Douzon Thibault B3331
Vogel Hugues

Compte rendu TP 4
Analyse de logs Apache

I. Introduction

Le but de ce TP est de réaliser une application capable de lire un fichier journal provenant d'un serveur Apache afin d'effectuer des opérations de statistique élémentaires. Ce dernier contient un grand nombre d'informations et dans une optique de réutilisabilité, nous nous efforcerons à fournir travail capable de toutes les consigner même si certaines ne nous sont pas utiles pour notre objectif.

L'objectif secondaire est d'apprendre à manipuler les fonctionnalités de la STL en C++. Cette dernière fournissant de nombreuses classes et fonctions qu'il faut absolument réutiliser.

II. Spécifications générales

L'exécution devra se dérouler ainsi :

- Lecture des arguments fournis par l'utilisateur. Faire remonter une erreur s'il manque un argument essentiel (e.g. le chemin d'accès au fichier) ou s'il manque une valeur après une option (e.g. -t sans nombre derrière). Si tout est correct, la phase suivante débute.
- Ouverture du fichier qui détermine les URL locales en lecture afin de stocker les valeurs contenues dans ce fichier. Si le fichier n'existe pas, ou si l'application n'est pas autorisée à le lire, faire remonter une erreur et en déterminer la cause.
- Ouverture du fichier de log Apache en lecture. De la même manière que précédemment, si cette opération échoue, il faut informer l'utilisateur de la cause de l'échec. Si tout est correct jusqu'à maintenant la lecture débute.
- Pour chaque ligne du fichier de log : lire chaque information et la stocker adéquatement, vérifier si elle passe à travers les filtres que l'utilisateur a demandés, si elle passe tous les tests, ajouter cette ligne à nos statistiques, sinon ne pas l'ajouter. Nous considérerons que le fichier respecte la syntaxe d'un fichier log Apache et ne ferons pas de vérification sur la sémantique des données recueillies.
- Après avoir lu le fichier, afficher selon les options décidées par l'utilisateur : les pages les plus visitées ou le graphe de parcours.

Nous testerons particulièrement certains points essentiels :

Fonctionnalité testée	Identifiant du test correspondant
Arguments incorrects : aucun argument donné	TestNoArg
Arguments incorrects : -l sans de nom de fichier local	Test-lNoArg
Arguments incorrects : -t suivi d'une chaine de caractères	Test-tNoInt
Arguments utilisateur : argument –t utilisé	Test-t
Arguments utilisateur : argument –e utilisé	Test-e
Ouverture du fichier URL locales : le fichier n'existe pas	TestNoLocal
Ouverture du fichier URL locales : les droits en lecture sur le fichier sont insuffisants	TestNoRightLocal
Lecture des URL locales : le fichier est vide	TestLocalEmpty
Lecture des URL locales : le fichier contient quelques lignes	TestLocalFilled
Ouverture du fichier log : le fichier n'existe pas	TestNoLog

Ouverture du fichier log : les droits en lecture sont insuffisants	TestNoRightLog
Lecture du fichier log : le fichier est vide	TestLogEmpty
Lecture du fichier log : le fichier contient quelques lignes	TestLogFilled
Graphe : aucune page n'est hit	TestGraphNOHit
Graphe : quelques pages accédées avec des liens entre elles, sans arguments	TestGraphSimple
Graphe : quelques pages accédées, avec chargement des URL locales	TestGraphLocal
Graphe : utilisation complète de l'application	TestGraphComplet

III. Conception de l'application

Lors de notre réflexion préliminaire, nous avons distingué 4 étapes essentielles à l'application qui se distinguent les unes des autres :

1) La lecture du fichier log. Un fichier log Apache étant un objet standard qui respecte une syntaxe, il est nécessaire de le traiter comme tel et de définir des opérations qui ne s'appliquent que ce type de fichier.
2) Le stockage et le tri des lignes lues précédemment. Chaque opération de lecture doit aboutir à un élément contenant toutes les informations décodées par la lecture. Il doit être possible de déterminer si cet élément répond aux filtres de l'utilisateur.
3) Le stockage des informations qui nous concernent pour l'application. Toutes les données utiles aux calculs statistiques effectués par la suite doivent être stockées en attendant d'être traitées.
4) Le traitement et l'affichage de la réponse à l'utilisateur. En utilisant toutes les informations recueillies, il faut élaborer une réponse correcte répondant à la requête de l'utilisateur.

