Algorithmique et Programmation : Mise à Niveau

Rapport

Projet : Vérificateur Orthographique

Étudiants :
KERKOFF LADEIRA Julia
MACHADO ALMEIDA Pedro Henrique
STELTER FENZKE Caio

Filière : Systèmes Embarqués et Objets Connectés

Grenoble, 11 décembre 2024

Table des matières

1 Introduction

Le projet consiste à développer un vérificateur orthographique en langage C, capable de vérifier l'orthographe d'un texte en le comparant à un dictionnaire. À partir d'un fichier dictionnaire, le programme devra identifier et signaler les mots qui ne correspondent pas aux entrées du dictionnaire, permettant ainsi de détecter les fautes d'orthographe dans un texte donné. Ce projet vise à explorer différentes structures de données et méthodes de recherche pour réaliser cette tâche de manière efficace, tout en prenant en compte les performances en termes de rapidité et d'empreinte mémoire.

1.1 Objectif du Projet

L'objectif principal de ce projet est de programmer un vérificateur orthographique performant en C. Le programme doit être capable de :

- Construire un dictionnaire à partir d'un fichier texte donné ;
- Analyser un texte et identifier les mots qui ne font pas partie du dictionnaire, en les affichant comme des erreurs potentielles ;
- Utiliser différentes structures de données pour l'implantation du dictionnaire et la recherche des mots :
  - Tableau redimensionnable et recherche séquentielle ;
  - Tableau redimensionnable avec tri et recherche dichotomique ;
  - Ensembles ;
  - Arbres lexicographiques ;
  - Tries ;
- Comparer les performances de ces structures de données en termes de rapidité et d'empreinte mémoire, et analyser la complexité théorique et expérimentale des solutions proposées.

2 Mode d'Emploi et Utilisation du Programme

2.1 Compilation et Installation

Le projet comprend cinq tests principaux, chacun utilisant une structure de données différente pour la gestion du dictionnaire et la recherche dans un texte. Avant d'exécuter les tests, vous devez compiler les fichiers source avec `gcc`. Assurez-vous que

toutes les commandes sont exécutées depuis le dossier `projet`. Voici les étapes pour chaque test :

— **Test linéaire avec vecteur dynamique** (`testDicoVect.c`) :

```
make vect
```

— **Test de recherche binaire avec tri préalable** (`testDicoBinarySearch.c`) :

```
make binarySearch
```

— **Test avec *HashSet*** (`testDicoHashlset.c`) :

```
make hashlset
```

— **Test avec Trie** (`testDicoTrie.c`) :

```
make trie
```

— **Test avec Arbre Patricia** (`testDicoPatricia.c`) :

```
make patricia
```

Une fois compilés, les exécutables sont créés dans le dossier `output/`.

2.2 Exécution des Commandes

Pour exécuter les tests, utilisez les commandes suivantes dans le dossier `projet`. Chaque test prend deux fichiers en paramètre : le fichier dictionnaire (`dico1.txt`) et le texte à vérifier (`a_la_recherche_du_temps_perdu.txt`).

— **Test linéaire avec vecteur dynamique** :

```
make run-dico-vect
```

— **Test de recherche binaire avec tri préalable** :

```
make run-dico-binarySearch
```

— **Test avec HashSet** :

```
make run-dico-hashlset
```

— **Test avec Trie** :

```
make run-dico-trie
```

— **Test avec Arbre Patricia** :

```
make run-dico-patricia
```

2.3 Vérification avec Valgrind

Pour garantir l'absence de fuites de mémoire et de mauvais accès à la mémoire, utilisez l'outil `Valgrind`. Chaque test peut être exécuté avec la commande suivante dans le dossier `projet` :

```
valgrind --leak-check=full ./output/<nom_du_test> dico1.txt \
a_la_recherche_du_temps_perdu.txt
```

Par exemple, pour tester `testDicoTrie.c`, exécutez :

```
valgrind --leak-check=full ./output/testDicoTrie dico1.txt \
a_la_recherche_du_temps_perdu.txt
```

2.4 Nettoyage des Fichiers Générés

Pour nettoyer les fichiers générés, y compris les exécutables dans `output/`, vous pouvez utiliser la commande suivante dans le dossier `projet` :

```
make clean
```

Cette commande supprimera tous les fichiers du dossier `output/`, permettant une nouvelle compilation propre.

3 Architecture du Projet

3.1 Structure du Projet

Le projet a été conçu et structuré de manière collaborative, chaque membre de l'équipe prenant en charge une partie des travaux. Les modules (*TDs*) ont été implémentés en suivant une répartition claire des responsabilités, afin d'assurer une progression harmonieuse et une intégration efficace des différentes parties du code.

3.2 Organisation de l'Équipe

L'équipe était composée de trois membres, chacun ayant des responsabilités spécifiques, tout en travaillant de manière collaborative pour garantir le succès du projet.

— KERKOFF LADEIRA Julia : Responsable de l'implémentation des parties suivantes :

  — *Tableaux dynamiques et Généricité* ;

  — *Tableaux redimensionnables* ;

  — *Tas.*

  Ces parties incluaient la gestion de structures fondamentales et dynamiques, en particulier le développement des tas, essentiels pour l'efficacité des algorithmes de traitement des données.

— STELTER FENZKE Caio : Responsable de l'implémentation des parties suivantes :

  — *Listes et Listes génériques* ;

  — *Piles et Files* ;

  — *Algorithmes de tri efficaces.*

  Ces modules comprenaient des structures critiques pour le tri, ainsi que la gestion des listes et piles, qui sont des composants essentiels de nombreuses fonctionnalités du projet.

— MACHADO ALMEIDA Pedro Henrique : Responsable de l'implémentation des parties suivantes :

  — *Ensembles, Tables de hachage* ;

  — *Structures Arborescentes lexicographique* ;

  — *Arbre Patricia.*

Pedro Henrique a également pris en charge l'unification de l'ensemble du travail, l'ajustement des parties les plus complexes, et a joué un rôle clé en fournissant une assistance dans les modules développés par les autres membres, assurant la cohérence et la qualité globale du projet.

Chaque membre a également contribué à la rédaction de la partie du rapport correspondant à ses contributions techniques, garantissant une documentation fidèle et détaillée du travail réalisé.

4 Implémentation des Structures de Données

4.1 Tableaux dynamiques et Généricité

Les tableaux dynamiques offrent une solution flexible pour gérer des ensembles de données de taille variable. Contrairement aux tableaux statiques, leur taille peut être ajustée dynamiquement selon les besoins, ce qui optimise l'utilisation de la mémoire et permet une meilleure adaptabilité des programmes.

En C, la généricité peut être mise en œuvre de deux façons principales :

— **Généricité par pointeurs** : Cette approche utilise le type `void*` pour manipuler des pointeurs génériques. Elle permet de créer des tableaux capables de contenir des données de tout type, mais exige une gestion manuelle des conversions et de la libération de mémoire, augmentant ainsi les risques d'erreurs.

— **Généricité par génération de code** : Avec cette méthode, des macros et des préprocesseurs génèrent des structures et fonctions spécifiques pour chaque type de données. Cela garantit une vérification stricte des types à la compilation et minimise les erreurs d'exécution.

