Projet Encyclopédie : Compte Rendu

I. Les structures de données

a. Les listes chaînées

L'encyclopédie sous forme de listes chaînées consiste à créer différents maillons liés les uns à la suite des autres. Pour cela, un maillon est un pointeur de structure, caractérisé par un identifiant, un titre, un contenu et un pointeur qui pointe vers l'adresse du maillon suivant. Le dernier maillon d'une liste pointe donc vers une adresse nulle.

b. Les arbres binaires de recherche

Avec cette structure de données, on définit un maillon par un pointeur de structure caractérisé par un identifiant, un titre, un contenu et deux pointeurs. L'un pointera vers un maillon dont l'identifiant est plus petit que celui du maillon étudié, l'autre vers un maillon dont l'identifiant est plus grand.

Pour créer l'encyclopédie, on insère donc selon ces propriétés d'arbres binaires de recherche. Le premier article du fichier est la racine de l'arbre. Une feuille est un maillon dont ses noeuds fils sont nulles.

c. Les tables de hachage

Pour implémenter une encyclopédie sous forme de table de hachage, on crée dans un premier temps un tableau de listes chaînées vide.

Puis pour compléter ce tableau, on transforme les identifiants de chaque article grâce à une fonction de hachage, cette fonction consistant à retourner l'identifiant modulo la taille du tableau.

Enfin, on insère les articles à leur indice trouvé. S'il y a collision à cet indice, l'article est inséré récursivement à la fin de la liste chaînée déjà existante.

II. Gestion de la mémoire

a. La pile

Sur la pile sont stockés les variables locales, l'adresse de retour et une sauvegarde des paramètres en entrée d'une fonction. Lorsque l'on sort de cette fonction, les données qui lui sont associées sur la pile sont supprimées pour être remplacées par des valeurs aléatoires. La pile est un espace mémoire limité de quelques Mo.

Par cette définition, on peut donc en conclure que l'ensemble de nos variables locales, les paramètres en entrée et les adresses de retour de nos fonctions sont stockés sur la pile.

C'est pourquoi, suivant cette logique, il est dans notre intérêt de programmer de manière itérative la plupart de nos fonctions dans le cas des listes chaînées. En effet, si on souhaite par exemple insérer récursivement un élément à la fin d'une liste chaînée, on devrait alors parcourir l'intégralité de la liste. Or à chaque retour de fonction, on stocke des données sur la pile et puisque celle-ci est limitée dans l'espace, le programme échoue pour des listes très grandes.

Ce problème se pose moins pour les arbres binaires de recherche et pour les tables de hachage puisqu'on effectue une sélection au préalable, soit en utilisant les propriétés des arbres binaires de recherche, soit par hachage. On ne parcourra donc pas ces structures de données dans leur intégralité. C'est pour cette raison que l'on peut réaliser des fonctions en utilisant la récursivité pour des encyclopédies de tailles très grandes.

b. Le tas

L'ensemble des données allouées dynamiquement, dans notre cas par la fonction **malloc**, pendant l'exécution du programme sont contenus dans le tas.

Dans ce projet, malloc a été utilisé lors de la définition de maillons, que ce soit un maillon d'une liste chaînée ou d'un arbre binaire de recherche, mais également lors de la définition de la table de hachage.

Malloc ayant pour but d'allouer de l'espace pour une structure de données, on peut théoriquement allouer infiniment de l'espace aux pointeurs concernés. Or, ceci n'étant pas efficace du point de vue optimisation de l'espace, nous avons pour chaque pointeur alloué un espace de la taille de son type.

L'une des exceptions est le tableau de hachage, auquel on a multiplié la taille de son type par la taille du tableau puisqu'on a besoin d'allouer de l'espace pour chaque indice.

Les suivantes sont lors de l'utilisation de **strcpy()**. En effet, il est nécessaire de réserver au préalable dans la chaîne destinataire une taille supérieure ou égale à la taille de la chaîne copiée pour ne pas écraser une zone de la mémoire. C'est pourquoi lors de l'utilisation de celle-ci, on s'est assuré de définir une taille idéale grâce à **malloc** et "*sizeof(char)*strlen(titre)+1*" avec +1 pour le caractère '\0'.

A chaque malloc correspond un free du pointeur afin d'éviter les fuites mémoires.

c. Le .data et le .text

Le .data est vide puisque nous n'avons pas utilisé de variables globales.

Le .texte stocke l'ensemble de notre code.

III. Temps d'exécution (en secondes)

Pour le calcul du temps nécessaire à la réalisation des fonctions, on a utilisé la librairie *time.h*.

Pour cela nous avons utilisé le type de variable **clock_t** et la fonction **clock()** qui retournait dans une variable **clock_t** le temps actuel (en ticks de processeur). Enfin, pour obtenir le temps passé on réalise la différence des deux temps (fin - début), le tout divisé par **CLOCKS_PER_SEC**, valeur définie par la bibliothèque. Elle correspond au nombre de tick réalisés par le processeur en une seconde.

Par ailleurs, il était nécessaire de "caster" ces valeurs au type doubles afin d'obtenir des temps à la milliseconde.

Le type **clock_t** était préférable au type **time_t** car ce dernier se limitait au secondes. Au vu des valeurs obtenus, la différenciation au millième état très importante.

Pour ce qui est de la taille des tables de hachages, une plus grande taille de table permet d'avoir un accès direct à chaque donnée, soit un gain de temps pour ne pas accéder à une liste dans chaque case. Ainsi, on va donc observer différentes tailles intermédiaires.

