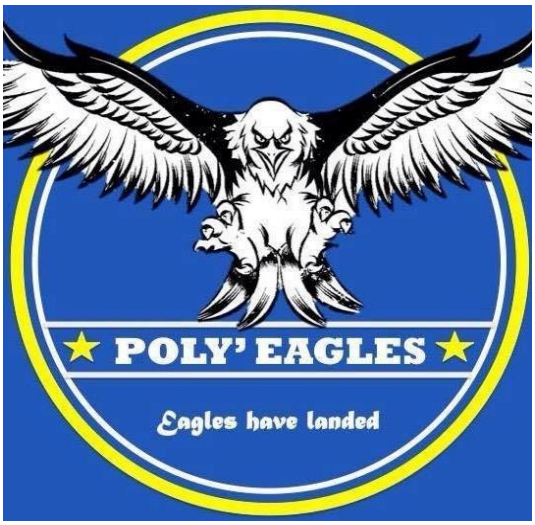
Huet Alexandre
IMA3 Polytech Lille.

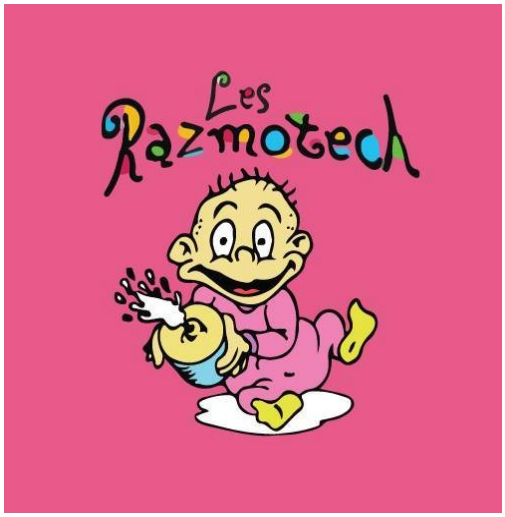
17/06/2016

Analyse de flux Facebook

(Listes BDE 2016 Poly'eagles et Razmotech)



VS



SOMMAIRE

Introduction :....3
I) Choix des structures de données utilisés....4
II) Fonctions....5
1) Chargement....5
a) Fonctions annexes....5
b) Particularités de la fonction....5
c) Difficultés....6
2) Compte....6
3) post_like....7
4) like_1post....7
5) liste_truant....8
III) Perspectives d'évolution, Limites....8
Conclusion :....9

Introduction :

Pour illustrer nos cours de programmation avancée lors de notre formation à Polytech Lille dans le cursus Informatique, Microélectronique et Automatique, nous allons mettre en œuvre ce que nous avons vu lors du cours théorique ainsi que les tp pour nous pencher sur un projet portant sur l'analyse de flux Facebook.

L'objectif de ce projet est de charger 2 fichiers YAML dans une structure de donnée choisi par nos soins et de les utiliser pour traiter des requêtes que pourra exiger l'utilisateur. Cela nous permettra d'approfondir nos connaissances en langage C, plus précisément des structures de données complexes, des lectures et écritures de fichiers, compilation séparée et automatique (Makefile) , et l'utilitaire de gestionnaires de version (git).

Dans cette application, nous compterons le nombre de posts d'une liste et le nombre total de likes obtenus. Nous exposerons les posts likés par une personne choisi par l'utilisateur. Nous compterons et listerons les utilisateurs qui ont liké au moins un post dans une liste donnée. Nous listerons les utilisateurs qui ont liké des posts dans les deux listes. Pour finir, nous définirons quelle liste a été la plus apprécié sur ce réseau social et si le résultat de l'élection est cohérent avec l'estimation de l'application.

I) Choix des structures de données utilisés

Nous définirons 2 types principaux. Le type "post" et le type "personne". Nous utiliserons 1 structure de donnée pour référencer tous les posts pour chaque liste et chaque post comportera une structure de donnée de personne ayant liké le post.