Pour ce projet, nous avons privilégié l'approche de la généricité par pointeurs, car elle offre une grande flexibilité en permettant de manipuler des données de tout type à travers des pointeurs génériques de type `void*`. Cette méthode est particulièrement utile dans des situations où la réutilisation du code est essentielle et où le type exact de données à manipuler peut varier dynamiquement, tout en étant plus simple à implémenter que la généricité par génération de code.

4.2 Tableaux redimensionnables

Les tableaux redimensionnables implémentés utilisent des techniques avancées pour gérer dynamiquement leur taille, combinant flexibilité et efficacité. Voici les points principaux de leur conception.

Création et allocation dynamique La fonction `vect_new` initialise un tableau avec une capacité donnée et alloue dynamiquement la mémoire requise. Lorsque la capacité maximale est atteinte, le tableau est redimensionné en doublant sa taille grâce à la fonction `realloc`, comme illustré ci-dessous :

```
vect_t vect_append(void* e, vect_t table) {
    if (table->actual_size == table->max_size - 1) {
        table->max_size *= 2;
        table->data = realloc(table->data, table->max_size *
        sizeof(*(table->data)));
    }
    table->data[table->actual_size++] = e;
    return table;
}
```

Principales opérations

- **Ajout** (`vect_append`) : Ajoute un élément à la fin du tableau. Si la capacité maximale est atteinte, le tableau est redimensionné dynamiquement.
- **Recherche** (`vect_find`) : Recherche un élément en parcourant le tableau. Une fonction de comparaison est utilisée pour gérer les types de données.
- **Suppression** (`vect_remove_at`) : Supprime un élément à une position donnée et réorganise les indices restants. Si la taille actuelle est inférieure à un tiers de la capacité, le tableau est redimensionné pour économiser de la mémoire.
- **Libération** (`vect_delete`) : Libère la mémoire allouée pour le tableau et ses éléments, en utilisant une fonction spécifique de suppression.

Exemple d'utilisation Un exemple typique de tests aléatoires montre comment manipuler ces tableaux :

```
vect_t t1 = vect_new(2, double_fprintf, double_delete, equal_double);
```

```
for (int j = 0; j < 100; j++) {
    switch (random() % 3) {
        case 0:
            // Suppression d'un élément
            int pos = random() % t1->actual_size;
            t1 = vect_remove_at(pos, t1);
            break;
        case 1:
            // Recherche d'un élément
            double val = random() % 100;
            int found = vect_find(&val, t1);
            break;
        case 2:
            // Ajout d'un élément
            double *new_val = malloc(sizeof(double));
            *new_val = random() % 100;
            t1 = vect_append(new_val, t1);
            break;
    }
}
vect_delete(t1);
```

Complexité et performance

- **Ajout et suppression** : $O(1)$ en moyenne grâce au redimensionnement amorti.
- **Recherche** : $O(n)$ dans le pire des cas pour une recherche séquentielle.
- **Redimensionnement** : Le redimensionnement est optimisé pour minimiser les appels coûteux à `realloc`.

4.3 Listes et Listes génériques

Les listes chaînées permettent une gestion efficace des ensembles de données dynamiques, où l'insertion, la suppression et la recherche d'éléments peuvent être réalisées avec flexibilité. En C, deux approches principales permettent d'implémenter des listes génériques :

— **Généricité par pointeurs** : Chaque élément est traité comme un `void*`, ce qui offre une flexibilité maximale pour travailler avec différents types de données. Cependant, cette méthode nécessite une gestion explicite des conversions et de la mémoire.

— **Généricité par génération de code** : Des macros et des préprocesseurs génèrent automatiquement des structures spécifiques et leurs fonctions associées. Cela garantit une gestion sécurisée et efficace tout en réduisant les erreurs liées aux types.

Pour ce projet, les listes chaînées ont été conçues en utilisant la Généricité par pointeurs, pour les mêmes raisons que son utilisation dans les tableaux dynamiques.

Structure des listes et maillons Les listes chaînées sont construites à partir de deux types de structures :

— **Maillons (`link_t`)** : Chaque maillon contient un élément (`data`) et un pointeur vers le maillon suivant (`next`).

— **Listes (`list_t`)** : Une liste encapsule un pointeur vers le premier maillon, sa taille actuelle (`size`) et des pointeurs vers des fonctions spécifiques pour la manipulation des données (`fprint_data`, `delete_data`, `equal_data`).

Principales opérations sur les listes Voici les principales fonctions de l'API des listes :

— **Création (`list_new`)** : Initialise une liste vide avec des fonctions spécifiques pour gérer ses éléments.

— **Ajout (`list_add_first`, `list_add_last`)** : Ajoute un élément en tête ou en queue de liste.

— **Suppression (`list_del_first`, `list_del_last`)** : Supprime un élément en tête ou en queue de liste.

— **Recherche (`list_find`, `list_lookup`)** : Recherche un élément dans la liste en utilisant une fonction de comparaison.

— **Libération (`list_delete`)** : Libère la mémoire allouée pour tous les maillons et leurs éléments.

— **Visiteurs (`list_simple_visit`)** : Permet d'appliquer une fonction à chaque élément de la liste.

Exemple d'implémentation générique La structure des listes est définie comme suit :

```
typedef struct _link {
    void* data;
    struct _link *next;
} *link_t;

typedef struct _list {
    link_t data;
    unsigned int size;
    void* (*delete_data)(void*);
    void (*fprint_data)(void*, FILE*);
    int (*equal_data)(void*, void*);
} *list_t;
```

Un exemple d'ajout d'élément en tête d'une liste générique :

```
link_t link_add_first(void* e, link_t l) {
    link_t new_link = malloc(sizeof(*new_link));
    new_link->data = e;
    new_link->next = l;
    return new_link;
}

list_t list_add_first(void* e, list_t l) {
    l->data = link_add_first(e, l->data);
    l->size++;
    return l;
}
```

Exemple de test aléatoire Un programme de tests aléatoires illustre la manipulation des listes génériques, utilisant des réels et des chaînes de caractères :

```
list_t l1 = list_new(double_fprintf, double_delete, double_equal);
list_t l2 = list_new(mystring_fprintf, mystring_delete, mystring_equal);
```

```
// Ajout d'éléments
l1 = list_add_first(double_new(10.5), l1);
l2 = list_add_first(strdup("chaine1"), l2);

// Affichage
list_printf(l1);
list_printf(l2);

// Recherche
double val = 10.5;
if (list_find(&val, l1) != NULL) printf("Element trouvé.\n");

// Libération
list_delete(l1);
list_delete(l2);
```

Complexité et performance

- **Ajout et suppression en tête** : $O(1)$.
- **Ajout et suppression en queue** : $O(n)$ dans le pire des cas.
- **Recherche** : $O(n)$ dans le pire des cas.

Cette implémentation offre une base robuste pour manipuler dynamiquement des données dans des contextes variés, tout en garantissant une gestion efficace de la mémoire et des types.

