c’est le cas, la méthode sort en retournant la liste à ordonner. Un « Integer (int) » est instancié avec la méthode « [TrouverPlusPetitIndex](#_TrouverPlusPetitIndex)» qui comme le nom laisse paraitre, trouve l’index de position du nœud contenant le pourcentage le plus petit. Une fois trouvé un nouveau nœud est instancié avec le nœud présent à l’index 0 dans la liste. Le nœud présent à l’index 0 est ensuite remplacé par le nœud présent à l’index acquis avec la méthode « [TrouverPlusPetitIndex](#_TrouverPlusPetitIndex)» . Puis, le nœud présent à l’index trouvé par la méthode est remplacé par le nœud instancié qui lui était anciennement positionné à l’index 0. En bref, l’index où se trouve le nœud avec le pourcentage d’apparition le plus petit est trouvé et ensuite échangé de place avec le nœud à l’index 0. Ainsi les nœuds avec des lettres plus rares sont ramenés au début de la liste. Le programme continue en rappelant la méthode « TrieQueueRecursif » tout en lui passant encore la liste où la méthode « skip » avec un paramètre de 1 est appliquée. La méthode « skip » permet de renvoyer le contenu de la liste tout en sautant le nombre de données précisé en paramètre. Dans notre cas, le rappelle de « TrieQueueRecursif », avec en paramètre la liste dont la première donnée est sautée, est utilisé pour initialiser une nouvelle liste de nœuds qui est appelée « resteDeListe » qui sera la liste retournée par défaut par la méthode.

Figure : exemple du trie d'une queue

### TrouverPlusPetitIndex

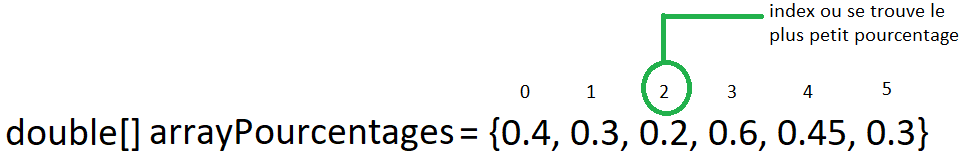
Cette méthode prend en paramètre une liste devant être ordonnée. Elle déclare un nouveau « int » avec une valeur de 0. Ce dernier représente l’index ou se situe le nœud avec le plus petit pourcentage. Le programme rentre ensuite dans une boucle « for » qui boucle tant que « i » est plus petit que le compte de la liste à ordonner. Le « i » incrémente à chaque tour afin de passer au travers de tous les objets présents dans la liste. Pour chaque tour de boucle, le programme doit déterminer si le pourcentage du nœud présent au point « i » est plus petit que celui présent au point de l’index déclaré avant la boucle. Si c’est le cas, la valeur du point « i » est attribuée à l’index. Ainsi de suite jusqu’à ce que la boucle termine. De cette façon, puisque la boucle est passée au travers de toute donnée présente dans la liste, pour faire l’échange d’index si nécessaire, nous sommes sûrs d’avoir le bon index à envoyer. Ayant comme type de retour un « Integer », la méthode finie par renvoyer l’index de la liste a ordonné ou se situe le nœud avec le plus petit pourcentage.

Figure : exemple de l'index de position de la plus petite donnée

## La classe nœud (Node)

Figure : Exemple graphique d'un nœud

La classe nœud (Node) est une classe utilisée pour créer un arbre binaire. Elle contient deux propriétés avec « setter » privé. Une propriété « char » pour le caractère et une propriété « double » pour le pourcentage d’apparition. Elle possède aussi deux propriétés sans « setter » privé de type « Node » pour positionner les nœuds suivants à gauche ou à droite. L’utilisation de « setter » privé pour les deux premières propriétés est pour rendre imputables les données une fois le nœud instancier. Contrairement aux propriétés de types « Node » que l’on veut pouvoir modifier plus tard. De plus, ces deux dernières propriétés ont la possibilité d’être nulles et sont initialisées de la sorte par le constructeur. Cela permet d’instancier un nœud sans obligatoirement spécifier les enfants à gauche ou à droite ainsi que de déclarer un nœud de type « feuille ». Toutes les propriétés de la classe nœud (Node) sont publiques afin d’y avoir accès en dehors de la classe.