Nous avons abouti à une application composée de 4 classes : InsertFluxApache, LigneRequete, IndexString et StatistiquesApache

- InsertFluxApache est une classe d'input stream spécifique aux fichiers logs Apache.
  Un objet instance de la classe possèdera comme attribut un conteneur où sont stockées les URLs locales.
  Est fourni un constructeur qui permet de spécifier le chemin d'accès au fichier contenant les URLs locales et le fichier de log Apache. En plus des constructeurs et destructeurs, la fonction membre GetLigneApache permet de lire une ligne du fichier et de renvoyer un objet LigneRequete contenant les informations lues.
  L'avantage d'une telle classe est la réutilisabilité du code ainsi développé : pour tous les travaux de lecture sur un fichier Apache, cette classe est suffisante.
- LigneRequete est la classe contenant toutes les données d'une ligne du fichier de log.
  Quelques attributs d'un objet instance : pageCible qui contient l'intitulé de la page demandée par un utilisateur, codeIP qui contient l'addresse IP de l'utilisateur, tailleReponse qui contient la taille en octet de la réponse renvoyée par le serveur etc.
  Un constructeur permet d'initialiser tous les attributs et la fonction membre Filtre permet de tester l'objet courant sur les différents filtres proposés à l'utilisateur. Si l'objet ne satisfait pas une des conditions, la valeur false est retournée.
  Cette classe simple et générale permet pour tout fichier Apache, de stocker l'ensemble des données disponibles sur une ligne du log. Encore, une fois, nous avons choisi d'élaborer cette classe afin de garantir la réutilisabilité du code produit.
- IndexString permet d'attribuer un index à chaque nom de page rencontrée. L'élaboration de la classe s'est faite afin d'éviter le stockage de données trop volumineuses et trop redondantes. Le fonctionnement est simple : à chaque nom de page rencontré est attribué un entier unique. Ensuite, partout où l'on veut faire référence au nom de page, on peut

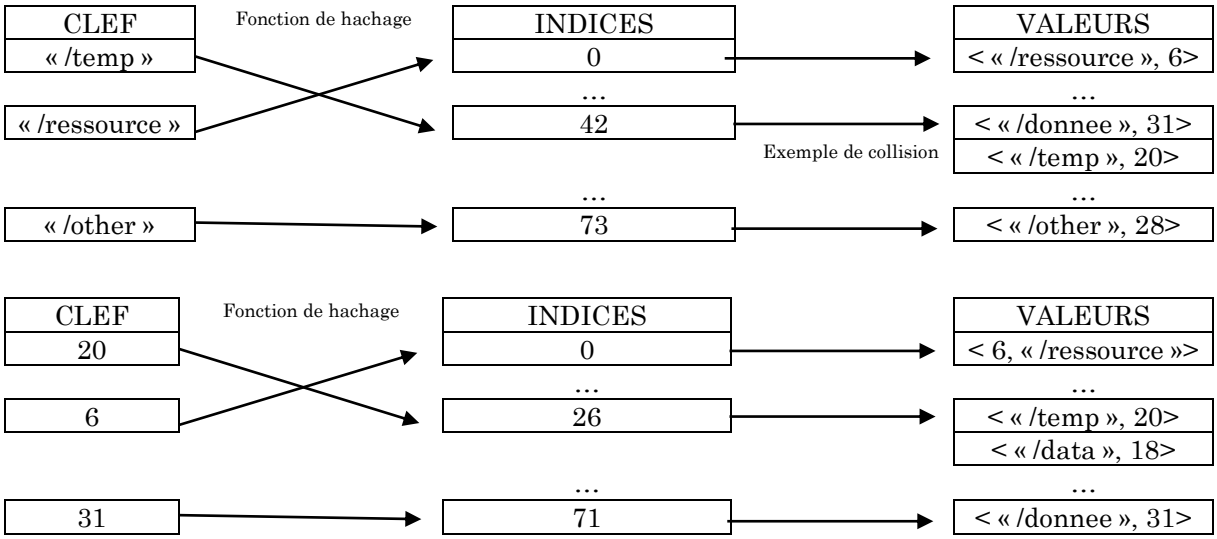
écrire l'entier à la place qui est moins volumineux. Lorsque l'on souhaite retrouver l'information sur la page, il suffit d'interroger l'objet instance d'IndexString en lui fournissant l'entier pour retrouver la chaine de caractère.
Une instance est constituée de deux dictionnaires complémentaires l'un de l'autre : mapBase, le premier ayant pour clef une string et pour valeur un entier ; et le second, mapReverse, ayant pour clef l'entier et pour valeur la string. À cela s'ajoute un compteur qui s'incrémente quand on insère des éléments dans les dictionnaires. C'est ce compteur qui directement donne l'index d'une chaine de caractère nouvellement ajoutée.
Un constructeur permet d'initialiser un objet vide et deux méthodes permettent d'interroger les dictionnaires : getIndex qui à partir d'une chaine de caractère renvoie l'index attribué à cette chaine (et effectue une insertion si la chaine n'existait pas dans les dictionnaires). Et getString qui à partir d'un index renvoie la chaine de caractère correspondante.
Grâce à cette classe, partout où nous aurions souhaité écrire une chaine de caractère, nous pouvons nous permettre d'écrire simplement un entier.

