# *RAPPORT PROJET SE*

*WILHELM Daniel, YAMAN Kendal*

SOMMAIRE

# 1. INTRODUCTION

La finalité de notre TP est de créer une bibliothèque de fonctions pour accéder à une base de données qui a pour but de stocker des couples <clef, valeur > le plus optimisé possible.

La particularité de la base est de permettre un accès rapide quelle que soit sa taille.

C'est pourquoi nous nous sommes mis en binômes. Au début de notre projet nous avons commencé chacun de notre coté pour avancer et mieux comprendre ce projet.

# 2. REALISATION DU PROJET

La réalisation de notre projet s'est déroulé en plusieurs étapes :

- Dans un premier temps la compréhension était une étape essentielle pour aboutir à notre projet. Il nous a fallu beaucoup de temps pour comprendre le sujet. De plus nous comprenions mieux le sujet demandé au fur et à mesure que nous avancions dans notre projet.

- Une fois que nous avons compris le concept nous nous somme mis d'accord sur l'utilisation des fonctions et leurs utilités.

- De ce fait nous avons implémenté les fonctions nécessaire et auxiliaire (utile) pour pouvoir créer cette base de données : La création de la base, l'accès aux données, les méthodes d'allocations ainsi que les fonctions d'hachage.

- Une fois la base terminée nous avons vérifié si le notre implémentation passait bien les tests du professeur Loechner. Puis à notre tour nous avons créée nos propres tests unitaires pour mesurer la différence entre plusieurs codes possibles ainsi choisir la plus optimisée. Nous avons aussi vérifié la couverture de code pour vérifier le taux de code source testé d'un programme. Cela nous a permis de mesurer la qualité des tests effectués.

# 3. LE PROGRAMME ET PROBLEME RENCONTRE

## A. Le Programme

Dans notre programme nous avons bufférisé dkv afin de rechercher les offsets et longueurs de la base plus rapidement. Ainsi lorsque nous avons ajouté un nouvel emplacement il nous a suffit de rechercher un emplacement assez grand dans dkv. De plus, nous avons trié dkv. Ainsi lorsque nous voulions avoir des informations sur l'offset le plus grand ou lorsque nous voulions voir si les emplacements adjacents étaient libres il nous suffisait de regarder dans les bonnes cases de dkv qu'il ne fallait plus rechercher. Nous avons également essayé de minimiser le nombre de read et write en transmettant les blocs entre les fonctions. Une fois cela fait nous avons remarqué que la recherche de nouveau bloc avait besoin d'énormément de read. Nous avons donc également fait un buffer pour savoir si un bloc est vide ou non. Ainsi, il nous à suffit de regarder dans le tableau de buffer pour trouver un bloc vide et il ne faut donc plus lire chaque bloc un à un pour savoir s'il y a quelque chose d'écrit ou si le bloc est vide (nombre d'emplacements à 0).

N'ayant pas vu assez tôt qu'il ne fallait pas désallouer les blocs vides, nous désallouons également les blocs vides dans .h lorsque ces derniers n'ont pas de fils.

Nous écrivons dkv à chaque fermeture de fichier. Nous tronquons également .kv et dkv à la fin de chaque fichier. Par contre, ne sachant pas l'emplacement des blocs et des haches dans .blk et .h, nous ne tronquons pas ces deux fichiers.

## B. Problème rencontré

Nous avons rencontré plusieurs problèmes. Tout d'abord il a été difficile de bien saisir le sujet. Ensuite nous avions cru bien le comprendre et nous avons commencé à coder avant de nous apercevoir que nous avions mal saisit quelque chose. Nous avons donc dû recommencer des parties entières plusieurs fois. Nous avons également eu du mal à débugger tous le programme. Puis beaucoup de bugs sont apparus dans les différentes étapes et ils n'étaient pas toujours clairs. Nous avions donc des fois besoin de beaucoup de temps avant de comprendre d'où venait le bug après avoir recherché durant longtemps au mauvais endroit.

# 4. MESURE DES TYPES D'ALLOCATIONS ET FONCTION D'HACHAGE

Pour mener à bien notre projet ainsi que faire fonctionner nos fonctions essentielles (kv_open, kv_put…) de la base nous avons dû utiliser différentes fonctions auxiliaires. C'est pourquoi dans cette partie nous ne parlerons que de la mesure de

temps selon le type d'allocation choisis pour ces fonctions essentielles du point de vue optimal.

