Université de Nantes — UFR Sciences et Techniques
Master informatique parcours “optimisation en recherche opérationnelle (ORO)”
Année académique 2018-2019

Projet de Langages de programmation de haut niveau

Marie Humbert–Ropers[1] – Arthur Gontier[2]

23 décembre 2018

Table des matières

1 Structures de données utilisées

L'objectif de ce projet est de comparer deux structures de données : un tableau et un arbre. Nous voulons déterminer laquelle est la plus utile dans le cadre d'un stockage d'informations sur les mots d'un texte et pour l'exploration de ces données.
Pour ce faire, huit fonctions ont été implémentées pour chacune des deux structures de données.

- remplissage : Remplissage de la structure de données à partir d'un texte donné
- motPresent : Test si un mot est présent dans le texte
- longMoyenne : Recherche de la longueur moyenne des mots du texte
- NbmotsDistincts : Recherche du nombre de mots distincts présents dans le texte
- ListeMotsDeb : Recherche d'une liste de mot avec un préfixe donné
- ListeMotsFin : Recherche d'une liste de mot avec un suffixe donné
- NbOccurLettre : Recherche du nombre d'occurrences d'une lettre
- NbMotsAvecNLettres : Recherche du nombre de mots avec un certain nombre de fois une lettre

2 Problèmes rencontrés lors de l'implémentation et solutions

2.1 Les caractères spéciaux

Le problème récurrent dans ce projet a été le traitement des caractères spéciaux en Unicode, en particulier sur les lettres accentuées.
En julia, les string peuvent être pris comme des tableux de caractères. Nous avons considéré chaque mot comme une string, et donc comme un tableau de lettres. Nous avons implémenté les fonctions donnant les listes de mots contenant un certain préfixe, ou suffixe, en utilisant cette particularité. Or, les caractères spéciaux d' Unicode prennent plus de place en mémoire. Nous avons pu constaté que, si l'un des caractères du mot est un caractère non présent dans la table ASCII mais Unicode, alors son codage est plus conséquent et ne prend pas une case du tableau de caractères mais deux. L'appel à la première de ces deux cases rend le bon caractère, cependant la lecture de la deuxième case renvoie une erreur : "UnicodeError : invalid character index". Ainsi, il était impossible de parcourir le tableau issu d'une string. Afin de ne pas prendre en compte cette case invalide, nous avons utilisé la fonction isvalid() qui permet de tester si la case donnée représente un caractère.
L'implémentation réalisée fait donc parcourir tout le tableau en vérifiant la validité de chacune des cases. Cependant, cette implémentation engendre un second problème. Nous remarquons que la fonction length() renvoie le nombre de caractère d'un mot et pas la taille du tableau, c'est-à-dire que length() compte bien chaque caractère Unicode comme un caractère. Nous avons donc utilisé la fonction sizeof() afin d'obtenir la taille des tableaux et de les parcourir.

2.2 Les structures

La définition des structures est bien reconnue par Julia. Cependant, la structure récursive des arbres ne peut être définie qu'une seule fois dans le REPL.

3 Comparaison des temps et espaces des différentes fonctions

Premiers constats après quelques tests :
— D'après les résultats, deux fonctions avec le même objectif mais pas la même structure de données renvoient bien entendu le même résultat mais, une liste rendue par la fonction utilisant un tableau sera toujours triée contrairement à celle utilisant un arbre.
— Les temps de recherches sont rapides et de l'ordre de 10^{-3} au maximum. Afin d'avoir des temps significatifs pour comparer les fonctions et de s'assurer d'avoir des différences de temps dues aux structures de données, les temps d'exécutions retenus correspondent à 1000 exécutions d'une même fonction, à l'exception des fonctions de remplissage.

Afin d'avoir une idée des fonctions les plus coûteuses en temps, nous estimons ces coûts dans le pire des cas. En particulier, le nombre de mots distincts dans le pire des cas sera le nombre de mots n. Cela implique, par exemple, que le nombre de noeuds de l'arbre correspond au nombre de mots fois la somme m de la taille de chacun des mots. La taille des mots en français étant inférieure ou égal à 26, on estime que la taille des mots est négligeable. On note S l'ensemble des noeuds de l'arbre. Grâce à la structure de cet arbre, la complexité du parcours de tous les noeuds est de $\mathcal{O}(Card(S))$. Les coûts théoriques des fonctions sont récapitulées dans le tableau suivant, avec Nb un nombre de lettre :

