Algorithmes et programmation
Période 3 - PROJET
Exercice en binôme - 20 points

1 Le Sujet

Nous voulons construire un index. Étant donné un ensemble d'entrées (un ensemble de mots) et un texte, l'index fournit, pour chaque entrée, les lignes du texte où elle apparait. Par exemple, soit le texte:

> Love of my life - you've hurt me
> You've broken my heart and now you leave me
> Love of my life can't you see
> Bring it back, bring it back
> Don't take it away from me
> Because you don't know -
> What it means to me.

et les entrées *life*, *heart*, *bring* and *love*. Nous voulons pouvoir générer comme résultat l'index suivant:

Mot	Ligne
bring	4
heart	2
life	1, 3
love	1, 3

Vous devez construire cet index en utilisant deux structures différentes et en suivant les indications ci-dessous. Commencez par construire des fonctions qui sont utilisées dans les deux versions. Ensuite vous allez construire les fonctions spécifiques à chaque type de structure. L'idée est de pouvoir utiliser l'une ou l'autre structure, sans changer le programme principal.

1.1 L'idée générale des deux programmes

1. Lire un ficher qui contient les N mots clés qui seront les entrées de l'index (pour cela, voir l'exemple et l'explication dans la section 3). Le fichier en question contient, pour chaque ligne, un mot.

2. Créer l'index avec ses N entrées.

3. Une fois que les entrées de l'index sont organisées (c'est-à-dire, l'index est créé, organisé et contient toutes les entrées), vous devez remplir l'index en informant les lignes où chaque mot apparaît. Pour cela vous devez lire un fichier qui contient le texte à analyser, et chercher, pour chaque mot dans le texte, s'il est une entrée de notre index. Si oui, il faut stocker la ligne en question. Sinon, il faut continuer la lecture du fichier.

4. En plus, le programme doit fournir à l'utilisateur, les possibilités suivantes pour visualiser le résultat:

(a) Placer dans un fichier texte l'index obtenu à partir du texte donné, dans un format similaire à celui présenté dans l'exemple ci-dessus. Ce fichier texte contient ainsi le résultat du programme.

(b) Afficher l'index à l'écran (cela correspond donc au contenu du fichier texte en 4a).

Ci-dessous un aperçu du programme principal à implémenter. Les paramètres des fonctions ne sont pas spécifiés. Il est conseillé de faire un menu où chaque tâche est proposée.

```
//Création de la structure pour l'index
tIndex = CreateIndex (...);
//Construire l'index avec les mots clés.
buildKeyWordIndex (....);
//Compléter l'index avec les lignes par rapport à un fichier donné
searchKeyWordsInsertLines(..... );
//Affichage et génération d'un fichier avec le résultat.
visit (..... );
```

2 Le travail à faire

2.1 Les fonctions de base: outils pour la suite du projet

Pour manipuler les fichiers et déterminer des mots vous aurez besoin des fonctions comme celles ci-dessous:

- `int getLineInFile (...)`: lit une ligne d'un fichier texte et rend le numéro de la ligne lue ainsi que la ligne elle même dans une variable (`line`).
- `bool getNextWord (...)`: reçoit une chaîne de caractères (`line`), sa taille et un curseur indiquant la position à laquelle la recherche d'un mot doit commencer. Cette fonction cherche le prochain mot, dans `line` et indique s'il existe ou non.
- Toutes les fonctions pour déterminer les séparateurs des mots, pour ne pas faire de différence entre majuscules et minuscules...
- REMARQUE: Nous supposons que les textes traités sont sans accents et sans caractères spéciaux comme @, $, etc. Néanmoins, pensez à considérer la ponctuation.

2.2 VERSION Arbre: Index en utilisant arbre binaire de recherche (BST)

1. Un BST va stocker les entrées et sert de base pour la construction du index. Ainsi, chaque entrée est tout de suite insérée dans le BST (les clés du BST sont les entrées de notre index).
2. Utiliser la fonction *partition* que nous avons vu en cours, pour organiser ce BST afin que sa racine contienne la médiane (par rapport aux valeurs des N clés). Rappelez vous que la médiane est la valeur qui permet de partager une série ordonnée en deux parties de même nombre d'éléments.
3. Une fois que les entrées de l'index sont organisées (c'est-à-dire, le BST est créé, organisé et contient toutes les entrées), vous devez remplir l'index en informant les lignes où chaque mot apparaît. Chaque noeud de l'arbre est associé à une liste contenant les numéros de lignes où ce mot apparaît. *Vous devez utiliser une liste chainée pour stocker les lignes*.

4. Pour la visualisation (fichier texte et écran), l'index doit présenter les entrées *dans l'ordre alphabétique et les numéros de ligne dans l'ordre croissant*.