D'après le cahier des charges, nous n'avons pas besoin de rechercher des posts précis. On nous demande juste de faire un traitement sur chaque post sans faire de distinctions entre eux. Cette analyse nous permet de penser que nous ne devons pas mettre l'accent sur une structure de donnée qui permette une recherche rapide, mais surtout une structure de donnée dynamique, à ajout rapide, non trié pour permettre de charger les fichiers Yaml le plus rapidement possible.

Par contre, la structure de donnée lié aux personnes ayant liké 1 post requiert d'avoir une recherche la plus rapide possible. La structure devra donc être nécessairement trié pour permettre d'utiliser la recherche dichotomique. Les arbres ont l'avantage d'avoir une recherche/ajout/suppression d'efficacité entre O(log2n) et O(n) selon le fait que l'arbre soit équilibré ou pas. Les listes contiguës ont quant à elle une recherche en O(log2n), un ajout/suppression en O(n) et possèdent dans leur structure directement le nombre d'élément total ce qui facilite grandement l'efficacité de certaines questions. Nous choisirons ici les listes contiguës car nous ne pouvons pas nous permettre d'avoir un arbre non équilibré et le coût de contrôle serait supérieur à l'efficacité d'une liste contiguë.

Pour conclure, nous choisirons donc une liste chaînée qui liste les posts d'une liste et chaque Cellule Post contiendra une liste contiguë triée des like que le post à reçu.

II) Fonctions

Premièrement, la première chose à aborder dans le projet est de charger le fichier Yaml dans notre structure de donnée. Pour cela nous devrons utiliser des fonctions annexes pour simplifier la lisibilité du programme et son débugage. Nous parlerons ensuite des fonctions répondants au cahier des charges.

1) Chargement

a) Fonctions annexes

Pour ajouter de nouvelles cellules à la liste chaînée de post, nous utiliserons la fonction d'ajout_tete et la fonction newpost. La fonction newpost sert à allouer la mémoire nécessaire à une Cellule mémoire et retourne son adresse qui sera offerte à un pointeur pt_newpost. Une fois le post rempli par les données du fichier, nous procéderons par un ajout_tete pour ajouter le nouveau post à la liste BDE.

Pour la liste contiguë, l'enjeu est de la trier. Pour cela nous utiliserons une fonction de recherche dichotomique et une fonction de décalage. Le principe de la manipulation est que l'on trouve l'indice où la nouvelle personne devrai être pour être triée, nous décalons la liste jusqu'à cet indice et nous insérons la valeur à cet endroit. Nous trierons par id. Cela nous permet d'être plus précis lors du trie et de différencier les personnes homonymes.

b) Particularités de la fonction

Le principe général de la fonction est de lire mot par mot le fichier YAML. On stocke la chaîne courante (s) et la chaîne précédentes (sprec) dans deux variables différentes. Tant que l'on est pas à la fin du fichier, nous comparerons la chaîne précédente aux chaînes tel que « id : », « messsage : » , « likes : », etc.

Nous prêterons une attention particulière au id des post et des personnes. Nous remarquerons que les id des post sont plus long que les id des personnes. En effet, les id des post comportent plus de 30 caractères tandis que les id des personnes en contiennent moins de 25.

c) Difficultés

Nous avons rencontré plusieurs problèmes lors de la conception de la fonction.

- Nous avions omis d'initialiser la liste à NULL ce qui nous faisait une segmentation fault.

- Les chaînes s et sprec était trop petite. Nous avions alloué 75 espaces et cela marchait pour la liste poly'eagles mais pas pour la liste razmotech qui ont mis dans leur message des chaînes de tiret de plus de 100 caractères.

- Nous avions pas initialisé les s et sprec avec le caractère '\0'

- Il fallait aussi faire attention aux posts qui ne comportent pas de j'aime.

- Nous avions toujours les même id de post au début. Le fait est que quand on remplissait la liste contiguë de like, notre condition d'arrêt était que l'on tombe sur un id de post. Nous retournions donc au début de la boucle ou nous scannions une nouvelle chaîne de caractère. Nous écrasions donc l'id que nous souhaitions ajout aux nouveaux post, d'où l'arrivé de nos boolean.