	Arbre	Tableau
remplissage	$\mathcal{O}(n * m)$ (pire des cas)	$\mathcal{O}(n)$
motPresent	$\mathcal{O}(1)$ (la taille du mot est une constante)	$\mathcal{O}(log(n))$
longMoyenne	$\mathcal{O}(Card(S))$	$\mathcal{O}(n)$
NbmotsDistincts	$\mathcal{O}(Card(S))$	$\mathcal{O}(1)$
ListeMotsDeb	$\mathcal{O}(Card(S))$ (pire des cas)	$\mathcal{O}(n)$ (pire des cas)
ListeMotsFin	$\mathcal{O}(Card(S))$	$\mathcal{O}(n)$
NbOccurLettre	$\mathcal{O}(Card(S))$	$\mathcal{O}(n)$
NbMotsAvecNLettres	$\mathcal{O}(n * Nb)$ (pire des cas)	$\mathcal{O}(n)$

TABLE 1 – Tableau du coût théorique de chacune des fonctions

(Pour les fonctions retournant des listes, il faut aussi prendre en compte le nombre d'opérations de la fonction append ! de julia.)

Nous avons comme fichiers de test, deux oeuvres : Cyrano de Bergerac et Le petit Prince. Les résultats sur ces deux instances sont les suivants :

3.1 Fonction de remplissage

Texte	Structure	Temps	Espace (allocations)
cyrano	arbre	0.029531 s	298.03 k allocations : 16.615 MiB
	tableau	0.088888 s	475.58 k allocations : 20.463 MiB, 11.61% gc time
le petit prince	arbre	0.013424 s	115.28 k allocations : 6.776 MiB
	tableau	0.059184 s	234.19 k allocations : 10.536 MiB

TABLE 2 – Tableau des tests de la fonction de remplissage passant du texte à un tableau ou un arbre

La fonction de remplissage de l'arbre est moins coûteuse en allocations. Cette structure de données a l'avantage de prendre moins d'espace mémoire et de ne pas nécessiter de garbage collector time. L'arbre est aussi plus rapide à produire que le tableau.

3.2 Recherche de mots présents dans l'arbre et le tableau

Texte	Mot	Méthode	Temps	Espace	Résultat
cyrano	À	arbre	0.000023 s		true
		tableau	0.000597 s	23.00 k allocations : 359.375 KiB	
	FÊTE	arbre	0.000137 s		false
		tableau	0.000978 s	35.00 k allocations : 546.875 KiB	
	FÊTES	arbre	0.000133 s		true
		tableau	0.000499 s	25.00 k allocations : 390.625 KiB	
	TARTELETTE	arbre	0.000272 s		true
		tableau	0.000667 s	33.00 k allocations : 515.625 KiB	
le petit prince	À	arbre	0.000025 s		true
		tableau	0.000532 s	29.00 k allocations : 453.125 KiB	
	FÊTE	arbre	0.000102 s		true
		tableau	0.000448 s	25.00 k allocations : 390.625 KiB	
	FÊTES	arbre	0.000129 s		false
		tableau	0.000503 s	27.00 k allocations : 421.875 KiB	
	TARTELETTE	arbre	0.000194		false
		tableau	0.000649	35.00 k allocations : 546.875 KiB	

TABLE 3 – Tableau des tests de la fonction de présence de mots dans un texte

Le temps de recherche de la présence du mot est le plus faible avec un arbre, pour toute taille de mot. Le temps de recherche dans un tableau est deux à trois fois supérieur mais, cela reste dans un temps très faible, par rapport au fonctions suivantes. L'arbre a aussi l'avantage de ne pas prendre autant d'espaces qu'un tableau : une centaine de KiB pour chaque recherche dans un tableau.
Si l'on augmente la taille du fichier, il serait peut-être possible d'observer des écarts de temps plus marqués. En effet, l'arbre permet de descendre dans les noeuds directement pour reconstruire le mot en ne parcourant qu'une branche de l'arbre, contrairement au tableau qui nécessite de faire une recherche dichotomique.

3.3 Recherche de la longueur moyenne des mots

Texte	Structure	temps	espace (allocations)	résultats
cyrano	arbre	1.941261 s		4.28028
	tableau	0.506642 s	14.87 M allocations : 226.913 MiB, 3.16% gc time	
le petit prince	arbre	0.515998 s		4.06085
	tableau	0.194193 s		

TABLE 4 – Tableau concernant la longueur moyenne des mots dans un texte

La longueur moyenne des mots se retrouve plus facilement dans un tableau, parce que cela ne nécessite qu'un passage sur chaque case du tableau. Dans le cas d'un arbre, il est nécessaire de parcourir tous les

noeuds de l'arbre pour reconstruire les mots et leurs nombres d'apparitions puis de récupérer ensuite toutes les tailles des mots. Le nombre d'opérations est donc supérieur pour l'arbre.