5. Pour cette version, vous devez aussi faire la sauvegarde (dans un fichier) de la structure de l'index (l'arbre sans les listes). Un utilisateur peut donc stocker l'arbre des mots pour l'utiliser avec différents textes. Implémenter la procédure `void backUp (Tree h)` pour stocker l'arbre des mots et `void getUp (Tree & h)` pour la récupérer. Ajouter ces possibilités d'opération à votre menu..

2.3 VERSION Hash: Index en utilisant une table de hachage (*hash table*)

Introduction au table de hachage Une table de hachage permet une association clé-élément. On accède à chaque élément de la table via sa clé. L'accès à un élément se fait en transformant la clé en une valeur de hachage par l'intermédiaire d'une fonction de hachage. Une *fonction de hachage* transforme la clé d'un élément à ranger dans un tableau en un indice du tableau. Cette fonction distribue les clés de manière aléatoire, mais elle est déterministe (la valeur de hachage d'une clé donnée est toujours la même). Par exemple, soit les clés *serge, odile, luc* que nous voulons stocker dans un tableau de 5 cases. En appliquant la fonction de hachage ci-dessous nous obtenons *hash(serge,5) = 1* , *hash(odile,5) = 3* et *hash(luc,5) = 0*, indiquant que *serge* sera stocké dans la position 1, *odile* dans la position 3 et *luc* dans la position 0.

```
int  hash(char v[], int M){
int i=0, h = 0, a = 127;
    while (v[i]!= '\0') {
      h = (a*h + v[i]) % M;
      i++;
     }
    return h;
  }
```

Remarquer que la fonction de hachage que nous utilisons calcule le nombre correspondant à une chaîne de caractères en considérant, de gauche à droite, caractère par caractère. Il est important d'avoir une bonne fonction de hachage. Elle doit transformer une clé en une indice dans un tableau ayant M positions (de 0 à $M-1$). La fonction idéale doit faire cela de manière aléatoire, c'est-à-dire, pour chaque entrée (chaque clé), toutes les sorties (les entiers entre 0 et $M-1$) doivent être également probables. Voir le livre base indiqué dans le cours (de Sedgewick) pour plus de explication sur le choix d'une fonction de hachage.

Le fait de créer un *hash* à partir d'une clé peut engendrer un problème de *collision*, c'est-à-dire qu'à partir de deux clés différentes, la fonction de hachage pourrait renvoyer la même valeur de *hash*, et donc par conséquent donner accès à la même position dans le *tableau*. Par exemple, avec notre fonction *hash(jean,5) = 1* . Pour minimiser les risques de collisions, il faut choisir soigneusement sa fonction de hachage (mais les collisions existent toujours).

Lorsque deux clés ont la même valeur de hachage, ces clés ne peuvent être stockées à la même position, on doit alors employer une stratégie de résolution des collisions. Une des politiques les plus utilisées (et que vous devez utiliser ici) consiste à associer à chaque position du tableau une liste contenant les données que vous voulez stocker. La figure 1 illustre cette situation. Comme Serge et Jean ont le même *hash number*, il sont placés dans la même liste.

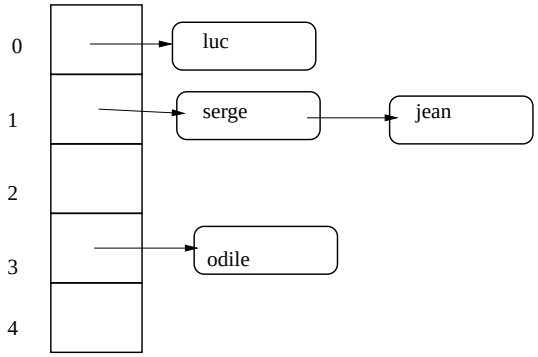

Figure 1: Exemple d'une table de hachage ou les collisions sont traitées via un chaînage

Les méthodes de hachage sont utilisables dans les cas où on doit gérer une collection C dont les clés appartiennent à un univers U très grand. La taille probable de C est relativement petite par rapport au nombre d'éléments de U.

Les étapes pour faire un index en utilisant une table de hachage

1. Une TABLE de hachage va stocker les entrées et sert de base pour la construction de l'index. Utiliser la fonction de hachage donnée ci-dessus. Réfléchir (et tester) à une bonne valeur de M (taille de la table). L'idée est d'avoir une table qui ne soit pas très grande, mais aussi de minimiser les collisions. Pour cela une bonne technique est de choisir comme valeur de M un nombre premier.

   Les collisions devront être réglées via une liste chainée selon la figure 1. Vous devez ainsi définir un tableau TAB de pointeurs. Chaque pointeur $TAB[i]$ indique le début d'une liste contenant les clés (et éventuellement d'autres informations).

2. Une fois que les entrées de l'index sont organisées, les lignes où chaque mot apparaît sont insérées. Chaque noeud de la liste est associé à une autre liste contenant les lignes où ce mot apparaît. Vous devez utiliser une liste chainée pour stocker les lignes. La figure 2 illustre cette situation.