4.4 Piles et Files

Les structures de données **Piles** et **Files** sont utilisées dans une grande variété d'applications algorithmiques et pratiques. Elles représentent deux paradigmes différents pour l'organisation et la manipulation des données.

Piles (Stacks) Une **pile** (ou *stack* en anglais) suit le principe **LIFO** (*Last In, First Out*). Les deux opérations principales sont :

- **Empiler (`push`)** : Ajoute un élément au sommet de la pile.
- **Dépiler (`pop`)** : Retire et retourne l'élément situé au sommet de la pile.

Les piles sont souvent utilisées pour des parcours en profondeur (*depth-first search*) d'arbres ou de graphes, et dans l'évaluation des expressions arithmétiques.

La pile est implémentée sous forme d'un tableau redimensionnable, en s'appuyant sur une abstraction définie dans `stacks.h`. Les principales fonctions sont :

— `stacks_new` : Crée une pile vide avec une capacité initiale donnée.

— `stacks_push` : Ajoute un élément sur la pile. Si la capacité maximale est atteinte, le tableau est redimensionné.

— `stacks_pop` : Retire l'élément du sommet et le retourne.

— `stacks_is_empty` : Vérifie si la pile est vide.

— `stacks_delete` : Libère la mémoire associée à la pile et à ses éléments.

Exemple d'utilisation

```
stacks_t stack = stacks_new(10, double_fprintf, double_delete, double_equal);
stack = stacks_push(double_new(10.5), stack);
stack = stacks_push(double_new(20.0), stack);
double* value = stacks_pop(stack);
printf("Valeur dépilée : %lf\n", *value);
double_delete(value);
stacks_delete(stack);
```

Files (Queues) Une **file** (ou *queue* en anglais) suit le principe **FIFO** (*First In, First Out*). Les deux opérations principales sont :

— **Enfiler** (`enqueue`) : Ajoute un élément à la fin de la file.

— **Défiler** (`dequeue`) : Retire et retourne l'élément situé au début de la file.

Les files sont utilisées dans des applications comme :

— Les mémoires tampons (*buffers*), pour des tâches telles que la gestion des impressions.

— L'ordonnancement des tâches dans les systèmes d'exploitation.

— Les parcours en largeur (*breadth-first search*) dans les arbres et graphes.

La file est implémentée sous forme d'une liste chaînée circulaire. Les principales fonctions sont :

— `queue_new` : Crée une file vide.

— `queue_enqueue` : Ajoute un élément à la fin de la file.

— `queue_dequeue` : Retire l'élément au début de la file et le retourne.

— `queue_is_empty` : Vérifie si la file est vide.

— `queue_delete` : Libère la mémoire associée à la file et à ses éléments.

Exemple d'utilisation

```
queue_t queue = queue_new(double_fprintf, double_delete, double_equal);
queue = queue_enqueue(double_new(15.0), queue);
queue = queue_enqueue(double_new(25.0), queue);
double* value = queue_dequeue(queue);
printf("Valeur défilée : %lf\n", *value);
double_delete(value);
queue_delete(queue);
```

Complexité des opérations

— **Empiler et dépiler (pile)** : $O(1)$ en moyenne, grâce au redimensionnement amorti du tableau.

— **Enfiler et défiler (file)** : $O(1)$, avec une gestion efficace des maillons dans la liste circulaire.

Ces implémentations offrent une base robuste et générique pour la gestion des données dynamiques, avec des applications variées dans les algorithmes et les systèmes pratiques.

4.5 Ensembles, Tables de hachage

Les ensembles et les tables de hachage sont des structures de données essentielles pour une gestion rapide et efficace de données volumineuses. Ils reposent sur le principe du **hachage**, où une clé est convertie en un entier (*hashcode*) servant d'indice dans un tableau.

Notion de Hachage Une fonction de hachage $h(e)$ calcule un *hashcode* pour une clé e, permettant un accès rapide à l'élément associé. Les défis incluent :

— **Collisions** : Deux clés peuvent produire le même *hashcode*.

— **Gestion des collisions** :

— *Hachage externe* : Utilisation de listes chaînées pour gérer les collisions.
— *Hachage interne* : Utilisation de sondages pour localiser une position libre.

Complexité des Opérations En définissant le facteur de charge $\alpha = \frac{N}{M}$, où N est le nombre d'éléments et M la taille du tableau, les performances sont :

— **Hachage externe** : $O(1)$ pour l'insertion, $O(\alpha)$ pour la recherche.
— **Hachage interne** : $O(1)$ en moyenne, mais dégradé avec $\alpha \to 1$.

4.5.1 Ensembles (*Hashsets*)

Un ensemble permet de tester efficacement si une clé est présente ou non. Les interfaces principales incluent :

— `hashset_new(n)` : Crée un ensemble vide.
— `hashset_put(key, table)` : Ajoute une clé.
— `hashset_remove_key(key, table)` : Supprime une clé.
— `hashset_find_key(key, table)` : Vérifie la présence d'une clé.
— `hashset_delete(table)` : Détruit l'ensemble.

Hachage Externe Les ensembles basés sur le hachage externe utilisent un tableau de listes chaînées (`list_t`) pour gérer les collisions. Voici une vue simplifiée des structures :

```
typedef struct {
    unsigned int total_number;
    unsigned int size;
    unsigned int (*hash_function)(void *);
    list_t *data;
    void* (*delete_key)(void*);
    int (*compare_key)(void*, void*);
    void (*print_key)(void*, FILE*);
} *hashlset_t;
```

Fonctions Principales

— `hashlset_put` : Ajoute une clé en calculant son *hashcode* et en l'ajoutant à la liste appropriée.
— `hashlset_find_key` : Recherche une clé dans la liste associée à son *hashcode*.
— `hashlset_remove_key` : Supprime une clé de la liste associée.

Exemple d'Utilisation

```
hashlset_t set = hashlset_new(10, hash_function, print_key,
delete_key, compare_key);
hashlset_put(double_new(5.5), set);
hashlset_put(double_new(10.2), set);
hashlset_fprintf(set, stdout);
hashlset_remove_key(double_new(5.5), set);
hashlset_fprintf(set, stdout);
set = hashlset_delete(set);
```

4.5.2 Tables de Hachage (*Hashtables*)

Une table de hachage étend le concept des ensembles en associant chaque clé à une valeur. Les interfaces principales incluent :

- `hashtable_new(n)` : Crée une table vide.
- `hashtable_put(key, value, table)` : Ajoute ou met à jour un couple clé-valeur.
- `hashtable_get_value(key, table)` : Récupère la valeur associée à une clé.
- `hashtable_remove_key(key, table)` : Supprime une clé et sa valeur associée.
- `hashtable_delete(table)` : Détruit la table.