Il en est de même pour les fonctions de hachages.

## A. Les types d'allocations

Pour justifier quelle type d'allocation est la plus optimal nous avons mesuré plusieurs test d'ajout et de suppression de clés valeurs de manière aléatoire en prenant soin de modifier le type d'allocation. La mesure de temps d'exécution s'est faite à l'aide de la commande time. Bien sur nos tests mesure également le nombre d'emplacements.

Le but du test est de faire appel à la fonction mainrandom, qui a pour fonctionnalité de faire des puts et des dels de façon aléatoire. Nous faisons deux exécution par types d'allocation pour avoir plus de mesures, ainsi de précision. Puis nous aurons le temps d'exécution.

### i. First_fit

Le First fit choisit le premier emplacement disponible suffisamment grand.

Nous obtenons les mesures suivantes :

**Nombre d'emplacement à la première exécution : 17236**

**Nombre d'emplacement à la première exécution : 33241**

**FIRST FIT 66.15 secondes**

### ii. Worst_fit

Le Worst fit choisit l'emplacement disponible le plus grand.

Nous obtenons les mesures suivantes :

**Nombre d'emplacement à la première exécution : 16716**

**Nombre d'emplacement à la première exécution : 33881**

**WORST FIT 86.42 secondes**

### iii. Best_fit

Le Best fit choisit l'emplacement le plus petit parmi les emplacements suffisamment grand.

Nous obtenons les mesures suivantes :

**Nombre d'emplacement à la première exécution : 14917**

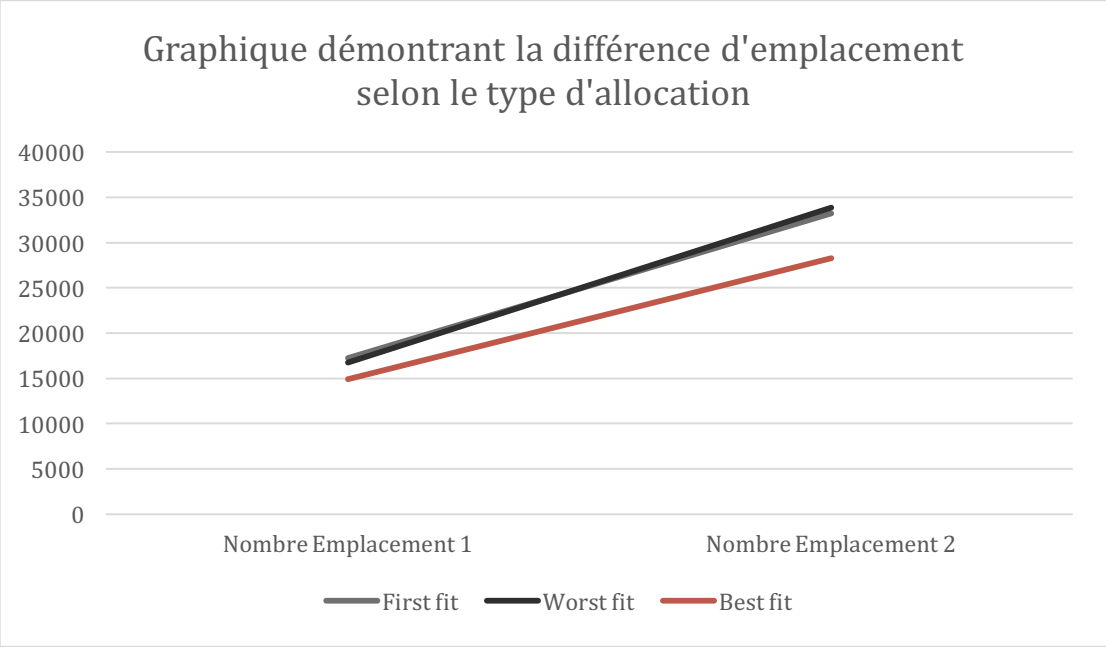
**Nombre d'emplacement à la première exécution : 28285**

**BEST FIT 86.42 secondes**

iv. Synthèse

Nous constatons qu'il y a plusieurs différences entre les différents type d'allocations. L'une des premières différences et le temps d'exécution. Le first fit est le type d'allocation le plus intéressant si on veut plus de performance en terme de vitesse que d'espace à utiliser. En effet les deux autre types d'allocations parcours tout le tableau alors que le first fit s'arrête au premier emplacement trouvé (c'est à dire suffisamment grand disponible).