Ainsi, on choisit des tailles qui donnent en moyenne par case 1 article [1] (tableau de 500.000), 10 articles [2] (tableau de 50.000) ou 100 articles [3] (tableau de 5.000).

a. Insertion des éléments dans la structure de données

Pour ce qui est de l'insertion on trouve les valeurs moyennes suivantes *(échantillons de 6 exécutions)* :

	Arbre	**Liste**	**Table de hachage [1]**	**Table de hachage [2]**	**Table de hachage [3]**
500 Test	0,024	0,015	0,012	0,015	0,013
500K Random	15,002	10,517	9,683	8,788	12,355
500K Sorted	**IMPOSSIBLE**	10,527	10,403	9,123	13,103

Le test est impossible pour les arbres pour les cas des 500.000 articles triés. En effet, pour ce qui est des arbres, il s'agit d'appels récursifs. La pile en mémoire ne supporte donc pas tous ces appels.

Les valeurs données pour le fichier de test de 500 articles est peu représentatif de implémentations différentes. En effet, on remarque sensiblement les mêmes valeurs.

Il semblerait que la table de hachage de taille 50.000 soit la plus optimisée pour l'insertion d'un grand nombre d'éléments. La répartition des articles dans le tableau doit donc se faire de manière homogène.

b. Suppression d'article

Pour la recherche par article, on choisit arbitrairement l'article suivant : 4491843 - Route du Fort-de-Gravelle. Cet article se trouve au milieu de la liste des articles (pour les articles rangés et aléatoirement réparti), et devrait donc se trouver au centre de toutes les implémentations. Les résultats devront donc être plutôt crédible et interprétable.

En pratique, il y a trop peu d'articles pour observer un écart de temps intéressant. En effet, les écarts observés sont de l'ordre de la milliseconde.

On trouve que la suppression d'articles est plus rapide pour l'arbre et pour la table de hachage de 5000 cases (47ms) contre les autres implémentations (78ms environ).

On retrouve donc ici un des avantages de l'arbre bien réparti. En effet, il est plus rapide dans ce cas d'accéder à une valeur souhaitée. Pourla table de hachage il faut juste trouver une taille qui serait plus optimale. La liste est elle peu représentative car influencée par le positionnement de l'article dans l'ordre d'insertion.

c. La recherche par article

Pour les mêmes raison, on prendra encore l'article 5468603 - Nerophis ophidion pour les tests. Les temps d'exécution de la suppression et de la recherche par article sont sensiblement les mêmes car ces fonctions réalisent le même parcours. Il n'y a comme seule différence les opérations réalisées par la suite. Respectivement une libération de l'espace mémoire alloué et une copie de l'article à retourner.

Les résultats de la recherche sont sensiblement les mêmes que pour la suppression. On note cependant que la taille intermédiaire pour la table de hachage est moins rapide que la grande taille. Cela s'explique par la répartition. En effet, comme il y a plus de cases, il y a plus de chance de trouver moins d'articles dans une case donnée, donc d'accélérer le processus.

Pour expliquer ces différences de temps, nous avons réalisé une analyse de la complexité des différents algorithmes de recherche : *(résultats théoriques)*

	Cas idéal	**Explication**	**Pire des cas**	**Explication**
Liste	n/2	Si le maillon cherché est celui au milieu de la liste	n	Parcourt la liste en entier si dernier maillon
Arbre	$\log_2(n)$	Si équilibré : n/2 à chaque itération donc $n/2^{p} \geq 1 \Leftrightarrow 2p \leq n$ $\Leftrightarrow plog(2) \square log(n)$ $\Leftrightarrow p \square log_{2}(n)$	n	Arbre dégénéré → devient une liste chaînée
Table de hachage	1 + n/2	Fonction de hachage + liste triée	1 + n	Fonction de hachage + parcourt la liste en entier si dernier maillon

d. La recherche par mot

La recherche par mot est assez similaire dans les 3 implémentations. En effet, le but est de parcourir la totalité des articles et copier dans une seconde encyclopédie ceux qui contiennent le mot désiré.

La recherche a été réalisée avec comme mots clefs "Arbre Binaire" (fichier 500K Random). Il est intéressant de noter que les 3 implémentations ont des temps quasi similaires (environ 0.65s à 0.7s). Cette fonction met donc en avant que pour parcourir la totalité des articles le temps d'exécution varie peu avec avec l'implémentation.

e. La destruction

Le cas de la destruction est sensiblement la même chose que la recherche par mot. Cette fois au lieu de réaliser des copies vers une autre encyclopédie, il suffit simplement de libérer l'espace mémoire.

On y observe le même résultat (pour les deux fichiers de 500K articles) que pour la recherche par mot clef; les temps sont quasi égaux (environ 2.5s). On arrive donc à la même conclusion qu'avant.

Cependant, pour le fichier trié, on observe des différences pour la table de hachage. En effet, plus la taille de la table est grande, moins la fonction met de temps à s'exécuter. (2,2s pour taille de 5000 contre 1,5s pour taille de 50.000 et 500.000). Cette différence est surprenant car les articles sont les mêmes, donc la disposition est logiquement identique une fois dans la table.

IV. Sources

Gestion de la mémoire : https://ilay.org/yann/articles/mem/
Complexité ABR : https://fr.wikipedia.org/wiki/Recherche_dichotomique