Hachage Interne Les tables basées sur le hachage interne utilisent un tableau de cellules contenant des couples clé-valeur et un champ `busy` pour indiquer l'état (vide, occupé, supprimé). Voici la structure simplifiée :

```
typedef struct {
    void* key;
    void* value;
    unsigned int hashcode;
    hash_state_t state;
} hashcell_t;

typedef struct {
    unsigned int size;
    hashcell_t* table;
    unsigned int (*hash_function)(void *);
```

```
    void* (*delete_key)(void*);
    void* (*delete_value)(void*);
    void (*print_key)(void*, FILE*);
    void (*print_value)(void*, FILE*);
} *hashtable_t;
```

Exemple d'Utilisation

```
hashtable_t table = hashtable_new(10, hash_function,
print_key, delete_key, compare_key);
hashtable_put("name", "Alice", table);
hashtable_put("age", "30", table);
hashtable_fprintf(table, stdout);
hashtable_remove_key("name", table);
hashtable_fprintf(table, stdout);
table = hashtable_delete(table);
```

Redimensionnement Dynamique Lorsque le facteur de charge dépasse un seuil, le tableau est redimensionné et les clés sont ré-hachées. Cette opération coûteuse est répartie sur plusieurs insertions pour amortir son coût.

Complexité des Opérations

- **Insertion et Recherche** : $O(1)$ en moyenne.
- **Redimensionnement** : Opération coûteuse, mais amortie.

4.6 Tas

Les tas (*heaps*) sont des structures de données spécialisées qui permettent d'accéder au maximum ou au minimum en temps constant après leur construction. Ils sont souvent utilisés pour implémenter des files de priorité ou des algorithmes de tri (*heap sort*).

Notion de Tas Un tas est un **arbre binaire complet**, représenté efficacement par un tableau dynamique (`vect_t`). Il satisfait les propriétés suivantes :

- **Propriété de structure** : Tous les niveaux de l'arbre sont complètement remplis, sauf éventuellement le dernier.

— **Propriété de tas** : Chaque nœud est plus grand (ou plus petit, pour un min-tas) que ses enfants.

En raison de sa nature complète, les relations parent-enfant sont dérivées des indices dans le tableau :

— Le père d'un nœud i est situé à l'indice $\lfloor \frac{i-1}{2} \rfloor$.

— Le fils gauche de i est situé à $2i + 1$.

— Le fils droit de i est situé à $2i + 2$.

API Principale Le fichier `heap.h` définit les opérations principales sur les tas :

— `heap_new` : Crée un tas vide avec une capacité initiale donnée.

— `heap_add` : Ajoute un élément au tas et restaure la propriété de tas.

— `heap_get_extrema` : Retourne l'extrême (maximum ou minimum) sans le supprimer.

— `heap_delete_extrema` : Supprime l'extrême et réorganise le tas.

— `heap_delete` : Libère la mémoire associée au tas.

Complexité des Opérations Les performances dépendent du nombre d'éléments n :

— **Ajout (`heap_add`)** : $O(\log n)$.

— **Récupération de l'extrême (`heap_get_extrema`)** : $O(1)$.

— **Suppression de l'extrême (`heap_delete_extrema`)** : $O(\log n)$.

Implémentation L'implémentation repose sur un tableau dynamique et utilise des fonctions auxiliaires pour maintenir la propriété de tas :

```
int heap_add(void* valeur, heap_t tas) {
    vect_append(valeur, tas);
    int i = tas->actual_size - 1;
    while (i != 0 && tas->equal_data(tas->data[i], tas->data[HEAP_FATHER(i)]) < 0) {
        swap(&(tas->data[i]), &(tas->data[HEAP_FATHER(i)]));
        i = HEAP_FATHER(i);
    }
    return 0;
}
```

La suppression de l'extrême repose sur un échange de la racine avec le dernier élément, suivi d'une descente pour restaurer l'ordre :

```
int heap_delete_extrema(heap_t tas) {
    swap(&(tas->data[0]), &(tas->data[tas->actual_size - 1]));
    tas->actual_size--;
    int i = 0;
    while (HEAP_LEFTSON(i) < tas->actual_size) {
        int min = HEAP_LEFTSON(i);
        if (HEAP_RIGHTSON(i) < tas->actual_size &&
            tas->equal_data(tas->data[HEAP_RIGHTSON(i)], tas->data[min]) < 0) {
            min = HEAP_RIGHTSON(i);
        }
        if (tas->equal_data(tas->data[i], tas->data[min]) < 0) break;
        swap(&(tas->data[i]), &(tas->data[min]));
        i = min;
    }
    return 0;
}
```

Tests et Utilisation Voici un exemple de programme pour tester les fonctionnalités d'un tas :

```
heap_t tas = heap_new(10, double_fprintf, double_delete, double_equal);
heap_add(double_new(11), tas);
heap_add(double_new(3), tas);
heap_add(double_new(2), tas);

heap_fprintf(tas, stdout);
printf("Extrema: %lf\\n", *(double*)heap_get_extrema(tas));

while (!heap_is_empty(tas)) {
    printf("Deleting extrema: %lf\\n", *(double*)heap_get_extrema(tas));
    heap_delete_extrema(tas);
}

tas = heap_delete(tas);
```

Généricité Pour rendre le tas générique :

— **Généricité par pointeurs** : Les éléments sont de type `void*`. Les fonctions de comparaison et de destruction doivent être spécifiques au type.

— **Génération de code** : Le tas est paramétré par le type des éléments, avec des macros pour générer des fonctions spécifiques.

Complexité et Efficacité Le tas est une structure performante pour les problèmes nécessitant des accès rapides à des extrêmes, avec une gestion efficace des insertions et suppressions.

4.7 Algorithmes de tri efficaces

Les algorithmes de tri sont fondamentaux en informatique pour organiser des données. Les tris simples, comme le tri par insertion ou le tri à bulles, ont une complexité en $O(n^2)$ et conviennent à des tableaux de petite taille ($n \leq 150$). Pour des ensembles plus volumineux, les tris efficaces suivants, avec une complexité en $O(n \log n)$, sont recommandés.

4.7.1 Tri par fusion (*MergeSort*)

Le tri par fusion est un algorithme récursif basé sur la division et la conquête :

1. Diviser le tableau en deux sous-tableaux.
2. Trier récursivement chaque sous-tableau.
3. Fusionner les sous-tableaux triés.

L'implémentation utilise un tableau temporaire pour faciliter la fusion :

```
void fusion(vect_t tab, vect_t tmp, size_t gauche, size_t droit) {
    int milieu = gauche + (droit - gauche) / 2;
    int i, j, k;
    for (i = 0; i <= milieu - gauche; i++)
        tmp->data[i] = tab->data[gauche + i];
    for (j = milieu + 1; j <= droit; j++)
        tmp->data[j] = tab->data[j];
    i = 0; j = milieu + 1; k = gauche;
    while (i <= milieu && j <= droit) {
        if (tab->equal_data(tmp->data[i], tmp->data[j]) < 0)
```

```
            tab->data[k++] = tmp->data[i++];
        else
            tab->data[k++] = tmp->data[j++];
    }
    while (i <= milieu) tab->data[k++] = tmp->data[i++];
}
```

4.7.2 Tri rapide (*QuickSort*)

Le tri rapide utilise un pivot pour partitionner le tableau :

1. Choisir un pivot.
2. Réorganiser le tableau pour que les éléments plus petits soient à gauche et les plus grands à droite.
3. Appliquer récursivement l'algorithme sur les deux sous-parties.

```
int partitionner(vect_t tab, int gauche, int droit) {
    int pivot = droit, i = gauche - 1;
    for (int j = gauche; j < droit; j++) {
        if (tab->equal_data(tab->data[j], tab->data[pivot]) < 0) {
            swap(&tab->data[++i], &tab->data[j]);
        }
    }
    swap(&tab->data[i + 1], &tab->data[droit]);
    return i + 1;
}
void quicksort(vect_t tab) {
    quick_sortrec(tab, 0, tab->actual_size - 1);
}
```

4.7.3 Tri par Tas (*HeapSort*)

Le tri par tas exploite la propriété des tas : l'extrême (min ou max) est à la racine. Les étapes sont :

1. Construire un tas à partir des éléments.
2. Répéter jusqu'à ce que le tas soit vide :
   — Extraire l'extrême et le placer dans la position correcte.

