Nathann Morand sciper :296190

Rapport Projet SAME GRANMA

Structure générale du projet

1) lire le dictionnaire via une fonction lecture qui se charge de d'ajouter les mots au dictionnaire dans l'ordre (trie par insertion), d'appeler les fonctions responsables de calculer les propriété de chaque mots (nbT,nbD, alpha, place dans l'index) et de traiter les éventuelles exceptions. Ceci jusqu'à ce que "." soit entrée.
2) Le programme affiche le dictionnaire.
3) La partie suivantes du programme est répété autant de fois que nécessaire :
    *) lire l'anagrammes en réutilisant la fonction lecture.
    *) enlever tous les mots d'une copie du dictionnaire qui sont fait de lettres qui ne sont pas dans l'anagramme
    *) calculer l'espace des anagrammes possible engendré par le sous-dictionnaire (2^n combinaisons)
    *) parcourir les combinaisons en vérifiant la valeur alpha de chaque candidat avec celle de l'anagramme.
    *) pour chaque anagramme trouvé afficher les permutation possibles des mots qui le compose (n!)

Pseudo code du trie hiérarchique

La fonction findPos et Insert sont utilisé successivement sur chaque élément ajouté au dictionnaire durant la lecture.

```
findPos

Trouve la position a laquelle un élément doit etre rangé dans un dictionnaire.
Utiliser la fonction compare pour comparer la taille, la diversité de lettre et le alpha.
vérifie manuellement l'ordre alphabétique si tout le reste est identique.

Entrée : Dictionnaire, element
Sortie : index
```

```
continuerLaRecherche <- true
Tant que continuerLaRecherche
  Si taille(dictionnaire) <= index
    retourner index // l'index est a la fin du dictionnaire. ne peut aller plus loin.
  resultatComparaison <- compare(Dictionnaire(i), element)
  Si resultatComparaison = égal
    Si entree.mot > Dictionnaire(i).mot //vérifie l'ordre alphabétique
      i <- i + 1
    Sinon
      continuerLaRecherche <- faux
  Sinon Si resultatComparaison = inférieur
    i <- i + 1
  Sinon Si resultatComparaison = supérieur
    continuerLaRecherche <- faux
Sortir i
```

```
insert

inserer un élément dans le dictionnaire a la position donné.

Entrée : index, element
Entrée modifié : Dictionnaire
Sortie : rien
```

```
buffer <- elementToInsert
Si positionInsertion < taille(dictionnaire)
  buffer <- dictionnaire(positionInsertion)
  dictionnaire(i) <- elementToInsert
  Pour i allant de positionInsertion +1 à la taille du dictionnaire
    bufferLocal <- dictionnaire(i)
    dictionnaire(i) <- buffer
    buffer <- bufferLocal
dictionnaire.push_back(buffer)
```

```
compare

compare deux mots selon leurs nbT nbD et alpha hiéerarchiquement

Entrée : motA, motB
Sortie : inférieur, égal ou supérieur
```

```
Si motA.nbT > motB.nbT
  retourner supérieur
Sinon Si motA.nbT < motB.nbT
  retourner inférieur
Sinon Si motA.nbT = motB.nbT
  Si motA.nbD > motB.nbD
    retourner supérieur
  Sinon Si motA.nbD < motB.nbD
    retourner inférieur
  Sinon Si motA.nbD = motB.nbD
    Si motA.alpha > motB.alpha
      retourner supérieur
    Sinon Si motA.alpha < motB.alpha
      retourner inférieur
    Sinon Si motA.alpha = motB.alpha
      retouner égal
```

Ordre de complexité

L'ensemble du processus de recherche d'anagramme est de composé de :

- trimDict qui est en O(n) avec n le nombre de mots dans le dictionnaire
- ComputeMultiverse qui est en O(n^2) avec n le nombre de mots dans le dictionnaire simplifier via trimDict
- Anagram qui est en O(n) avec n le nombre d'entrée dans anaSpace et O(n!) avec n le nombre de mots qui compose chaque anagramme.

Le coût dominant étant O(n!) du nombre de mots qui compose chaque anagramme
les résultats ne dépendent que peu de nombre de mot dans le dictionnaire (dont le coût dominant est $O(n^2)$). Ceci est cohérent avec le graphique ci-dessous si on tiens compte du nombre de mots qui compose chaque anagramme (nbA).

Graph

Voici les temps obtenu à l'exécution des fichiers de test originaux :

le point Vert	le point Bleu	le point Rouge
t14_perf3.txt: 6.285[s]	t13_perf2.txt: 0.590[s]	nbM=9, nbA=8
nbM=11, nbA=10	nbM=10, nbA=9	nbM=9, nbA=8