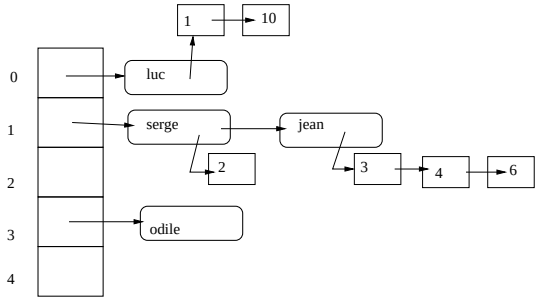

Figure 2: Exemple d'une table de hachage où les collisions sont traitées via chaînage et avec une liste associée à chaque noeud, pour stocker d'autres informations.

3. Affichez l'index et stockez le dans un fichier texte: vous n'avez pas besoin de présenter l'index dans l'ordre alphabétique, mais les numéros de ligne doivent apparaître dans l'ordre croissant, pour chaque entrée.

3 Information sur la lecture et l'écriture dans un fichier texte

Dans cet exercice vous devez pouvoir lire les données d'un fichier texte (*.txt*) donné en entrée. Cette lecture sera faite ligne par ligne. Pour lire un fichier texte, ligne par ligne vous pouvez utiliser *getline*:

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

Nous remarquons que *getline()* lit une ligne entière dans *stream* et stocke l'adresse du tampon contenant le texte dans $*lineptr$. Si $*lineptr$ vaut NULL, *getline()* alloue un tampon pour recevoir la ligne. Ce tampon devra être libéré par le programme utilisateur. La valeur dans $*n$ est ignorée.

Vous trouverez des explications sur *getline*, par exemple, sur le site
`http://manpagesfr.free.fr/man/man3/getline.3.html`.

Dans la suite nous présentons un exemple de l'utilisation de *getline()*. Notre fonction `getLineInFile` reçoit le nom d'un fichier *.txt*, par exemple, *test1.txt*. Le fichier est ouvert et ensuite chaque ligne est lue et affichée à l'écran.

```
void getLineInFile (const char *filename) {
// Reads the text file and prints each line of this file
    FILE * fp;
    char * line = NULL;
    size_t len = 0;
    ssize_t read;
    fp =  fopen (filename, "r"); // open the file
    if (fp == NULL)
       throw (" ERROR ");
    else {
       while ((read = getline(&line, &len, fp)) != -1) {
            cout << "Line :  " << line << endl;
        }
      if (line)  delete(line);
      fclose(fp);
    }
}
```

Pour écrire un fichier texte contenant l'index final, vous pouvez utiliser *fputs()*. Nous référençons
`http://www.cplusplus.com/reference/clibrary/cstdio/fputs/`
pour des explications et des exemples.

4 Le planning

4.1 Les dates limites

Le travail sera rendu en deux étapes:

1. **Le 11 mars 2013 (20h)**: Les fonctions décrites dans la section 2.1. Dépôt sur ENT.

   Faites aussi un programme qui teste les fonctions. Pour tester `getLineInFile` ce programme doit lire un fichier, ligne par ligne et afficher chaque ligne avec son numéro. Pour tester `getNextWord` ce programme doit afficher les mots d'une ligne.

2. **La première semaine d'avril (jour à préciser)**: Le travail complet. Dépôt sur ENT et soutenance organisée par votre responsable TP de la période P3.

La soutenance consiste en une démonstration sur un fichier texte fourni par l'enseignant et des questions à tous les membres du groupe. **Les notes pour chaque membre peuvent être différentes.**

4.2 Les documents et fichiers à rendre

- Pour le 11 mars, seulement un fichier `nom.zip` contenant vos sources (les .h, .cpp ...) et des éventuelles explications dans un fichier texte.
- Un document expliquant votre travail, les choix faits. Il faut aussi faire une analyse des deux méthodes utilisés, en présentant **la complexité pour chaque version du programme avec des explications**. Vous devez, en particulier, expliquez les points suivants:
  - Le fait d'organiser le BST de manière à placer la médiane à la racine a une importance quelconque (est-ce que cela peut changer la performance de l'algorithme, quel peut être l'intérêt ou l'inconvénient de faire cela?)?
  - Dans la version HASH, comment faire pour obtenir l'index dans l'ordre alphabétique? Analyser votre proposition en terme de complexité.
  - Comparer les deux méthodes d'implémentation d'un index. N'hésitez pas à lire sur le sujet (voir les références données en cours).
- Un fichier `nom.zip` contenant toutes sources (les .h, .cpp ...), y compris la partie déjà rendue le 11 mars.
- **Une soutenance est prévue pour présenter votre projet. Voir avec votre responsable de TP de la période P3 pour la date et les horaires**.
- Le programme doit être préparé pour fonctionner avec des fichiers textes donnés/fournis par chaque enseignant. Autrement dit, le programme sera testé sur des fichiers *.txt* fourni par l'enseignant au moment de la correction du programme.