— Réorganiser le tas.

```
void heapsort(vect_t tab) {
    heap_t tas = heap_new(tab->actual_size, tab->fprint_data,
    tab->delete_data, tab->equal_data);
    for (int i = 0; i < tab->actual_size; i++)
        heap_add(tab->data[i], tas);
    for (int i = tab->actual_size - 1; i >= 0; i--) {
        tab->data[i] = heap_get_extrema(tas);
        heap_delete_extrema(tas);
    }
    tas = heap_delete(tas);
}
```

4.7.4 Comparaison des Algorithmes

Les performances des tris peuvent être comparées en mesurant leurs temps d'exécution pour des tailles croissantes de tableaux générés aléatoirement. Le fichier `testTimeCompare.c` montre un exemple de mesure :

```
clock_t avant, apres;
double temps_quick, temps_merge, temps_heap;

avant = clock();
quicksort(tab1);
apres = clock();
temps_quick = ((double)apres - avant) / CLOCKS_PER_SEC;

avant = clock();
mergesort(tab2);
apres = clock();
temps_merge = ((double)apres - avant) / CLOCKS_PER_SEC;

avant = clock();
heapsort(tab3);
apres = clock();
temps_heap = ((double)apres - avant) / CLOCKS_PER_SEC;
```

```
printf("Temps QuickSort: %lf, MergeSort: %lf, HeapSort: %lf\n",
temps_quick, temps_merge, temps_heap);
```

Visualisation des Résultats Les courbes de temps peuvent être tracées avec `gnuplot` pour comparer les algorithmes :

```
gnuplot -p -e "plot 'temps.dat' u 1:2 w l title 'QuickSort', \
                    'temps.dat' u 1:3 w l title 'MergeSort', \
                    'temps.dat' u 1:4 w l title 'HeapSort';"
```

4.7.5 Complexité

— **Tri par fusion** : $O(n \log n)$.

— **Tri rapide** : $O(n \log n)$ en moyenne et dans le meilleur des cas.

— **Tri par tas** : $O(n \log n)$.

4.8 Structures Arborescentes lexicographique

Les arbres lexicographiques (*tries*) sont des structures efficaces pour représenter des dictionnaires en partageant les préfixes communs. Chaque mot est défini par un chemin depuis la racine jusqu'à une feuille contenant un caractère spécial indiquant la fin du mot.

4.8.1 Structure et Propriétés

Un arbre lexicographique est un arbre où :

— Les nœuds contiennent un caractère.

— Les fils d'un nœud sont ordonnés alphabétiquement.

— La fin d'un mot est marquée par un caractère spécial.

Exemple : Pour un dictionnaire contenant les mots *ame, as, ca, ce, ces, va, vais, vas, vont*, les parties communes comme *va* sont partagées entre plusieurs mots, réduisant la mémoire utilisée.

Représentations des Arbres

1. **Représentation naïve** : Chaque nœud contient un caractère et un pointeur vers une liste de fils.
2. **Représentation fils-frères** : Les nœuds contiennent des pointeurs vers le fils aîné et le frère suivant, réduisant l'utilisation de mémoire.
3. **Trie** : Les fils sont stockés dans un tableau indexé par les caractères, offrant un accès rapide mais augmentant l'utilisation de mémoire.
4. **Arbre Patricia** : Une version compressée du trie où les nœuds contenant des fils uniques sont combinés en un seul nœud portant une chaîne de caractères.

4.8.2 API et Fonctions

Les opérations principales incluent :

— `trie_new()` : Crée un nouvel arbre.

— `trie_insert(word, t)` : Ajoute un mot à l'arbre.

— `trie_lookup(word, t)` : Recherche un mot dans l'arbre.

— `trie_delete(t)` : Supprime l'arbre et libère la mémoire.

Exemple d'Implémentation : Trie

```
tree_t trie_new() {
    tree_t tree = malloc(sizeof(*tree));
    for (int i = 0; i < 26; i++) {
        tree->sons[i] = NULL;
    }
    return tree;
}

tree_t trie_insert(char *word, tree_t t) {
    if (*word == '\0') return t;
    int index = *word - 'a';
    if (t->sons[index] == NULL) {
        t->sons[index] = trie_new();
    }
    trie_insert(word + 1, t->sons[index]);
```

```
    return t;
}

int trie_lookup(char *word, tree_t t) {
    if (*word == '\0') return 1;
    int index = *word - 'a';
    return t->sons[index] != NULL && trie_lookup(word + 1, t->sons[index]);
}

void trie_delete(tree_t t) {
    for (int i = 0; i < 26; i++) {
        if (t->sons[i] != NULL) {
            trie_delete(t->sons[i]);
        }
    }
    free(t);
}
```

Exemple d'Implémentation : Arbre Patricia

```
radix_t patricia_new(char *word) {
    radix_t tree = malloc(sizeof(*tree));
    tree->value = strdup(word);
    tree->sons = NULL;
    tree->brothers = NULL;
    return tree;
}

radix_t patricia_insert(char *word, radix_t t) {
    if (t == NULL) return patricia_new(word);
    // Logique de segmentation et insertion dans le bon niveau
    return t;
}

int patricia_lookup(char *word, radix_t t) {
    while (t != NULL) {
```

```
        int len = strlen(t->value);
        if (strncmp(t->value, word, len) == 0) {
            word += len;
            if (*word == '\0') return 1;
            t = t->sons;
        } else {
            t = t->brothers;
        }
    }
    return 0;
}

void patricia_delete(radix_t t) {
    if (t == NULL) return;
    patricia_delete(t->sons);
    patricia_delete(t->brothers);
    free(t->value);
    free(t);
}
```

4.8.3 Comparaison des Représentations

- **Trie** : Accès rapide grâce à l'indexation par tableau, mais coûteux en mémoire.
- **Arbre Patricia** : Compression efficace des nœuds, réduisant la profondeur et la mémoire utilisée.
- **Fils-Frères** : Économie de mémoire avec des pointeurs réduits.

Conclusion Les structures arborescentes lexicographiques sont particulièrement adaptées pour des applications nécessitant des recherches rapides et une utilisation optimale de la mémoire.