3.4 Nombre de mots distincts

Texte	Structure	Temps	Résultats
cyrano	arbre	1.991831 seconds	5467
	tableau	0.000000 seconds	
le petit prince	arbre	0.559963 seconds	2355
	tableau	0.000000 seconds	

TABLE 5 – Tableau des tests du nombre de mots distincts dans un texte

La longueur du tableau étant stocké en mémoire, le nombre de mots distincts est donc très rapide d'accès (temps constant). À l'inverse, l'arbre nécessite un parcours dans toutes les feuilles et donc, plus il y a de feuilles, plus le temps augmente. Cette quantité de feuilles a tendance à augmenter selon la taille du texte, en particulier si le nombre de mots avec un même préfixe est différent.

3.5 Liste de Mots commençant par un préfixe donné

Dans cette partie (et les deux suivantes), les résultats ne sont pas affichés. Cependant, on notera que les résultats sont des listes contenant les mêmes mots mais pas nécessairement rangés dans le même ordre. Le tableau rend une liste triée contrairement à l'arbre. Pour vérifier les résultats, il a été nécessaire de trier la liste provenant de l'arbre.

Texte	Préfixe	Structure	Temps	Espace (allocations)
cyrano	L	arbre	0.122319 s	2.04 M allocations : 107.529 MiB, 11.62% gc time
		tableau	0.007886 s	377.00 k allocations : 7.172 MiB
	ÂM	arbre	0.000358 s	7.00 k allocations : 406.250 KiB
		tableau	0.001112 s	45.00 k allocations : 796.875 KiB
	FON	arbre	0.003041 s	80.00 k allocations : 4.135 MiB
		tableau	0.001149 s	50.00 k allocations : 906.250 KiB
	REPRÉSENT	arbre	0.004947 s	37.00 k allocations : 1.984 MiB, 67.99% gc time
		tableau	0.001255 s	44.00 k allocations : 796.875 KiB
le petit prince	L	arbre	0.037506 s	752.00 k allocations : 39.642 MiB, 17.54% gc time
		tableau	0.002586 s	148.00 k allocations : 2.777 MiB
	ÂM	arbre	0.000035 s	
		tableau	0.000600 s	33.00 k allocations : 578.125 KiB
	FON	arbre	0.001255 s	34.00 k allocations : 1.785 MiB
		tableau	0.000969 s	45.00 k allocations : 812.500 KiB
	REPRÉSENT	arbre	0.000680 s	15.00 k allocations : 828.125 KiB
		tableau	0.001009 s	40.00 k allocations : 718.750 KiB

TABLE 6 – Tableau des tests de la fonction retournant les mots avec un certain préfixe

La taille des préfixes ne semble pas avoir d'impact sur le temps pris : pour un même préfixe, une structure de données peut-être plus rapide sur l'un des textes mais moins bon sur l'autre. On peut supposer

que cela provient du nombre de branches à parcourir. Entre deux textes, le lexique n'est pas le même et donc, pour un même préfixe, il peut y avoir des sous-arbres plus fournies. Dans ce cas, le temps nécessaire devient plus important. Le nombre d'allocations varie en fonction du texte. Néanmoins, dans tous les cas, l'espace mémoire pris est inférieur avec une structure en tableau.

3.6 Liste de Mots finissant par un suffixe donné

Texte	Suffixe	Structure	Temps	Espace (allocations)
cyrano	LETTE	arbre	4.942784 s	64.91 M allocations : 2.919 GiB, 4.45% gc time
		tableau	0.227852 s	4.98 M allocations : 76.462 MiB, 1.81% gc time
	AL	arbre	4.727459 s	64.94 M allocations : 2.921 GiB, 4.52% gc time
		tableau	0.207198 s	5.03 M allocations : 79.361 MiB, 1.43% gc time
	S	arbre	4.940402 s	66.01 M allocations : 3.047 GiB, 5.04% gc time
		tableau	0.310152 s	8.06 M allocations : 238.098 MiB, 4.83% gc time
	Q	arbre	4.801536 s	64.91 M allocations : 2.919 GiB, 4.50% gc time
		tableau	0.170656 s	4.96 M allocations : 76.004 MiB, 1.87% gc time
le petit prince	LETTE	arbre	1.843743 seconds	29.03 M allocations : 1.303 GiB, 5.26% gc time
		tableau	0.082639 seconds	1.85 M allocations : 28.412 MiB, 1.77% gc time
	AL	arbre	1.837409 seconds	29.04 M allocations : 1.303 GiB, 5.10% gc time
		tableau	0.071352 seconds	1.86 M allocations : 28.778 MiB, 3.63% gc time
	S	arbre	1.881201 seconds	29.55 M allocations : 1.356 GiB, 5.60% gc time
		tableau	0.124459 seconds	3.15 M allocations : 95.001 MiB, 6.15% gc time
	Q	arbre	1.783175 seconds	29.03 M allocations : 1.303 GiB, 5.26% gc time
		tableau	0.068902 seconds	1.85 M allocations : 28.366 MiB, 1.55% gc time

