Rapport Mini-Projet:
Gestion d’une bibliothèque (en langage C)

Réalisé par:

- Sarah Beldjoudi
- Mélissa Chikbouni
- Alicia Achat

Présentation du mini-projet:

Le projet "Gestion d'une Bibliothèque" explore l'utilisation des listes chaînées et des tables de hachage pour organiser les données bibliographiques. Il inclut l'implémentation de fonctions de base comme la création, l'insertion et la suppression de livres, ainsi qu'une comparaison des performances entre les deux approches.

Partie 1: Gestion d'une bibliothèque avec une liste chaînée de struct.

```c
typedef struct livre {
    int num ;
    char * titre ;
    char * auteur ;
    struct livre * suiv ;
} Livre ;

typedef struct {
    Livre * L ;
} Biblio ;
```

Fichiers utilisés: `biblioLC.c entreeSortieLC.c entreeSortieLC.h biblioLC.h GdeBiblio.txt main.c`

Fonction implémentées:

```c
Livre* creer_livre(int num,char* titre,char* auteur);
void liberer_livre(Livre* l);
Biblio* creer_biblio();
void liberer_biblio(Biblio* b);
void inserer_en_tete(Biblio* b,int num,char* titre,char* auteur);
void afficher_livre(Livre *livre);
void afficher_biblio(Biblio *biblio);
Livre * recherche_ouvrage_num(Biblio *b,int n);
Livre * recherche_ouvrage_titre(Biblio *b,char* t);
Biblio * recherche_ouvrage_auteur(Biblio *b,char* a);
void supp_ouvrage(Biblio *b, int num, char *titre, char *auteur);
void fusion(Biblio *b1, Biblio *b2);
Livre * exemplaire(Biblio *b);
Biblio* charger_n_entrees(char* nomfic, int n);
void enregistrer_biblio(Biblio *b, char* nomfic);
```

Exécution de la fonction main: `./main <GdeBiblio.txt> <nombre de lignes>`

Partie 2: Gestion d'une bibliothèque avec une liste table de hachage.

```c
typedef struct livreh {
    int clef;
    int  num;
    char* titre;
    char* auteur;
    struct livreh* suivant;
} LivreH;

typedef struct table {
    int nE;
    int m;
    LivreH** T ;
} BiblioH;
```

Fichiers utilisés: `biblioLH.c entreeSortieLH.c entreeSortieLH.h biblioLH.h GdeBiblio.txt mainH.c`

Fonctions implémentées :

```c
int fonctionClef(char* auteur);
LivreH* creer_livre_H(int num, char* titre, char* auteur);
void liberer_livre_H(LivreH* l);
BiblioH* creer_biblio_H(int m);
void liberer_biblio_H(BiblioH* b);
int fonctionHachage(int cle, int m);
void inserer_H(BiblioH* b, int num, char* titre, char* auteur);
void afficher_biblio_H(BiblioH* b);
LivreH* recherche_par_numero_H(BiblioH* b, int num);
LivreH* recherche_par_titre_H(BiblioH* b, char* titre);
BiblioH * recherche_par_auteur_H (BiblioH* b, char* auteur);
void suppression_ouvrage_H (BiblioH* b, int num , char * auteur , char * titre) ;
void fusion_biblios_H(BiblioH* b1, BiblioH* b2);
BiblioH* recherche_exemplaires_multiples_H(BiblioH* b) ;
```

Partie 3: **Comparaison des deux structures**

Fichiers utilisés : `test.c bibliotheque_enregistree.txt GdeBiblio.txt result_liste.txt result_table.txt courbe_livre_inexistant_auteur.png courbe_livre_inexistant_titre.png courbe_livre_inexistant_num.png courbe_livre_milieu_num.png courbe_livre_inexistant_taille_table.png courbe_livre_milieu_taille_table.png`

Fonctions implémentées :

```
void comparer_temps_recherche(int repetitions)
void comparer_temps_taille()
void comparaison_taille ()
```

➔ **Comparaison du temps de calcul pour la recherche d'un ouvrage par son numéro en utilisant la fonction de signature `void comparer_temps_recherche(int repetitions)` par numéro , titre et auteur pour les deux structures :** nous avons constaté que la table de hachage était la plus appropriée :

```
Temps moyen de recherche avec une liste chaînée : 0.000002 secondes
Temps moyen de recherche avec une table de hachage : 0.000001 secondes
La table de hachage est plus appropriée pour cette recherche.
```

Explications :

- **Par auteur:**

Dans le cas d'une recherche par auteur :
1) Si l'élément est présent : la table de hachage serait généralement plus appropriée. En utilisant auteur comme clé de hachage, la recherche peut être effectuée en identifiant l'indice ou se trouve la liste qui contient les livre d'un auteur et de la parcourir . En revanche, dans une liste chaînée, la recherche par numéro nécessiterait de parcourir toute la liste, ce qui prendrait un temps proportionnel à la taille de la liste, soit O(n) .
2) Si l'élément n'est pas présent, la liste chaînée est mieux.