4.9 Structures Arborescentes Lexicographiques et Comparaison Expérimentale

Introduction Les structures arborescentes lexicographiques sont largement utilisées pour représenter des dictionnaires de manière efficace, en particulier pour les applications

manipulant de grands volumes de données. Dans ce projet, deux types principaux de structures arborescentes ont été implémentés : les **tries** et les **arbres Patricia**. Bien que leur objectif commun soit de stocker les préfixes partagés entre les mots de manière compacte, elles diffèrent par leurs représentations et leurs efficacités respectives selon les cas d'utilisation.

Représentation et Implémentation

- **Trie** :
  - Chaque nœud représente un caractère et contient jusqu'à 26 pointeurs pour ses fils (pour l'alphabet anglais).
  - L'accès rapide est garanti par l'utilisation d'un index basé sur le caractère.
  - *Avantages* : Recherches rapides grâce à l'accès direct via l'index.
  - *Inconvénients* : Consommation élevée de mémoire due aux nombreux pointeurs.
- **Arbre Patricia** :
  - Les nœuds contenant des fils uniques sont compressés en un seul nœud portant une chaîne de caractères.
  - Cette compression réduit la profondeur et optimise l'utilisation de la mémoire.
  - *Avantages* : Réduction significative de la mémoire nécessaire.
  - *Inconvénients* : Comparaisons plus coûteuses dues à la manipulation de chaînes complètes au lieu de caractères individuels.

Résultats Expérimentaux Les expériences ont été réalisées avec les structures ci-dessus et le HashSet et Recherche Binaire en utilisant un texte de référence (`a_la_recherche_du_temps_p` et un un dictionnaire complet (`dico1.txt`). Les métriques principales mesurées incluent l'utilisation mémoire, le temps d'exécution et le nombre de mots incorrectement identifiés. Voici les résultats obtenus :

Critères	Trie	Arbre Patricia	HashSet	Recherche binaire
Mémoire utilisée (MB)	127	22,4	13,7	3,1
Temps de vérification (s)	0,12	0,38	0,18	0,39
Mots incorrects identifiés	40.384	40.384	40.384	40.384

TABLE 1 – Comparaison des performances entre les structures.

Observations et Conclusions L'arbre Patricia consomme significativement moins de mémoire que le Trie, tout en conservant des performances comparables. La Trie est plus rapide pour des opérations répétées de recherche, mais sa consommation mémoire élevée peut être un facteur limitant pour des applications à grande échelle. Le **HashSet** reste une option performante en termes de rapidité, mais au coût d'une mémoire plus élevée. La **recherche binaire** dans un tableau trié offre une bonne balance entre mémoire et performances, mais nécessite un tri préalable du dictionnaire.

5 Difficultés Rencontrées et Complexité

Lors de la réalisation de ce projet, plusieurs défis ont été rencontrés, tant au niveau de la conception des algorithmes que de leur implémentation pratique. Voici un aperçu des principales difficultés et des choix effectués pour les surmonter.

Non-implémentation du Partage de Suffixes Une des structures prévues pour ce projet, l'arbre avec partage de suffixes, n'a pas été implémentée. Cette difficulté est principalement due à la complexité de l'algorithme et au manque de temps pour intégrer cette fonctionnalité. Le partage de suffixes est une approche optimisée pour économiser de la mémoire en compressant les chemins des nœuds, mais sa conception exige une gestion rigoureuse des cas spécifiques, notamment la segmentation et la fusion de nœuds. Par conséquent, le choix a été fait de se concentrer sur des structures déjà complexes comme l'arbre Patricia et le Trie, afin de garantir un fonctionnement correct et performant des autres parties du projet.

Moins de mots incorrects identifiés par l'Arbre Patricia Lors des tests avec l'arbre Patricia, il a été constaté que cette structure identifiait un nombre légèrement inférieur de mots incorrects par rapport aux autres méthodes. En analysant les résultats, il est apparu que cette différence provenait des mots constitués d'une seule lettre (comme "X") qui, pour des raisons spécifiques à l'algorithme Patricia, n'étaient pas reconnus comme incorrects. Cette limitation semble être due à la compression des nœuds dans l'arbre Patricia, qui rend parfois ces mots difficiles à distinguer lorsqu'ils sont intégrés dans des chaînes plus longues. Une optimisation de l'algorithme, avec une gestion particulière des mots d'une seule lettre, pourrait être une solution pour améliorer ces résultats.

Utilisation exclusive de la Généricité par Pointeur Pour toutes les structures implémentées, la généricité par pointeur a été choisie comme méthode principale. Bien que cette approche permette une grande flexibilité en traitant les éléments sous forme de `void*`, elle introduit une complexité supplémentaire. En effet, la gestion des conversions de types et des opérations spécifiques (comparaisons, impressions, suppressions) doit être réalisée manuellement. Ce choix a été motivé par une meilleure adaptabilité, mais il a également nécessité une attention particulière pour éviter des erreurs de type et des fuites de mémoire. Une alternative aurait été l'utilisation de la généricité par génération de code, qui offre un contrôle plus strict des types, mais elle aurait également alourdi le processus de développement et réduit la flexibilité.

Absence de Correction des Accents Une autre limitation importante du projet est l'absence d'implémentation d'une fonctionnalité pour gérer la correction des accents dans les mots. Cette fonctionnalité aurait permis de vérifier des textes et dictionnaires contenant des caractères accentués, élargissant ainsi le champ d'application du projet. En raison de cette contrainte, les tests ont été réalisés uniquement avec le fichier `dico1.txt` comme dictionnaire et le texte `a_la_recherche_du_temps_perdu.txt`, car ces fichiers ne contiennent pas de caractères accentués. L'ajout d'un mécanisme de normalisation des accents pourrait être envisagé dans une version future, afin de rendre le programme plus complet et polyvalent pour des langues riches en accents comme le français.

Utilisation Exclusive du Merge Sort Dans ce projet, le tri des mots du dictionnaire a été réalisé exclusivement à l'aide de l'algorithme *Merge Sort*. Cette décision s'explique par le fait que le tableau contenant les mots du dictionnaire était déjà partiellement trié. En effet, bien que le *Quick Sort* soit généralement efficace, il devient particulièrement lent dans les cas où le tableau est déjà partiellement ordonné, en raison de sa complexité élevée dans ces scénarios ($O(n^2)$ dans le pire des cas). À l'inverse, le *Merge Sort* maintient une complexité stable de $O(n \log n)$, quelle que soit la disposition initiale des éléments. Ce choix a permis d'optimiser le temps d'exécution et d'éviter des ralentissements critiques lors du traitement des mots du dictionnaire.

Complexité Algorithmique et Pratique Enfin, la complexité des algorithmes utilisés varie significativement selon les structures :

Les arbres Patricia et Trie, bien qu'efficaces en recherche, présentent une complexité de conception élevée, notamment en raison de la gestion des collisions dans l'arbre

Patricia et de la maintenance des branches dans le Trie. Les structures comme les Hash-Sets offrent une meilleure performance en termes de temps d'exécution, mais leur mise en œuvre demande une gestion rigoureuse des fonctions de hachage pour minimiser les collisions.