TABLE 7 – Tableau des tests de la fonction retournant les mots avec un certain suffixe

La structure de l'arbre prend énormément de temps à retourner la liste de mots, en particulier pour des textes de plus grande taille comme Cyrano de Bergerac. Dans le cas du tableau, retrouver la liste de mots avec un même suffixe nécessite seulement un parcours de tableau. Or, dans le cas de l'arbre, il est nécessaire de parcourir toutes les branches et de construire toutes les solutions possibles afin de les tester et de les récupérer. Le temps nécessaire est donc beaucoup plus long et le garbage time énorme par rapport au tableau, pour n'importe quelle instance. Pour un même texte et qu'importe le suffixe, le temps pour chaque recherche dans l'arbre est similaire.

3.7 Nombre d'occurrences d'une lettre donnée

Structure	Texte	Lettre	Temps	Espace (allocations)	Résultats
cyrano	Â	arbre	1.824526 s		131
		tableau	1.659301 s	73.93 M allocations : 1.102 GiB, 2.21% gc time	
	E	arbre	1.939374 s		22735
		tableau	1.772380 s	78.52 M allocations : 1.170 GiB, 2.18% gc time	
	X	arbre	1.974165 s		1154
		tableau	1.707353 s	74.03 M allocations : 1.103 GiB, 2.16% gc time	
petit prince	Â	arbre	0.535990 s		32
		tableau	0.677904 s	26.93 M allocations : 410.843 MiB, 2.38% gc time	
	E	arbre	0.573226 s		9225
		tableau	0.752647 s	28.57 M allocations : 435.974 MiB, 1.94% gc time	
	X	arbre	0.551769 s		233
		tableau	0.669091 s	26.98 M allocations : 411.606 MiB, 1.99% gc time	

TABLE 8 – Tableau des tests de la fonction retournant le nombre d'occurrences d'une lettre dans un texte

La fréquence de la lettre importe peu dans les temps de calculs. Dans le cas d'un tableau, il est obligatoire de parcourir chaque lettre de chaque mot. Dans le cas d'un arbre, il est nécessaire de parcourir chaque noeud. Pour nos exemples, le tableau est meilleur en temps pour Cyrano et l'arbre est meilleur pour le petit prince. Cela signifie donc qu'il y probablement plus de répétitions de mots dans le petit prince et donc, qu'il y a moins de noeuds à parcourir. Il est important de remarquer que seul le tableau prend beaucoup d'espace mémoire.

3.8 Liste de Mots d'une taille donnée

Texte	Taille	Structure	Temps	Espace (allocations)
cyrano	2	arbre	0.052327 s	1.30 M allocations : 62.454 MiB, 8.68% gc time
		tableau	0.269252 s	5.09 M allocations : 85.114 MiB, 1.82% gc time
	4	arbre	0.940385 s	16.36 M allocations : 768.890 MiB, 6.26% gc time
		tableau	0.277955 s	5.80 M allocations : 129.868 MiB, 2.87% gc time
	6	arbre	2.846962 s	40.81 M allocations : 1.886 GiB, 4.99% gc time
		tableau	0.312665 s	6.95 M allocations : 216.858 MiB, 4.06% gc time
	8	arbre	4.173200 s	57.20 M allocations : 2.593 GiB, 4.61% gc time
		tableau	0.299343 s	6.43 M allocations : 173.065 MiB, 3.61% gc time
le petit prince	2	arbre	0.042836 s	1.05 M allocations : 49.393 MiB, 9.60% gc time
		tableau	0.096633 s	1.93 M allocations : 34.103 MiB, 3.56% gc time
	4	arbre	0.419668 s	8.93 M allocations : 418.442 MiB, 7.19% gc time
		tableau	0.101533 s	2.26 M allocations : 55.359 MiB, 3.30% gc time
	6	arbre	1.005271 s	19.17 M allocations : 898.376 MiB, 6.31% gc time
		tableau	0.116624 s	2.69 M allocations : 84.686 MiB, 3.76% gc time
	8	arbre	1.571460 s	25.81 M allocations : 1.169 GiB, 5.24% gc time
		tableau	0.108334 s	2.43 M allocations : 70.480 MiB, 4.43% gc time