- **Par titre et numéro:**

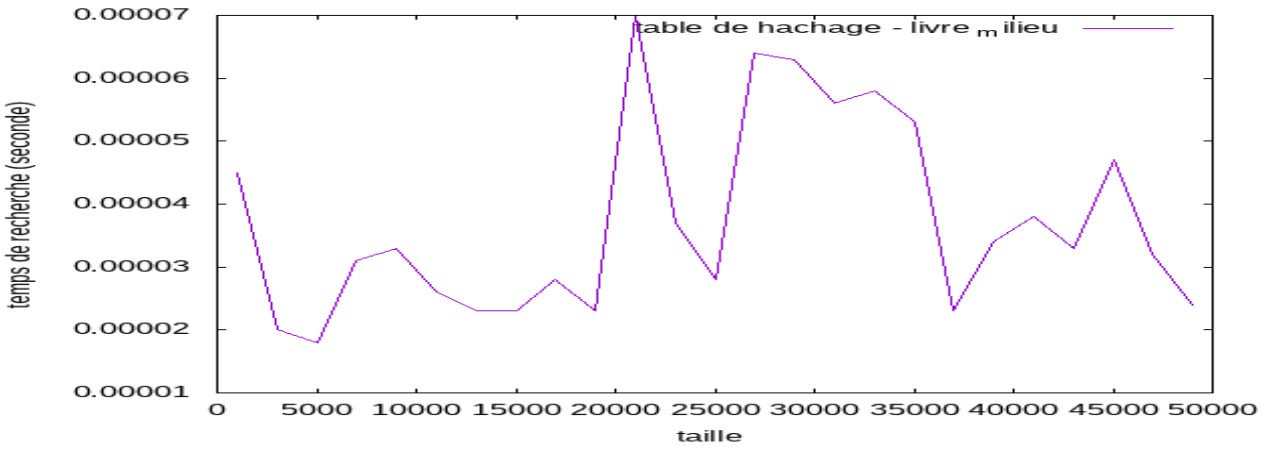
• Dans le cas de la liste chaînée, une recherche linéaire devrait être effectuée, ce qui implique un temps proportionnel à la taille de la liste, encore une fois O(n) dans le pire des cas.
• Dans une table de hachage, la performance de la recherche dépend de la répartition des éléments s'ils sont repartis de manière uniforme dans la table,

cela permettrait une recherche en temps constant en moyenne par contre si on a des collisions fréquentes, cela ralentirait les performances de recherche.

➔ **Comparaison du temps de calcul avec tailles variables pour la recherche d'un ouvrage par son numéro en utilisant la fonction de signature**

`void comparaison_taille() :`

On a tracé les variations du temps de recherche par numéro en fonction de la taille de la table de hachage avec un nombre fixe d'éléments.



Il semble y avoir une tendance générale à la baisse des temps de recherche jusqu'à une certaine taille de table de hachage (environ 18 000), suivie d'une tendance à la hausse. Cela suggère que pour des tailles de table de hachage plus petites, l'efficacité de recherche s'améliore avec l'augmentation de la taille
de la table, mais au-delà d'une certaine taille, les performances commencent à se dégrader Cela pourrait indiquer des problèmes spécifiques à ces tailles de table de hachage, comme une répartition des données moins efficace ou des collisions plus fréquentes.

➔ **Détermination les temps de recherche des ouvrages en plusieurs exemplaires en fonction de la taille de la bibliothèque et de la structure de données utilisée en utilisant la fonction de signature**

`void comparer_temps_taille():`

- **La recherche par titre (photo1) et numéro(photo 2):**

Les temps de recherche pour la liste chaînée est constamment inférieure à celle de la table de hachage pour la recherche par titre et numéro , cela suggère que la table de hachage n'est pas aussi efficace que prévu dans ce contexte spécifique cela est peut être du au fait que la fonction de hachage utilisée pour la table de hachage ne répartit pas efficacement les éléments de manière uniforme dans la table, cela peut entraîner des collisions fréquentes et des

performances réduites.