- StatistiquesApache est la classe qui effectue le dénombrement des pages visitées et qui permet d'écrire un fichier GraphViz. On rappelle que l'on souhaite connaitre pour chaque page le nombre de fois qu'elle a été visitée, et vers quelles pages les visiteurs se sont ensuite dirigés.
  Afin de répondre à ces besoins, nous avons décidé d'utiliser un premier dictionnaire nbHit qui à partir d'un entier clef (qui fait référence à un nom de page via un objet IndexString) est associé un autre entier (qui lui indique directement le nombre de fois que la page est accédée). Le deuxième attribut de la classe, nbLink, permet de compter le nombre de fois qu'un utilisateur est passé d'une page A à une page B. Cela se présente sous la forme d'un dictionnaire qui à un entier (la page A) associe un second dictionnaire, ce dernier associant un entier (la page B) au nombre de fois que le lien A->B est pris. Enfin, afin de pouvoir se permettre d'utiliser des entiers à la place des noms de page, le dernier attribut de la classe est un objet IndexString.
  Trois méthodes sont définies en plus des constructeurs et destructeurs : AjouteLigne permet d'ajouter le contenu d'un objet LigneRequete afin de le prendre en compte dans les statistiques, Afficher permet d'afficher dans l'ordre décroissante de nombre de hit les n premières pages, où n est passé en paramètre et enfin, ExportGraph qui permet d'écrire dans un fichier les caractères nécessaire à la visualisation du graphe reflétant ce qui a été lu dans le fichier log.

IV. Structures de données

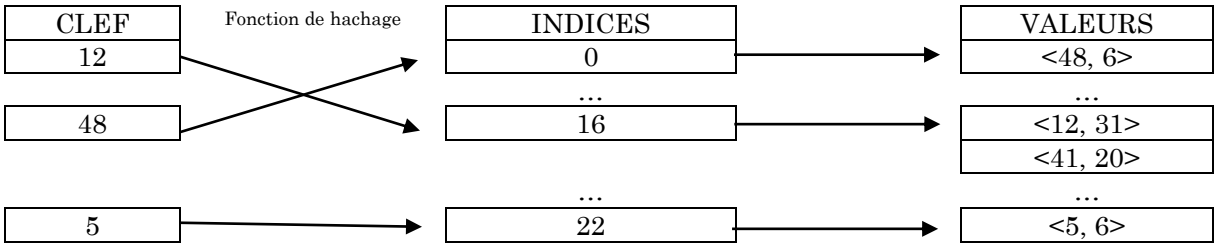
À partir des besoins suggérés par la spécification et la conception de l'application, nous sommes parvenus à trois structures de données différentes au sein de l'application : une première afin de limiter la duplication et la redondance des informations devra permettre d'établir une bijection entre les noms de pages que nous avons rencontrées et les entiers naturels. Une deuxième permettra de comptabiliser pour chaque page le nombre de fois qu'un utilisateur a accédé à la ressource. La dernière devra comptabiliser pour chaque couple de page <source, destination> le nombre de fois que la page 'destination' est accédée depuis la ressource 'source'.