TABLE 9 – Tableau des tests de la fonction retournant les mots avec un certain nombre de lettres

Pour des mots très courts, la structure d'arbre est utile afin de recupérer les mots de cette taille. Cependant, à partir de 4 lettres ou plus, le tableau devient intéressant au niveau du temps. De plus, l'espace mémoire pris par l'arbre est énorme par rapport au tableau pour toutes les recherches. Le temps de recherche pour un tableau reste similaire pour chaque recherche dans un texte.

4 Conclusion

Aucune des deux structures de données est toujours plus efficace que l'autre. La structure de données en arbre est plus performante en temps et en espace mémoire pour la fonction de remplissage, de la recherche d'un mot. Tandis que la structure de données en tableau est recommendable lors de la recherche : de la longueur moyenne des mots d'un texte, des mots finissant par un suffixe, ou du nombre de mots distincts. Pour la fonction recherchant la liste de mot selon un préfixe donné et pour la fonction donnant le nombre d'occurrences d'une lettre donnée, la rapidité varie selon le lexique présent dans le texte, puisque cela change le nombre de branche à parcourir dans l'arbre. Au niveau de l'espace mémoire, le tableau est le plus économe pour la liste de mots avec préfixe. Cependant, la fonction calculant le nombre d'occurrences d'un mot est plus coûteuse en espace avec la structure en tableau. Pour la fonction cherchant les mots d'une taille donnée, l'arbre n'est intéressant que pour sa rapidité à récupérer les mots de quelques lettres, sinon le tableau est plus rapide pour des mots plus longs et moins coûteux en espace mémoire dans tous les cas.

Ainsi, la structure de données en arbre est très utile lors de la manipulation de texte de très grande taille, en particulier dans le but de faire des recherches de données présentes dans un lieu précis de l'arbre, comme la présence d'un mot. La fonction de remplissage de l'arbre est peu coûteuse en temps et en espace, tout comme les fonctions de recherches de mot. Cependant, s'il faut fréquemment faire des recherches retournant une liste de mot, la structure de données en tableau est plus indiquée car, elle est plus économe en espace et en temps.

A Annexe

A.1 Structures

```
# Deux types de structures de données ( à lancer uniquement lors du premier
    appel dans le terminal)

mutable struct TableauMots
  nbMots::UInt64 # nombre total de mots
  nbMotsDistincts::UInt64 # nombre de mots différents les uns des autres
  mots::Array{String}
  decompte::Array{UInt64}
  function TableauMots()
    return new(0,0,Array{String,1}(),Array{UInt64,1}())
  end
end


mutable struct ArbreMots
  terminal::Bool
  nb::Int64
  suite::Dict{Char,ArbreMots}
  function ArbreMots()
    return new(false,0,Dict{Char,ArbreMots}())
  end
end
```