Cependant en terme d'espace alloué le best fit est intéressant car selon notre test on remarque qu'il utilise beaucoup moi emplacement que les deux autres types d'allocations.




Comme on peut le voir sur ce graphe il y a une importante différence entre Best fit et les deux autres types d'allocations. Donc si on préférait miser sur une base qui économise de l'espace, la Best fit est plus conseiller.

Si on préfère minimiser le temps d'exécution alors le first fit est plus conseiller par rapport aux deux autres.

## B. Fonctions d'hachage

Nous avons fait 4 tests différents qui utilise chacune des fonctions de hachage qui sont : la fonction d'hachage par défaut, la fonction de Bernstein (en version modifié) qui est une amélioration mineure au hachage de Bernstein que l'on remplace la somme par un XOR pour l'étape de combinaison. Ainsi qu'une variante de la fonction de Bernstein modifié qui est le fvn_hach (Pour Fowler, Noll, Vo). C'est un algorithme puissant dont les constantes de hache de Bernstein ont été choisi de façon volontaire. Nous avons également fait une fonction de test qui retourne toujours le même résultat (0).

Pour plus de précision et fluidité, nous avons exécuté un même test au minimum 5 fois. Les valeurs qui seront inscrites seront les moyennes pour chaque fonction de hachage.

### i. Premier test

Dans ce test on entre 50000 clefs valeurs de taille comprise entre 0 et 10 ainsi que pour en supprimer de nouveaux 25000. Les résultats sont les suivants :

**Hache par défaut : 14.25 secondes.**

**Hache de Bernstein : 13.63 secondes.**

**Hache FVN : 13.63 secondes.**

Dans ce test, dont les temps d'exécution sont assez importants, on remarque que la fonction d'hachage proposé est moins performante que les deux fonctions que nous proposons qui sont tout les deux équivalentes. Cependant les temps d'exécution restent similaires. Cela est dû au fait que les clefs choisies sont des mots très aléatoire.

### ii. Deuxième test

Dans ce deuxième test nous refaisons le même test que le premier pour étudier si le temps d'exécution est toujours aussi différent entre ces fonctions. C'est pourquoi nous entrons cette fois 4000 clefs valeurs, de taille comprise entre 0 et 10, et ensuite pour en supprimer 5000. Nous obtenons les résultats suivants.

**Hache par défaut : .68 secondes.**

**Hache de Bernstein : 0,77 secondes.**

**Hache FVN : 0,77 secondes.**

Dans ce cas de figure on additionne toujours des zéros ascii donc 42 pour la somme de la fonction de hachage. Vu qu'il y a 4000 zéros au maximum et que

4000*42<1 000 000 la fonction de hachage par défaut est un peu plus performante que les autres fonctions.

iii. Troisième test

Dans le troisième test nous entrons 10000 clefs constitués de nombre entre 0 et 5000 aléatoirement, puis pour rentrer 10000 mots de longueur maximum 10 de manière aléatoire. On en supprimera 10000 d'entre elles. Voici les résultats :

**Hache par défaut : 4.72 secondes ;**

**Hache de Bernstein : 4.14 secondes ;**

**Hache FVN : 4.14 secondes ;**

Ici il y a une légère différence entre la fonction proposée et celle que nous proposons. Nos fonctions commencent à être plus performantes. C'est particulièrement le cas grâce au stockage des nombres entre 0 et 5 000 où il y a beaucoup de collision avec la première fonction de hachage (par ex : 21=12). Encore une fois nos deux fonctions ont le même temps d'exécution.

iv. Quatrième test

Pour ce dernier test qui sera basique, nous entrons 10000 clefs constituées de nombre entre 0 et 5000 et pour en supprimer de nouveau 5000.On a les résultats suivant :

