TP4 C++

Table des matières :

1. Introduction ……………………………………………………………………………………………………..
2. Spécifications ……………………………………………………………………………………………………..
3. Architecture Globale ……………………………………………………………………………………………
4. Structure des Données ……………………………………………………………………………………………
5. Conclusion ………………………………………………………………………………………………………………
6. Introduction

L’objectif de ce TP est d’appliquer l’utilisation de la Standard Library C++ (STL) et de manipuler des flux de données lors de la conception d’une application. Nous avons essayé de développer des solutions génériques et réutilisables, qui peuvent facilement être adaptées pour des nouveaux problèmes.

Plus spécifiquement, notre application doit lire un fichier journal qui contenait des logs Apache (des lignes décrivant des “parcours” entre des pages Web), les modéliser et les ranger dans une structure de données qui nous permettrait de les manipuler facilement. Étant donné que le fichier peut être très volumineux, le choix d’une structure de données cohérente devient crucial.

Ce projet a été fait en deux parties principales : la conception des classes et la réflexion aux structures de données qui seraient utilisées pour mieux répondre à la problématique d’une part et l’implémentation en langage C++ d’autre part.

ll. Spécifications

Le programme fonctionne en ligne de commande. L’installation est faite en suivant les instructions dans le fichier README.md. Nous pouvons découper le fonctionnement général en trois parties :

Main :

Point d’entrée du programme : réalise la lecture du fichier et le traitement (filtrage ou non) des logs.

Cas normaux :

* + On peut mettre les options dans n'importe quelle ordre (comme une commande linux classique) **(Tests 6,7,8,9,10,11)**
  + Options possibles :
    - -g nomFichier.dot : génère un fichier GraphViz
    - [default] Afficher les 10 documents les plus consultés **(Test 13)**
    - -e : exclut documents de type image, CSS ou JS. **(Test 14)**
    - -t heure : filtre sur les heures [heure, heure+1[ **(Test 15)**
    - -h : affiche le manuel d’utilisation du programme

Cas limites:

* + Si nomFichier.dot est déjà existant : on demande confirmation à l’utilisateur**(Test 4)**
  + Option inconnue : option ignorée
  + Absence d'argument nom de fichier : afficher le manuel d’utilisation**(Test 1)**

Cas d'erreurs :

* + Le paramètre t doit être un entier entre 0 et 23. **(Test 5 et 17)**
  + « Heure » et « nomFichier.dot » sont des arguments obligatoires : leur absence doit générer une erreur. **(Test 2)**

Parsing et traitement des données :

Cas normaux :

* + Pour chaque ligne de log, on ajoute la requête à la structure de données. Si la cible est déjà connue, incrémenter son compteur de hits et ne pas l’ajouter une seconde fois. (de même pour le referer)
  + On doit pouvoir exclure tous les documents images/css/javascript
    - Est définie comme « image », une requête contenant une des extensions suivantes : .jpg, .jpeg, .jfif, .jif, .png, .gif, .webp, .svg, .ico, .tiff, .tif, .jp2, .jpx, .j2k, .fpx, .pcd
    - Est définie comme « css » une requête contenant une des extensions suivantes : .css
    - Est définie comme « javascript » une requête contenant une des extensions suivantes : .js

Cas limites :



Cas d'erreurs :

* + Ligne de log mal formée : indiquer l’échec en mettant le flag « failbit » du flux à 1 **(Test 16)**

Génération du GraphViz :

Cas normaux:

* + Pouvoir passer de la structure de données des logs analysées à un graphe GraphViz bien formé

Cas limites :

* + Pas de requêtes à exporter : générer un graphe vide**(Test 3)**

lll. Architecture Globale de l’Application

Diagramme de Classes


* Request:

Chaque ligne de log est parsée dans un objet de type Request. Il contient tous les attributs correspondants ainsi que les opérateurs surchargés pour la lecture et écriture. Ceux-ci permettent notamment de parser une requête correctement formattée (format type Apache/Nginx) depuis n’importe quel flux d’entrée, et d’écrire dans ce même format sur n’importe quel flux de sortie.