A.2 Main

```
#include("structures.jl")
include("prefixes_abr.jl")
include("prefixes_tab.jl")


function pretraitement(filePath)
    words = Vector{String}(undef,0)
    flux=open(filePath,"r")
    while ! eof(flux)
        ligne=readline(flux)
        append!(words,split(uppercase(ligne),vcat([c for c in "
            ?,.()/1234567890#~&{}[]|`%@*=+;:!-_<>'"],['$',’"’,'\t'])))
    end
    close(flux)
    return words
end



function main(fichier::String,numFonction::Int,mot = "TARTE",Nb = 3,lettre = '
    Â')

    println("Pour le texte : "*fichier)
    println()
    println("Remplissage de l'arbre")
      @time arb = remplissageArb(fichier)
    println("Remplissage du tableau")
      @time tab = remplissagetab(fichier)
    println()
    println()
```

```
    if numFonction == 1
        println("Recherche d'un mot présents dans l'arbre :" * mot)
      @time  for i in 1:1000 motPresent(arb,mot) end
        println(motPresent(arb,mot))
        println("Recherche d'un mot présents dans le tableau :" * mot)
      @time for i in 1:1000 motPresent(tab,mot) end
        println(motPresent(tab,mot))

    elseif numFonction == 2
        println("Recherche de la longueur moyenne des mots dans l'arbre")
          @time for i in 1:1000 longMoyenne(arb) end
        println("Moyenne :",longMoyenne(arb))
        println("Recherche de la longueur moyenne des mots dans le tableau")
        @time for i in 1:1000 longMoyenne(tab) end
        println("Moyenne : ",longMoyenne(tab))

    elseif numFonction == 3
        println("Nombre de mots distincts dans l'arbre")
        @time for i in 1:1000 NbmotsDistinct(arb) end
        println("Nb : ",NbmotsDistinct(arb))
        println("Nombre de mots distincts dans le tableau")
          @time for i in 1:1000 NbmotsDistinct(tab) end
        println("Nb : ",NbmotsDistinct(tab))


    elseif numFonction == 4
        prefixe = mot
        println("Liste de Mots commençant par (arbre): "*prefixe)
        println(ListeMotsDeb(arb,prefixe))
      @time for i in 1:1000 ListeMotsDeb(arb,prefixe) end
        println("Liste de Mots commençant par (tab): "*prefixe)
        println(ListeMotsDeb(tab,prefixe))
      @time for i in 1:1000 ListeMotsDeb(tab,prefixe) end


    elseif numFonction == 5
        suffixe = mot
        println("Liste de Mots finissant par (arbre): "*suffixe)
        println(ListeMotsFin(arb,suffixe))
      @time for i in 1:1000 ListeMotsFin(arb,suffixe) end
        println("Liste de Mots finissant par (tab): "*suffixe)
        println(ListeMotsFin(tab,suffixe))
      @time for i in 1:1000 ListeMotsFin(tab,suffixe) end

    elseif numFonction == 6
        println("Liste de Mots de taille (arbre): $(Nb)")
        println(NbMotsAvecNLettres(arb,Nb))
      @time for i in 1:1000 NbMotsAvecNLettres(arb,Nb) end
        println("Liste de Mots de taille (tab): $(Nb)")
        println(NbMotsAvecNLettres(tab,Nb))
      @time for i in 1:1000 NbMotsAvecNLettres(tab,Nb) end

    elseif numFonction == 7
        println("Nombre d'occurences de la lettre (arbre): "*lettre)
        println(NbOccurLettre(arb,lettre))
      @time for i in 1:1000 NbOccurLettre(arb,lettre) end
        println("Nombre d'occurences de la lettre (tab): "*lettre)
        println(NbOccurLettre(tab,lettre))
      @time for i in 1:1000 NbOccurLettre(tab,lettre) end

    else
        println("Seulement 7 fonctions de disponible alors choisissez un nombre
             entre 1 et 8 !")
    end

end
main("data/cyrano.txt",1)
```