- Nous avions des fuites mémoire, il fallait bien penser à libérer l'espace de pt_newpost à chaque ajout de post.

2) Compte

En vue du fait que nous connaissons l'indice de la dernière personne de la liste des personnes qui ont liké un post, il nous suffit de parcourir la liste chaîné et n'additionner aux variables nbpost et nblike. On utilise des pointeurs d'entier en paramètres pour pouvoir « retourner » 2 variables au lieu d'une.

3) post_like

Cette fonction imprime les posts likés par la personne prise en argument et liste les indices des posts correspondants. Ici nous utilisons la fonction de dichotomie pour rechercher l'utilisateur dans la liste de j'aime de chaque post précédemment trié. Si la personne recherchée est à l'indice où elle devrait être, alors on imprime le poste et on ajoute la valeur d'indice au tableau.

La recherche d'indice par dichotomie nous permet d'avoir une efficacité de O(log2(n1)*n2) avec n1 la moyenne de nombre de like par post et n2 le nombre de post pour la liste placée en paramètre.

4) like_1post

La fonction like1_post traite un tableau de Personne passé en argument. Elle rempli et imprime ce tableau des personnes ayant liké au moins un post dans la liste BDE passé en paramètre.

Notre première idée était de travailler par recherche dichotomique comme précédemment. Malheureusement notre programme faisait l'erreur « segmentation fault » et nous n'avons pas réussi à détecter l'erreur. Nous avons donc décidé de parcourir entièrement les listes de personnes ayant aimé chaque post afin de trouver ceux qui ne sont pas encore dans le tableau passé en paramètre.

Le coût du processus est donc ici de O(n1*n2) au lieu de O(log2(n1)*n2) avec la recherche dichotomique selon notre idée initiale. La fonction renvoie l'entier correspondant à l'indice de la dernière valeur du tableau.

5) liste_truant

La fonction liste_truant[1] prend en paramètre 2 tableaux de personnes correspondants aux personnes ayant liké au moins 1 post pour chacune des listes. Elle imprime les personnes ayant liké dans les 2 listes. Nous nous sommes aidé de la fonction like_1post vue précédemment. Dans notre programme les tableaux associés ne sont pas triés, nous parcourons simplement les 2 tableaux dans leur intégralité.

Nous avons donc ici un coût moyen de O((dern1+1)*(dern2+1)/2) car l'espérance que nous trouvions la personne dans l'autre liste est de (dern2+1)/2.

Ici les tableaux sont relativement petits (350 et 276) donc nous pouvions nous permettre de ne pas les trier et d'y appliquer une méthode de recherche rapide mais ici, en vue de nos soucis précédents, nous ne n'y sommes pas attardé.

III) Perspectives d'évolution, Limites

Bien que réfléchi, notre solution n'est pas la plus efficace. La solution de la table de hachage semble séduisante avec une fonction de hachage pertinente et taille de liste contiguë bien évaluée. En effet, cela nous permettrait de trier par prénom (ce qui est plus accessible pour un utilisateur Lambda). Puis ensuite faire choisir l'id correspondant. De plus, l'efficacité en recherche/ajout/suppression peut s'en voir optimisé (plus de décalage à faire).

Au niveau de l'ergonomie, ce programme pourrait être améliorer. Beaucoup d'informations sont affichés sans que l'utilisateur arrive à discerner les informations essentielles rapidement.

1 Veuillez excuser mon humour. Est-ce pardonnable en fin de projet ?

Conclusion :

Ce projet nous aura permis d'approfondir nos notions en langage C et nous exposer à la vaste amplitude d'application qu'offre nos nouvelles compétences. Il nous aura offert aussi une expérience en travail de groupe, à travailler avec de nouveaux outils tels que git, les Makefile et avec des contraintes de cahier des charges.

Cette expérience nous aura aussi appris à travailler en temps limité, dans un contexte de travail autonome sur un thème passionant.