- Cette première structure défini notre manière de géré les noms de ressources. Elle doit être très efficace en insertion et en recherche d'élément dans les deux sens : rechercher si une ressource est déjà présente dans la structure et rechercher le nom de la ressource associée à un nombre donné. Aucune suppression n'est prévue, seuls des ajouts seront effectués. Ces éléments nous ont conduits à utiliser deux tables de hachage, l'une permettant en hachant le nom de la ressource de retrouver l'entier associé, et l'autre permettant en hachant un entier de retrouver le nom de la ressource. Étant donné les relations très fortes entre ces deux tables, il est nécessaire qu'elles contiennent à tout instant les mêmes données.
  Ci-dessous un schéma de la structure :



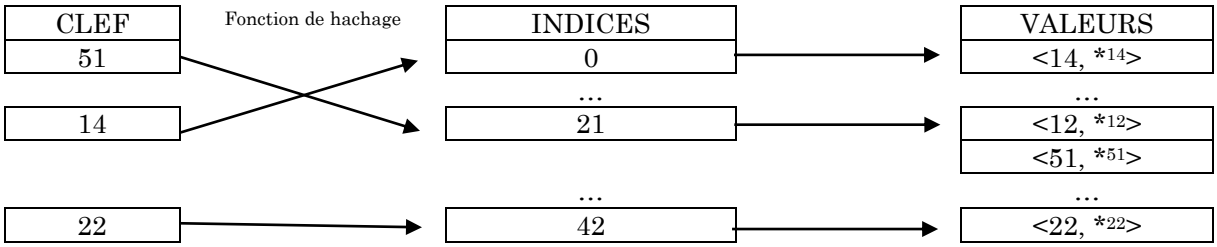
L'utilisation des tables de hachage permet de rechercher et d'insérer des données en temps constant O(1) aussi bien dans un sens que dans l'autre. Un autre choix aurait pu être celui d'un arbre binaire de recherche qui lui effectue ces opérations en temps logarithmique O(log(n)). Seulement, les données à traiter provenant d'un serveur Apache, il est probable qu'elles soient en très grande quantité. C'est pourquoi nous avons choisi d'utiliser les solutions algorithmiques en temps constant. Ce schéma peut paraitre très gourmand en mémoire, l'information étant dédoublée afin de gagner de temps sur l'exécution, cependant son implémentation permettra de gagner beaucoup d'espace mémoire : chaque chaine de caractère pourra être remplacée par un entier qui à travers cette structure fera référence à la chaine de caractère. C'est un gain de mémoire énorme quand une page est accédée plusieurs milliers de fois.

La deuxième structure a pour objectif de compter les hits de chaque page. C'est une structure non ordonnée qui met en relation une page et un entier qui s'incrémente à chaque fois que cette page est accédée. Elle sera appelée en recherche pour chaque ligne lue qui passe les filtres, il faut donc qu'elle soit très efficace dans la recherche d'un élément. Elle devrait être dans de moindres quantités appelée en insertion de ressource, si cette donnée n'est pas déjà présente dans la structure (après la recherche de l'élément donc). Nous avons choisi d'utiliser encore une fois une table de hachage car cette structure permet de faire efficacement des opérations de recherche en temps constant. Étant donné la structure précédente, les noms de ressources peuvent être convertis en entier, c'est pourquoi sur le schéma de la structure, nous utilisons des entiers comme clefs :

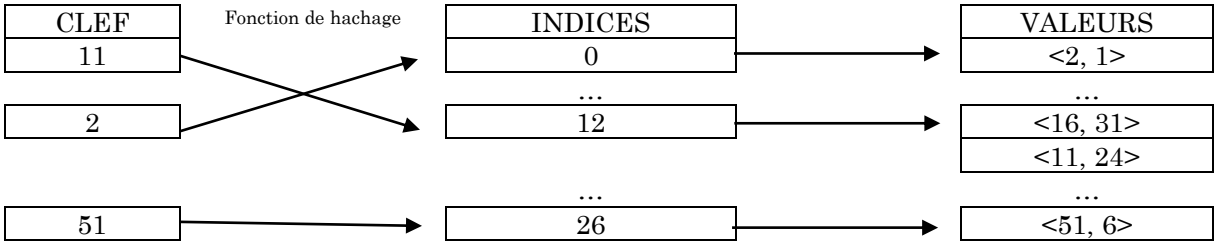


Les clefs sont forcément uniques, une ressource ne peut apparaitre deux fois dans la structure. Quand une ressource déjà présente est rencontrée à nouveau dans le fichier, il faut alors incrémenter la valeur associée au nom de la ressource afin de compter le nombre de hit. Encore une fois, il est possible qu'il y ait des collisions lors de la fonction de hachage : deux ressources différentes qui auraient une image identique à travers la fonction. Cela a pour conséquence, si elles sont nombreuses, une chute des performances.