A.3 Arbres

```
function remplissageArb(filePath)
        flux=open(filePath,"r")
        sep = vcat([c for c in "
             ?,.()/1234567890#~&{}[]|`%@*=+;:!-_<>'"],['$',’"’,'\t'])
        abr = ArbreMots()
        currentAbr = abr

        while ! eof(flux)
            ligne=readline(flux)
            li = split(uppercase(ligne),sep)
            for mot in li
                    i = 1
                    for lettre in mot
                      if !haskey(currentAbr.suite,uppercase(lettre))
                          currentAbr.suite[uppercase(lettre)] = ArbreMots()
                      end
                      currentAbr = currentAbr.suite[uppercase(lettre)]
                      if i == length(mot)
                          currentAbr.terminal = true
                          currentAbr.nb = currentAbr.nb + 1
                      end
                i = i + 1
                    end
                    currentAbr = abr
            end
        end
        close(flux)

        return abr
  end


function motPresent(abr,mot::String)
    existe = false
    k = 1
    i = 1
    currentAbr = abr
    while !existe && k <= length(mot)
  if isvalid(mot,i)
    lettre = mot[i]
    if haskey(currentAbr.suite,lettre)
        currentAbr = currentAbr.suite[lettre]
        if k == length(mot) && currentAbr.terminal == true
            existe = true
        end
    end
    k = k + 1
  end
  i = i + 1
    end
    return existe
  end


function ParcoursAbr(arb,taille)
    if isempty(arb.suite) && arb.terminal == true
           nomb = arb.nb
           sum = arb.nb * taille
    elseif arb.terminal == true
           sum = arb.nb * taille
           nomb = arb.nb
           for l in arb.suite
                n,t = ParcoursAbr(l[2],taille+1)
                sum = sum + t
                nomb = nomb + n
           end
    else
           sum = 0
           nomb = 0
           for l in arb.suite
```

```
                n,t = ParcoursAbr(l[2],taille+1)
                sum = sum + t
                nomb = nomb + n
            end
    end
    return nomb, sum
end


function longMoyenne(arb)
    taille = 0
    n,t = ParcoursAbr(arb,taille)
    if n == 0
        found = 0
    else
        found = t / n
    end
    return found
end




function ParcAbrD(arb)
    if isempty(arb.suite) && arb.terminal == true
        n = 1
    elseif arb.terminal == true
        n = 1
        for sousA in arb.suite
            n = n + ParcAbrD(sousA[2])
        end
    else
        n = 0
        for sousA in arb.suite
            n = n + ParcAbrD(sousA[2])
        end
    end
    return n
    end


function NbmotsDistinct(arb)
    n = ParcAbrD(arb)
    return n
    end


function ParcoursDeb(prefixe,arb,liste)
    if isempty(arb.suite) && arb.terminal == true
        append!(liste,[prefixe])
    elseif arb.terminal == true
        for sousA in arb.suite
            append!(liste,[prefixe])
            liste = ParcoursDeb(prefixe*sousA[1],sousA[2],liste)
        end
    else
        for sousA in arb.suite
            liste = ParcoursDeb(prefixe*sousA[1],sousA[2],liste)
        end
    end
    return liste
    end


function ListeMotsDeb(arb,prefixe)
    currentAbr = arb
    i = 1
    possible = true
    deb = prefixe
    while i <= sizeof(prefixe) && possible
  if isvalid(prefixe,i)
    if !haskey(currentAbr.suite,prefixe[i])
        possible = false
    else
```

```
        currentAbr = currentAbr.suite[prefixe[i]]
    end
  end
  i = i + 1
    end
    liste = Array{String,1}
    if possible
        liste = ParcoursDeb(prefixe,currentAbr,[])
    end
    return liste
    end

function testSuffixe(mot,suffixe)
  k = sizeof(mot) # ncodeunits
  i = sizeof(suffixe)
  possible = true
  while k > 0 && possible && i > 0
    if isvalid(suffixe,i) && isvalid(mot,k)
      possible = suffixe[i] == mot[k]
      k = k - 1
      i = i - 1
    elseif isvalid(suffixe,i) || isvalid(mot,k)
      possible = false
    else
      k = k - 1
      i = i - 1
    end
  end
  return possible
end

function ParcoursFin(arb,mot,suffixe,liste)
    if isempty(arb.suite) && arb.terminal == true
        if length(suffixe)<=length(mot) && testSuffixe(mot,suffixe)
            append!(liste,[mot])
        end
    elseif arb.terminal == true
        if length(suffixe)<=length(mot) && testSuffixe(mot,suffixe)
            append!(liste,[mot])
        end
        for sousA in arb.suite
            liste = ParcoursFin(sousA[2],mot*sousA[1],suffixe,liste)
        end
    else
        for sousA in arb.suite
            liste = ParcoursFin(sousA[2],mot*sousA[1],suffixe,liste)
        end
    end
    return liste
end


function ListeMotsFin(arb,suffixe)
    liste = ParcoursFin(arb,"",suffixe,[])
    return liste
end


function ParcoursNbLettre(nb,arb,liste,prefixe)
    if arb.terminal == true && nb == 0
        append!(liste,[prefixe])
    end
    if !isempty(arb.suite) && nb > 0
        for sousA in arb.suite
              liste = ParcoursNbLettre(nb-1,sousA[2],liste,prefixe*sousA[1])
        end
    end

    return liste
    end


function NbMotsAvecNLettres(arb,N)
  liste = ParcoursNbLettre(N,arb,[],"")
```

```
  return liste
end



function ParcoursOccur(arb,lettre,nbs)
  somme = 0
    if arb.terminal == true
  somme = somme + nbs * arb.nb
    end
    if !isempty(arb.suite)
        for sousA in arb.suite
    if sousA[1] == lettre
              somme = somme + ParcoursOccur(sousA[2],lettre,nbs+1)
    else
        somme = somme + ParcoursOccur(sousA[2],lettre,nbs)
    end
        end
    end
    return somme
    end

function NbOccurLettre(abr,lettre)
  return ParcoursOccur(abr,lettre,0)
end
```