**Hache par défaut : 2.47 secondes ;**

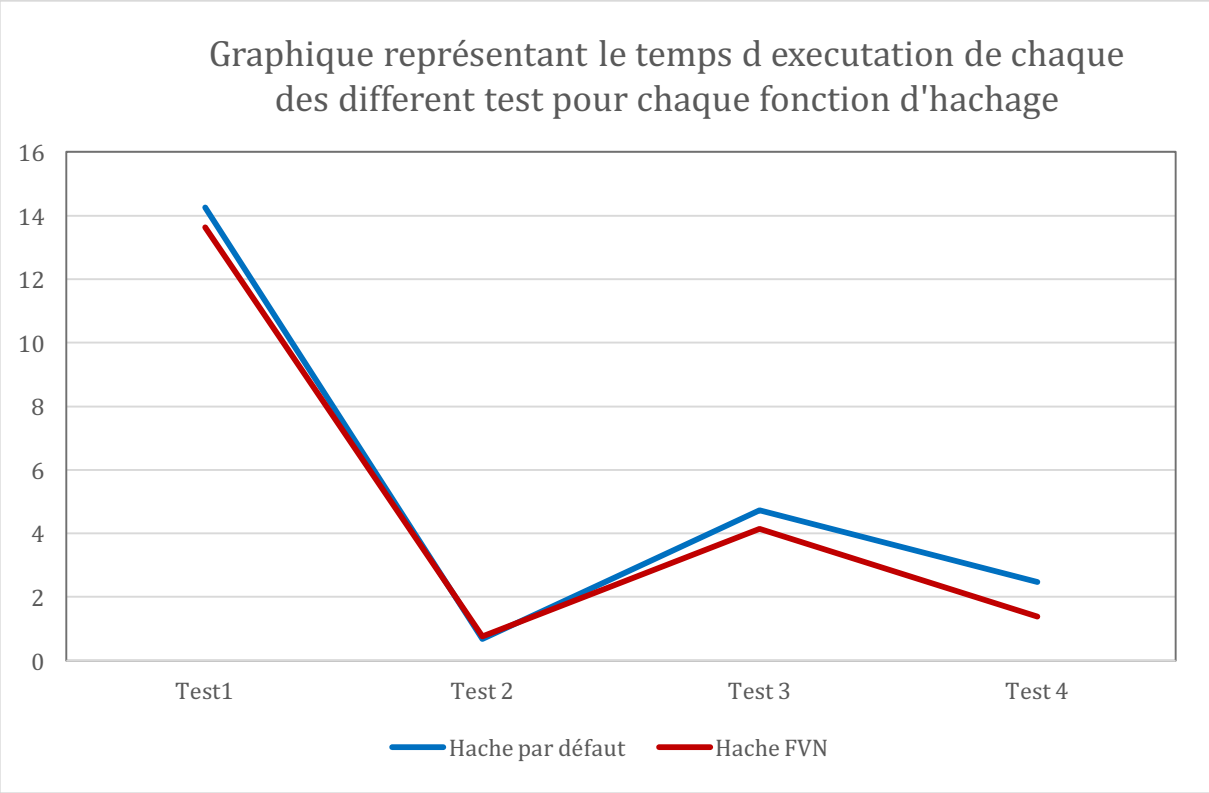
**Hache de Bernstein : 1.39 secondes ;**

**Hache FVN : 1.39 secondes ;**

Ici les différences sont les plus marquées. Ceci est dû au fait qu'on entre des int de 0 à 10 000 et qu'il y a donc beaucoup de collision pour la fonction hache_defaut pour les mêmes raisons que citées précédemment. (Hache_defaut (21) =hache_defaut (12)).

v. Synthèse

Pour résumer ces tests nous avons réalisé un graphique montrons la différence de chaque fonction (Comme pour nos deux fonctions nous avons les mêmes valeurs nous avons pris que la fonction FVN) temps selon le temps




En faisant des tests avec des valeurs plus grandes l'écart se creuse encore plus.

Mais nous pouvons aussi en déduire que la fonction de hachage à choisir dépend des valeurs que l'on veut stocker dans la base de donnée. Car si l'on veut par exemple stocker des zéros il est préférable d'utiliser la fonction de hachage par défaut. Pour les autres les fonctions que nous avons choisies sont soit équivalentes soit plus performantes selon les cas. Finalement on peut remarquer que d'habitude on stock des valeurs différentes dans des bases de données. Ou encore que l'on stock des mots et qu'il y a des ressemblances dans chaque langue et que les fonctions que nous avons choisies sont donc plus performantes.

## 5. TESTS UNITAIRES ET COUVERTURE DE CODE

### A. Tests unitaires

Afin de tester la base nous avons implémenté 4 fonctions main qui ajoutent toutes des couples différents. Nous avons fait un main qui ajoute des clefs et des valeurs contenant des chaines de caractères contenant des zéros (ex ''000''). Ce test ajoute des clefs de longueur comprise entre 0 et n-1 ainsi que des valeurs de longueur comprise entre 1 et n. Nous avons fait un deuxième main qui lui transforme les int de 0 à n en chaînes de caractères. (ex : 1 = ''1'') et qui ajoute ensuite à la base les couples < i, i+1>.