- La dernière structure de donnée a pour but de compter le nombre parcours d'une page à une autre. Encore une fois, elle sera très sollicitée en recherche et en insertion. Aucune suppression n'est à prévoir, mais de nombreuses modifications seront faites. L'enjeu est de pouvoir rechercher pour chaque ressource possible, si depuis cette dernière a déjà été accédée une autre ressource. Et si oui, combien de fois. Afin de pouvoir répondre à cette question efficacement, nous avons choisi d'utiliser une première table de hachage qui prend pour clef la ressource source et qui donne sur une seconde table de hachage, cette dernière prenant pour clef la ressource cible. Au total ce ne sont pas deux tables de hachages, mais N+1, N étant le nombre de ressources qui sont référencées, car pour chaque ressource source une table de hachage est nécessaire afin de rechercher la page cible ad hoc. Cette structure peut se représenter ainsi :



Où $*_i$ est une table de hachage comme la suivante :



Par exemple, disons que $*_{14}$ soit représenté ci-dessus, alors pour que 14 apparaisse en clef dans cette table, il faudrait qu'une personne essaie d'accéder à la ressource 14 alors qu'il se trouve déjà sur cette dernière. L'avantage d'une telle structure complexe sur de simples tableaux 2D est sa faible utilisation mémoire lorsque les liens possibles entre les pages ne sont pas uniformément répartis : si c'était le cas, de n'importe quelle page, on pourrait accéder à n'importe quelle autre. Alors effectivement toutes les cases d'une matrice avec en colonne les sources et en ligne les destinations seraient remplies. Seulement, ce n'est pas une représentation fidèle du web : d'une page ne sont accessibles que certaines pages bien définies. Alors, la précédente matrice serait un gâchis de mémoire, la grande majorité des cases étant remplies de 0.

V. Architecture de l'application

Voici le schéma de classes de l'application finale :

InsertFluxApache
Fonctions membres publiques : InsertFluxApache(string nomLog, string nomURL) ~InsertFluxApache() LigneRequete getLigneApache() Attributs privés : Vector<string> localURL

IndexString
Fonctions membres publiques : IndexString() ~IndexString() unsigned int getIndex(const string obj) string getString(unsigned int index)const const unordered_map<string, unsigned int> & getMapBase()const Attributs privés : unordered_map<string,unsigned int> mapBase unordered_map<unsigned int,string> mapReavers unsigned int compteur

LigneRequete
Fonctions membres publiques : LigneRequete(<tous les attributs de la classe>) ~LigneRequete() bool filtre(bool excludeFileTypes, bool useTime, int hour)const string getCible()const string getReferer()const Attributs privés : string codeIP string user string pseudo tm date string methodeHTTP string pageCible string argumentsCible string extensionCible string versionHTTP int codeErreur int tailleReponse string pageReferer string argumentsReferer string browser

StatistiquesApache
Fonctions membres publiques : StatistiquesApache() ~StatistiquesApache() bool Ajouterligne(const LigneRequete & ligne, bool excludeFileTypes, bool useTime, int hour) void Afficher(unsigned int nbLignes) bool ExportGraph(const string & nomFichierGraph)const Fonctions membres privées : String & getGraph(string a stringRetour)const Attributs privés : IndexString indexPageName MapHit nbHit MapLink nbLink

Couple
Fonctions membres publiques : Couple(unsigned int page, unsigned int hit) ~Couple() unsigned int getPage()const unsigned int getHit()const Fonctions amies de la classe : bool operator<(const Couple & c1, const Couple & c2) Attributs privés : unsigned int page unsigned int hit

VI. Améliorations

Nous aurions souhaité pouvoir exécuter les différents tests depuis une dépendance d'un makefile. Cette fonctionnalité ne fonctionnant pas, il est tout de même possible de lancer tous les tests en exécutant le script mktest.sh directement.

VII. Conclusions

Ce TP nous a fait réfléchir sur les différentes implémentations des structures classiques de données que propose la STL. En cherchant à mieux définir nos besoins nous avons pu élaborer une application efficace.

Penser à la réutilisabilité du code, et à son utilisation ultérieure pour d'autres besoins, nous a conduits à des classes générales qui peuvent offrir les outils nécessaires à la réalisation d'autres projets.