* LogFileParser:

Cette classe fait la lecture du fichier en appliquant les options demandées et les filtres.

* HourFilter et ExtensionFilter:

Filtres implémentés à partir de la classe abstraite AbstractFilter. HourFilter peut être utilisé pour faire un filtre de [heure, heure+1[ (défaut) ou [heureDebut, heureFin[ .ExtensionFilter peut être utilisé pour filtrer n’importe quelle extension. Nous avons choisi de laisser les extensions en paramètre pour pouvoir réutiliser les classes.

* Caches:

Afin de gagner en espace mémoire, chaque string URI/referer est associée à un identifiant entier unique. Ainsi l’application manipule principalement des entiers qui sont moins coûteux en mémoire et ne récupère la valeur de la chaîne de caractères uniquement si nécessaires (affichage ou export).

* RefererEdge

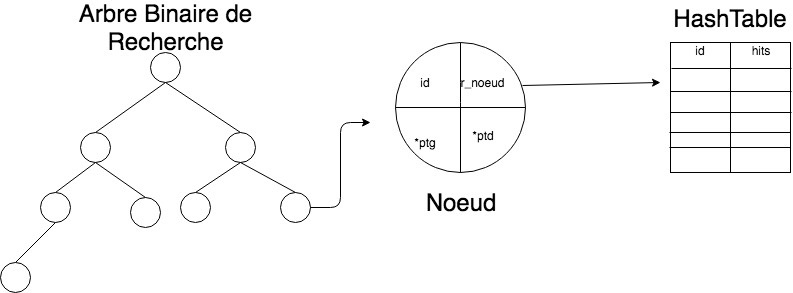
Cette classe représente un arc du graphe (ie. les données associées au referer). Elle contient l’identifiant du referer et le nombre de hits associés.

* DirectedGraph:

Il s’agit de l’abstraction d’un graphe orienté générique utilisable pour deux classes (S, T). S représente un nœud et T un arc. Chaque nœud est associé à une collection (set) d’arcs dont il est la destination.

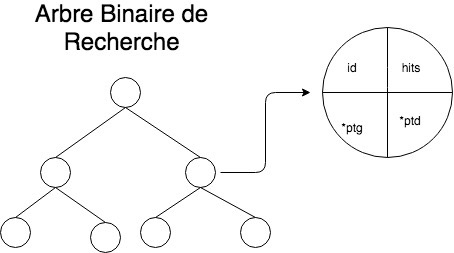
IV. Structure de Données

Pour concevoir notre structure de données, nous avons auparavant réfléchi aux opérations les plus utilisées. Dans notre cas, nous pouvons dire que c’est l’insertion et la recherche.

Ceci nous élimine déjà quelques possibilités comme des listes, listes chaînées, piles et files. Nous avons choisi un Arbre Binaire de Recherche (BST), qui fait les opérations mentionnées en temps quasi-constants (O(log n)).

Chaque nœud de notre arbre possède un identifiant de la cible et aussi une deuxième structure des données qui contient des couples <identifiant referer, nombre de consultations>. Pour cette deuxième structure, nous n'avons pas besoin de maintenir l'ordre ou d'accéder à des éléments, simplement de pouvoir ajouter et rechercher très rapidement. La table de Hachage fait cela en temps constant (O(1)) et est donc un choix privilégié.

Nous avons aussi un deuxième BST qui est utilisé pour stocker des pairs contenant l’identifiant de la cible et le nombre de consultations. La premiere structure de données est utilisée surtout pour générer le graphe et la deuxième pour obtenir les n premiers éléments selon le nombre de consultations.



Le Cache des strings :

Vu que nous n'avons pas besoin de maintenir l'ordre ou d'accéder à des éléments, simplement de pouvoir ajouter et rechercher très rapidement, nous avons choisi la Table de Hachage, qui fait cela en temps constant (O(1)).

1. Conclusion

En conclusion, ce TP fût une première plongée très enrichissante dans la STL. Le sujet du TP a réussi à soulever de nombreuses problématiques de conception, et la recherche du meilleur compromis performance/mémoire/réutilisabilité fût à la fois difficile et très intéressant.