Initiation à la Programmation C - Projet

Nombre d’occurrences des mots d’un texte
Comparaison d’algorithmes

Formation	L2-MI - Semestre 3 - 2025
Binôme	Dani Hugo
	6 janvier 2026

Objectif : concevoir un programme en C qui lit un fichier texte, compte les occurrences des mots (définis comme une suite de lettres, éventuellement séparées par des tirets), puis affiche les mots les plus fréquents. Plusieurs algorithmes sont comparés en temps d’exécution et en mémoire.

1. Organisation et répartition du travail

Répartition globale : Dani 60 % / Hugo 40 %.

Hugo : gestion mémoire (InfoMem + myMalloc / myRealloc / myFree) et implémentation de l'algorithme 1 (liste chaînée non triée).
Dani : implémentation des autres algorithmes (arbre, dictionnaire, liste triée), mise en place du mode « top N », génération et exploitation des mesures, script de visualisation et analyse des résultats.

Chaque membre est capable d'expliquer le fonctionnement global du programme et les résultats présentés.

2. Arborescence du rendu

Le rendu est organisé dans **un seul dossier** nommé **projet/** contenant :

```
projet/
|-- projet.c
|-- performances.csv
|-- visualisation.py
`-- Readme.txt
```

Le fichier **Readme.txt** explique comment compiler et lancer le programme, ainsi que comment générer les graphiques à partir de **performances.csv** via le script Python **visualisation.py**.

3. Fonctionnalités du programme

Le programme répond aux exigences minimales du sujet :

• Prend un fichier texte en argument.
• Extrait les mots en filtrant la ponctuation et en acceptant les mots contenant un tiret.
• Compte les occurrences et affiche les **N mots** les plus fréquents (tri décroissant des occurrences).
• Mesure le temps CPU et estime la mémoire maximale via la structure InfoMem.
• Enregistre les mesures dans **performances.csv** (temps + mémoire).
• Amélioration : option **k** (taille minimale) pour ne garder que les mots d'au moins **k** lettres.
• Implémente **4 algorithmes** (liste, arbre, dictionnaire, liste triée).

4. Algorithmes comparés

4.1 Algorithme 1 - Liste chaînée (naïf)

Structure : liste chaînée non triée. Pour chaque mot lu, on parcourt la liste pour le chercher. S'il est trouvé, on incrémente son compteur ; sinon on crée une nouvelle cellule et on l'insère (en tête). Recherche en O(U) où U est le nombre de mots uniques.

4.2 Algorithme 2 - Arbre binaire de recherche (ABR)

Structure : arbre binaire de recherche ordonné alphabétiquement. Insertion et recherche par descentes gauche/droite selon strcmp. Bon comportement en pratique si l'arbre n'est pas trop déséquilibré.

4.3 Algorithme 3 - Dictionnaire (tableau de listes)

Structure : tableau de 26 listes chaînées, indexées par la première lettre du mot. La recherche se limite à un bucket, ce qui réduit le nombre de comparaisons et accélère l'exécution.

4.4 Algorithme 4 - Liste chaînée triée

Structure : liste chaînée maintenue triée. L'insertion implique de trouver la bonne position, ce qui reste coûteux. Les performances se situent entre la liste naïve et les structures plus efficaces (dico/arbre).

5. Gestion de la mémoire (InfoMem)

Pour comparer la consommation mémoire, le programme encapsule malloc, realloc et free dans myMalloc, myRealloc et myFree. À chaque allocation/désallocation, des compteurs cumulés sont mis à jour et on conserve la mémoire maximale observée (cumul_alloc - cumul_desalloc).

Cette mesure fournit une estimation cohérente permettant de comparer les algorithmes sur les mêmes données.

6. Résultats expérimentaux et analyse

Les mesures sont enregistrées dans performances.csv puis visualisées avec un script Python (matplotlib). Les graphiques suivants comparent les algorithmes en fonction du nombre de mots traités.