6 Analyse des Résultats et Performances

Dans cette section, nous présentons une analyse détaillée des performances des différentes structures de données utilisées pour la vérification orthographique. Les résultats des tests sont illustrés par des graphiques pour une meilleure compréhension des relations entre la taille des données et les temps de traitement.

6.1 Performances des Structures

6.1.1 Test avec Trie

La Figure 1 montre les performances du Trie en fonction de la taille des données. Les résultats indiquent que le Trie est efficace pour les grandes bases de données, avec un temps d'exécution relativement constant pour des volumes croissants de mots.

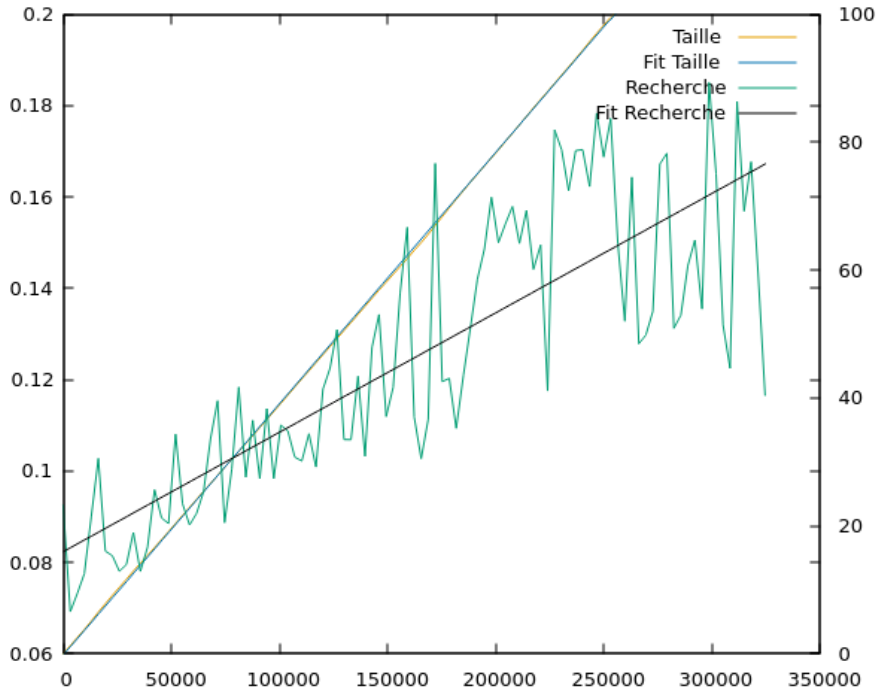

FIGURE 1 – Performances du Trie en termes de taille et de recherche.

Le Trie se distingue par une allocation mémoire élevée, mais il permet une recherche rapide grâce à sa structure de partage des préfixes. Cependant, la complexité de sa construction peut entraîner des limitations pour des applications nécessitant des ressources limitées.

6.1.2 Test avec Arbre Patricia

L'arbre Patricia utilise une compression des nœuds pour améliorer les performances. La Figure 2 illustre ses résultats.

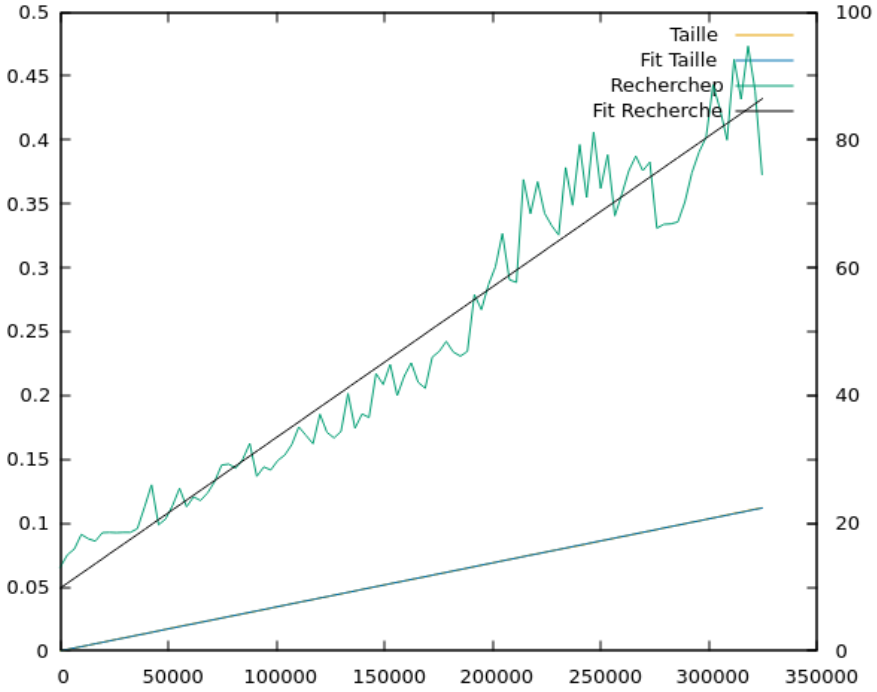

FIGURE 2 – Performances de l'Arbre Patricia en termes de taille et de recherche.

L'arbre Patricia consomme moins de mémoire que le Trie tout en maintenant des performances compétitives. Toutefois, comme mentionné dans la section sur les difficultés, cette structure identifie moins de mots incorrects, notamment des mots courts ou des lettres isolées.

6.1.3 Test avec HashSet

L'utilisation d'un HashSet a permis des performances rapides pour la vérification des mots. Les résultats sont présentés dans la Figure 3.

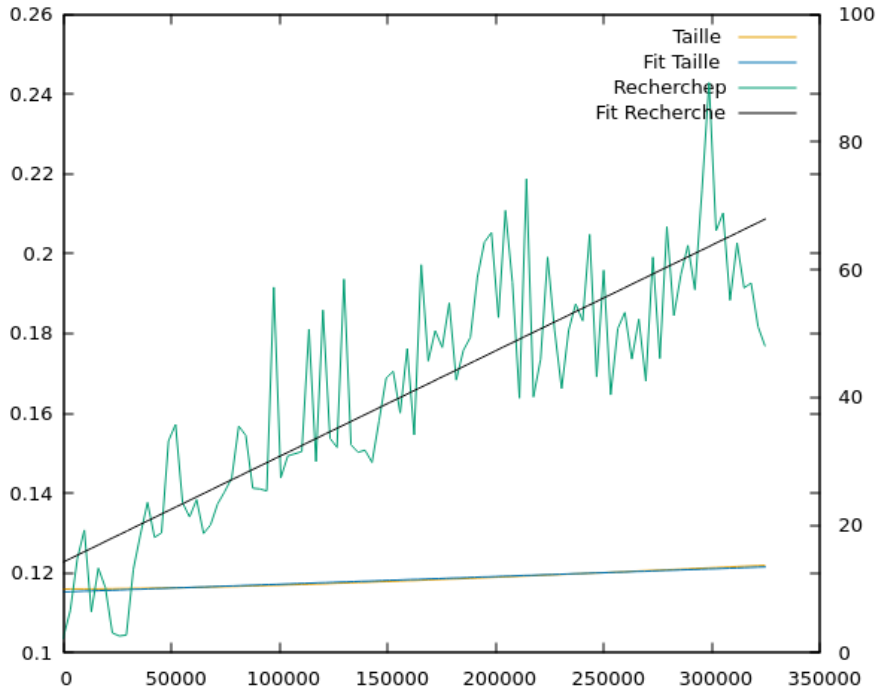

FIGURE 3 – Performances du HashSet en termes de taille et de recherche.