A.4 Tableau

```
function remplissagetab(filePath::String)
    motss = sort!(pretraitement(filePath))

  tab = TableauMots()
  nb = size(motss,1)

  tab.decompte = zeros(UInt64,nb)
  tab.mots = fill(" ",nb)

  j = 1
  tab.mots[j] = motss[j]
  tab.decompte[j] = 1

  for i in 2:nb
    if tab.mots[j] != motss[i]
      j = j + 1
      tab.mots[j] = motss[i]
    end
    tab.decompte[j] = tab.decompte[j] + 1
  end
  if tab.mots[1] == ""
    tab.mots = tab.mots[2:j]
    tab.decompte = tab.decompte[2:j]
  else
    tab.mots = tab.mots[1:j]
    tab.decompte = tab.decompte[1:j]
  end
  tab.nbMots = sum(tab.decompte)
  tab.nbMotsDistincts = size(tab.decompte,1)
  return tab
end



function recpre(tab::TableauMots,e::String,deb,fin)
  i = deb + div(fin-deb,2)
  if tab.mots[i] == e
    return true
  elseif e < tab.mots[i] && deb<i
    return recpre(tab,e,deb,i-1)
```

```
  elseif tab.mots[i] < e && i<fin
    return recpre(tab,e,i+1,fin)
  else
    return false
  end
end

function motPresent(tab::TableauMots,e::String)
  return recpre(tab,e,1,size(tab.decompte,1))
end



function longMoyenne(tab::TableauMots)
  nb = size(tab.decompte,1)
  sum = 0
  nb2 = 0
  for i in 1:nb
    sum = sum + tab.decompte[i]*length(tab.mots[i])
    nb2 = nb2 + tab.decompte[i]
  end
  return sum/nb2
end


function NbmotsDistinct(tab::TableauMots)
    return tab.nbMotsDistincts
end


function testPrefixe(mot,prefixe)
  k = 1
  i = 1
  possible = true
  while possible && i <= sizeof(prefixe)
    if isvalid(prefixe,i) && isvalid(mot,k)
      possible = prefixe[i] == mot[k]
      k = k + 1
      i = i + 1
    elseif isvalid(prefixe,i) || isvalid(mot,k)
      possible = false
    else
      k = k + 1
      i = i + 1
    end
  end
  return possible
end


function RecupIndice(tab::TableauMots,prefixe::String,deb,fin)
  i = deb + div(fin-deb,2)
  if tab.mots[i] == prefixe
    return i
  elseif prefixe < tab.mots[i] && deb<i
    return RecupIndice(tab,prefixe,deb,i)
  elseif tab.mots[i] < prefixe && i<fin
    return RecupIndice(tab,prefixe,i+1,fin)
  elseif sizeof(prefixe) <= sizeof(tab.mots[i]) &&
      testPrefixe(tab.mots[i],prefixe)
    return i
  else
    return 0
  end
end



function ListeMotsDeb(tab::TableauMots,prefixe::String)
  f = RecupIndice(tab::TableauMots,prefixe,1,size(tab.decompte,1))
  if f != 0
    d = f
    while d <= size(tab.decompte,1) && testPrefixe(tab.mots[d],prefixe)
      d = d + 1
```

```
    end
    return tab.mots[f:d-1]
  else
    return []
  end
end



function testSuffixe(mot,suffixe)
  k = sizeof(mot)
  i = sizeof(suffixe)
  possible = true
  while k > 0 && possible && i > 0
    if isvalid(suffixe,i) && isvalid(mot,k)
      possible = suffixe[i] == mot[k]
      k = k - 1
      i = i - 1
    elseif isvalid(suffixe,i) || isvalid(mot,k)
      possible = false
    else
      k = k - 1
      i = i - 1
    end
  end
  return possible
end


function ListeMotsFin(tab::TableauMots,suffixe)
  liste = []
  for k in 1:size(tab.mots,1)
    if testSuffixe(tab.mots[k],suffixe) && length(tab.mots[k]) >=
        length(suffixe)
      append!(liste,[tab.mots[k]])
    end
  end
  return liste
end



function NbMotsAvecNLettres(tab::TableauMots,N)
  liste = []
  for k in 1:size(tab.mots,1)
    if length(tab.mots[k]) == N
      append!(liste,[tab.mots[k]])
    end
  end
  return liste
end



function NbOccurLettre(tab::TableauMots,lettre::Char)
  nb = 0
  for k in 1:size(tab.mots,1)
    for j in 1:sizeof(tab.mots[k])
      if isvalid(tab.mots[k],j) && (tab.mots[k])[j] == lettre
        nb = nb + tab.decompte[k]
      end
    end
  end
  return nb
end
```