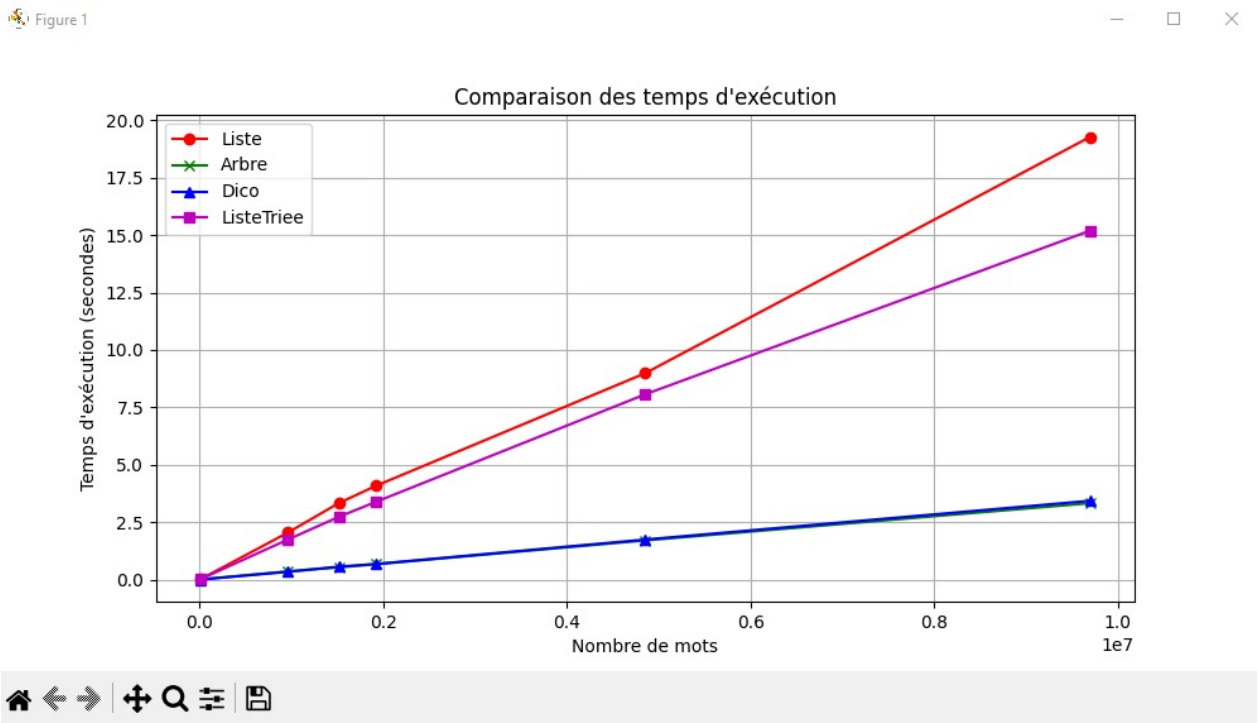

Figure 1 - Comparaison des temps d'exécution.

Analyse (temps) : la liste chaînée non triée est la plus lente et se dégrade fortement quand la taille augmente, car chaque mot nouveau implique une recherche linéaire. La liste triée reste coûteuse (recherche + insertion). Le dictionnaire (26 listes) et l'arbre sont nettement plus rapides car ils réduisent le nombre de comparaisons (buckets pour le dico, hauteur de l'arbre pour l'ABR).

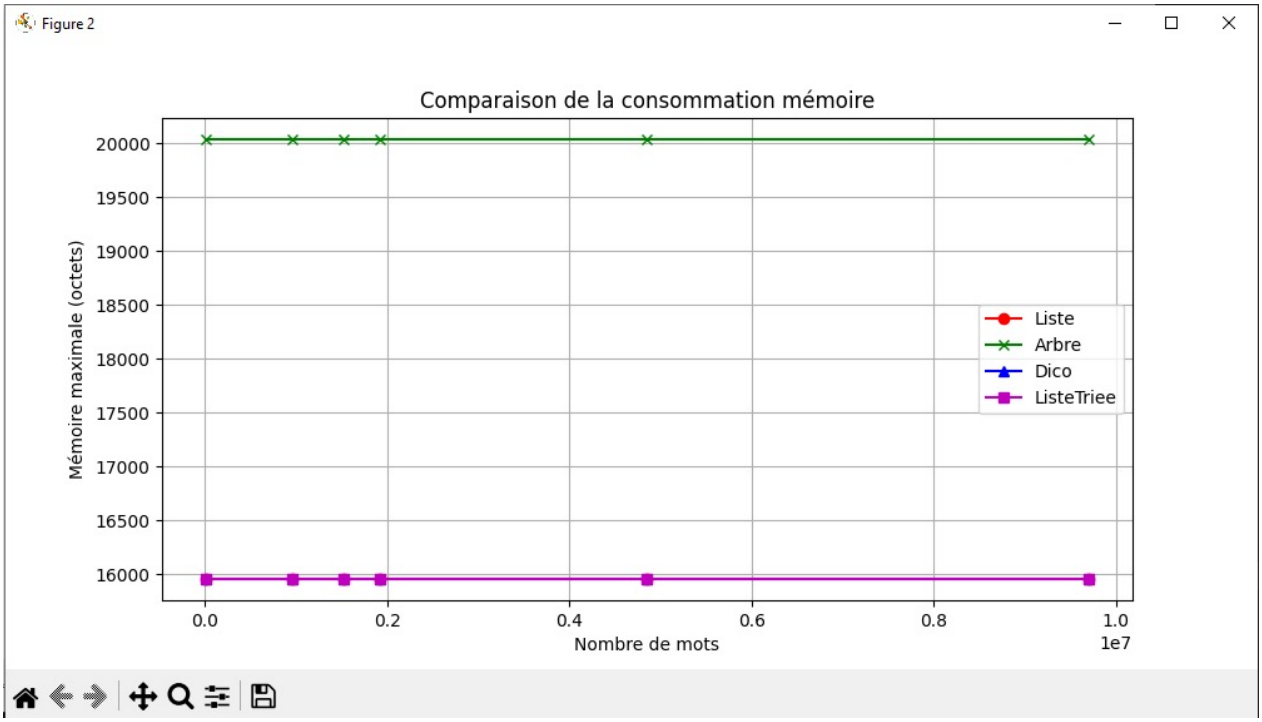

Figure 2 - Comparaison de la consommation mémoire maximale.

Analyse (mémoire) : l'arbre consomme davantage car chaque mot unique nécessite un nœud avec deux pointeurs (gauche/droite) en plus de la chaîne de caractères. Les structures à base de listes utilisent un seul pointeur par cellule (suivant), ce qui limite le surcoût. On observe ainsi un coût mémoire plus élevé pour l'ABR.

7. Conclusion

La comparaison met en évidence l'importance des structures de données. La liste naïve est simple mais inefficace à grande échelle. Le dictionnaire et l'arbre améliorent fortement le temps d'exécution, avec un surcoût mémoire plus marqué pour l'arbre. L'option k (taille minimale) apporte une amélioration pratique en filtrant les mots courts.