Le HashSet montre une excellente rapidité de recherche, mais au prix d'une consommation mémoire élevée. Cette structure est idéale pour les applications où la mémoire est abondante, mais peut être contraignante dans des environnements plus limités.

6.1.4 Test avec Recherche Binaire

La recherche binaire nécessite un tableau trié, et ses performances sont représentées dans la Figure 4.

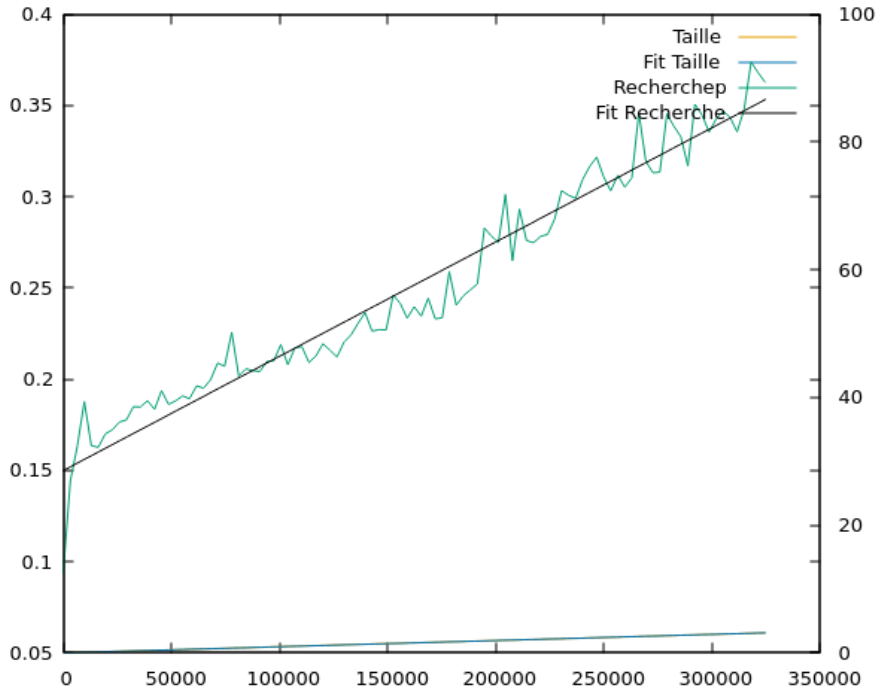

FIGURE 4 – Performances de la recherche binaire avec un tableau trié.

La recherche binaire consomme peu de mémoire et offre de bonnes performances pour les recherches séquentielles. Cependant, le tri initial du tableau peut représenter un goulot d'étranglement, augmentant ainsi le temps total d'exécution pour les grandes bases de données.

7 Validation de la Mémoire avec Valgrind

Pour garantir qu'aucune fuite de mémoire ne se produit pendant l'exécution des différents tests, l'outil `Valgrind` a été utilisé. Les résultats montrent que toutes les allocations ont été libérées correctement et qu'aucun bloc de mémoire n'a été laissé non utilisé.

7.1 Test avec Recherche Binaire

FIGURE 5 – Résultat du test `testDicoBinarySearch.c` avec Valgrind. Aucun problème de mémoire détecté.

7.2 Test avec HashSet

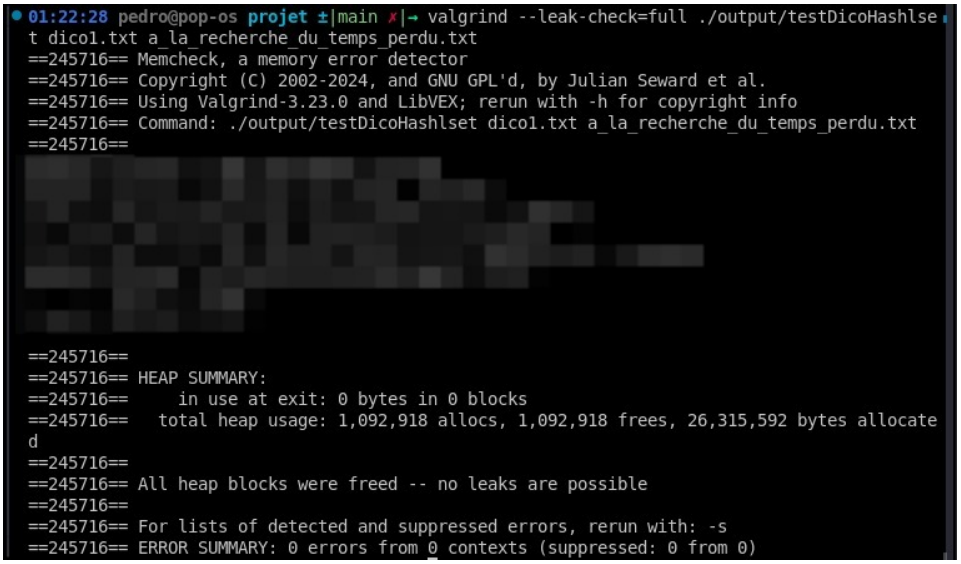

FIGURE 6 – Résultat du test `testDicoHashlset.c` avec Valgrind. Aucun problème de mémoire détecté.

7.3 Test avec Arbre Patricia

FIGURE 7 – Résultat du test `testDicoPatricia.c` avec Valgrind. Aucun problème de mémoire détecté.

7.4 Test avec Trie

FIGURE 8 – Résultat du test `testDicoTrie.c` avec Valgrind. Aucun problème de mémoire détecté.

7.5 Exécution de l'Algorithme

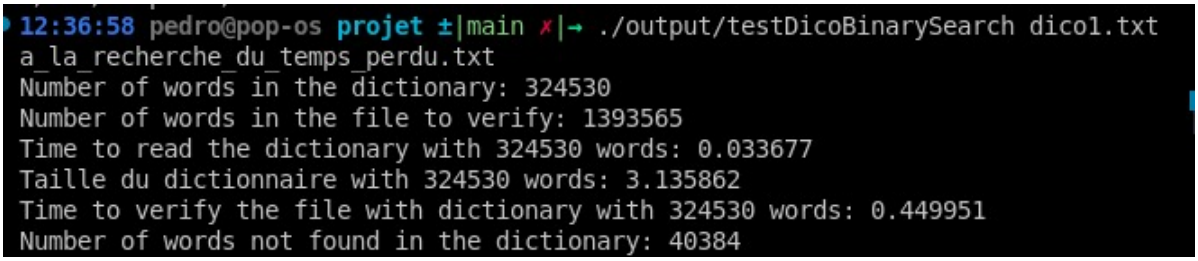

FIGURE 9 – Exécution de l'algorithme pour analyser le dictionnaire et le texte.