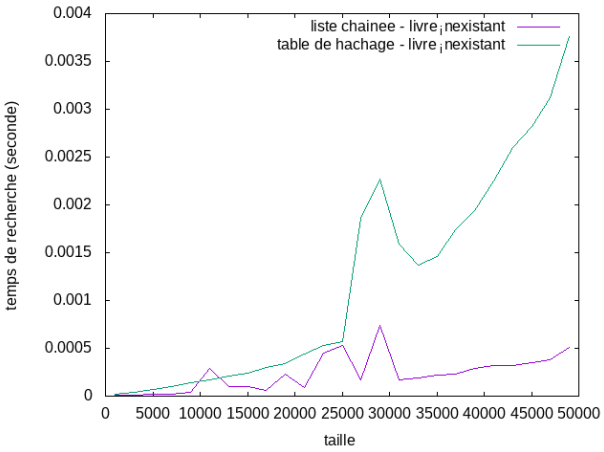


photo 1

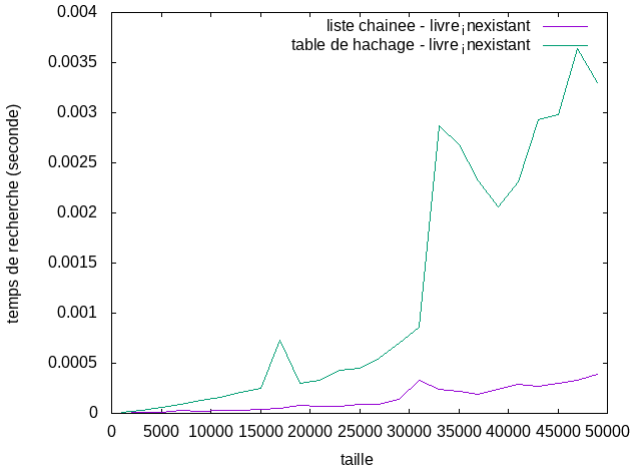
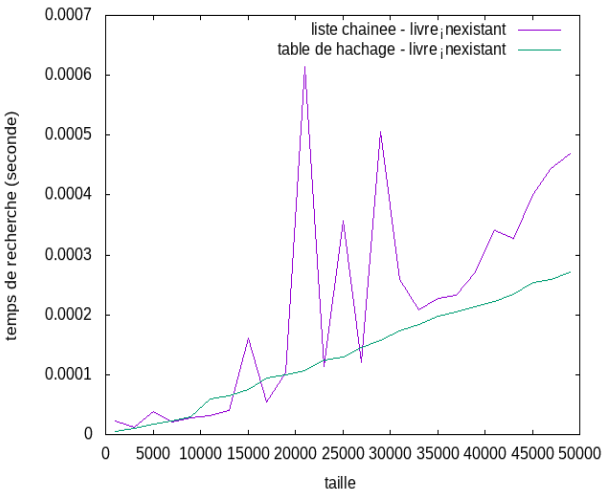


photo 2

- **La recherche par auteur:**

La courbe des temps de recherche pour la recherche par auteur (voir photo 2) montre que la table de hachage est en dessous de la liste chaînée, cela indique que la table de hachage est plus efficace dans ce cas particulier, la recherche est effectuée efficacement en identifiant l'indice où se trouve la liste contenant les livres de cet auteur spécifique dans la table de hachage. Ensuite, il suffit de parcourir cette liste pour récupérer tous les livres de cet auteur. Cela permet d'éviter le parcours de tous les éléments de la bibliothèque comme dans le cas de la liste chaînée pour une recherche par auteur.



Qualité du code:

`valgrind ./main GdeBiblio.txt 100`

```
==4805==
==4805== HEAP SUMMARY:
==4805==     in use at exit: 0 bytes in 0 blocks
==4805==   total heap usage: 310 allocs, 310 frees, 16,506 bytes allocated
==4805==
==4805== All heap blocks were freed -- no leaks are possible
==4805==
==4805== For lists of detected and suppressed errors, rerun with: -s
==4805== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

`valgrind ./test`

```
sarah@sarah-HP-Pavilion-Laptop-14-dv1xxx:~/projet$ valgrind ./test
==4894== Memcheck, a memory error detector
==4894== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4894== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==4894== Command: ./test
==4894==
Temps moyen de recherche avec une liste chaînée : 0.000002 secondes
Temps moyen de recherche avec une table de hachage : 0.000002 secondes
La table de hachage est plus appropriée pour cette recherche.
==4894==
==4894== HEAP SUMMARY:
==4894==     in use at exit: 0 bytes in 0 blocks
==4894==   total heap usage: 7,567,030 allocs, 7,567,030 frees, 151,724,110 byte
s allocated
==4894==
==4894== All heap blocks were freed -- no leaks are possible
==4894==
==4894== For lists of detected and suppressed errors, rerun with: -s
==4894== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```