Dans notre troisième main nous avons fait un tableau de mots aléatoire de longueur comprise entre 0 et 10.

Dans notre quatrième main nous avons repris la 3ème et 2ème main. Nous les avons exécutés à nouveau. Puis nous avons fait un certain nombre d'ajout et de suppression de façon aléatoire.

### B. Couverture de code

Grâce à gcov nous avons pu repérer de lignes de codes qui n'étaient jamais exécutée et que nous avons donc supprimé. Gcov nous a également permit de débugger notre programme lorsque nous faisions des tests avec une grosse quantité de valeurs à ajouter. Nous avons obtenu une couverture de code avec la fonction maingcov qui couvre quasiment tous le code. Cependant il reste beaucoup de lignes non exécutée qui sont des lignes seulement exécutées en cas d'erreur.

## 6. CONCLUSION

### A. Bilan général

Bien que nous ayons eu un temps important pour rendre le projet qui nous était proposé nous avons pris du retard non seulement sur la compréhension du sujet et de la structure à adopter mais aussi sur l'implémentation du code.

Au début de notre projet nous avons eu du mal à nous accorder sur l'utilité de certaine fonction, et aussi de l'implémentation de certaine structure (tel que kv).

Puis nous avons pris conscience que certaine fonction auxiliaire nous étaient utile (comme l'utilisation de read_control).

Ce projet nous a été très formateur, par rapport à notre organisation mais aussi du point de vue technique tels que l'utilisation d'outils comme github qui nous a permis de travailler à distance sur un même fichier.

Concernant le programme nous avons, au début du projet, fournis un nombre important de fonctions auxiliaire qui nous étaient nécessaire, puis nous avons remarqué l'inutilité de certaine fonction et aussi la redondance dans certaine fonction, de ce fait on a réduit au maximum le nombre de fonction auxiliaire et implémenté autrement nos différentes structures pour justement évité des redondances.

Suivant les divers tests, nous avons aussi modifié à mainte reprise nos fonctions afin d'améliorer le temps d'exécution et avoir une meilleure stabilité au niveau de l'espace mémoire.

## B. Bilan Personnel

### i. WILHELM Daniel

Je trouve que ce projet a été formateur. Il m'a permit de comprendre l'importance de l'organisation préalable avant de coder pour mieux gérer l'interaction entre les différentes fonctions car coder et tout garder dans la tête est devenu très vite compliqué et m'a conduit à certains moment à faire n'importe quoi. Cela m'a alors obligé de relire plusieurs fois le code. La phase débuggage m'a permis de mieux connaître et reconnaître les bugs qui peuvent arriver en écrivant aux mauvaises positions ou en lisant/écrivant aux mauvais endroits dans des fichiers. C'est la phase de débuggage qui nous a posé le plus de problème car avec l'écriture dans les fichiers il a pas toujours été facile de repérer d'où viennent les bugs qui sont parfois apparus qu'une fois la base rouverte. Mais finalement nous avons réussi à éliminer les différents bugs.

Nous avons également appris à nous partager le travail qui a été conséquent. Nous avons appris qu'il faut bien spécifier ce que fait chaque fonction pour pouvoir travailler en groupe sur un projet avec autant de fonctions. Pour conclure, je trouve que le projet a été formateur et qu'il m'a surtout permis de gagner de l'expérience et de coder plus rapidement.

### ii. YAMAN Kendal

D’un point de vue extérieur, j’ai trouvé ce projet très formateur et très enrichissant au niveau de l’organisation. Cela a été beaucoup de travail et nous avons dû nous investir beaucoup pour pouvoir finir le projet. Les tâches que nous nous sommes donnés était clair et explicite. Nous avons eu beaucoup de mal à trouver les petites erreurs une fois le code écrit. C'est la phase de débuggage qui nous a prit le plus de temps.

Du point de vue technique, cela a été difficile de bien partager qui fait quoi et notamment s'assurer que les fonctions fassent et renvoient ce qu'il faut. Nous nous avons également partagé l'élaboration des différents programmes de test. J’ai aussi participé aux codes de chacune des fonctions de hachage.

De plus j’ai pu remarquer que les moments les plus long étaient lorsque nous débuguions, mais cela m’a permis de progresser plus vite lorsque l’on débuguait.