Algorithme & Structure de données I
Projet

Etienne PENAULT

Semestre 3

Sommaire

Introduction

Le but du projet est d'implémenter, à notre manière, une méthode permettant d'indexer des mots, provennant de plusieurs fichiers textes différents, tout en stockant le nombres d'occurrences de chaques mots récupérés, autrement dit, d'implémenter nous même une sorte de table de hachage.

Pour cela, nous allons d'abord utiliser deux méthodes avec un résultat identique à ce que nous recherchons: Celle utilisant la table de hachage de la bibliothèque standard (ici appelé hmap), et une autre utilisant une classe nommée "trie", découlant d'éléments de la bibliothèque standards.

L'interêt d'une table de hachage repose sur la rapidité d'accés des éléments stockés. Ils y sont stockés selon leurs "hash", c'est à dire la clef défini à partir du mot en question. On peut donc accéder aux éléments par leurs clefs (c'est le cas pour la première méthode utilisée, mais pas de hash ni de clefs dans la seconde méthode).

Une fois cela fait, nous comparerons le temps d'exécution de chaque méthode pour voir si notre implémentation, avec un résulat identique, est plus efficace, ou non, que ces deux dernières.

Au cours de ce rapport nous verrons comment nous avons réussi à implémenter nos 3 versions différentes d'indexation de mots, les difficultés rencontrées au cours de ce projet, ou encore, les explications liées à la conception de chaque méthode.

Mode D'Emploi

"./ProjetAS1" pour lancer le programme.
"make" pour compiler.
"make clean" pour nettoyer les traces de compilations.

Hmap

Pour l'implémentation de notre première méthode, nous utiliserons le

type "hmap", défini comme suit:

```
using hmap = std::unordered_map<std::string,int>;
```

Notre type "hmap" est donc une unordered map, composée de chaines de caractères et d'entiers. Une fois ce type implémenté, nous pouvons commencer! L'utilisation de hmap repose sur la fonction "methode_1" qui a pour prototype:

```
hmap methode_1(const vector<std::string>);
```

Naturellement, cette fonction retournera notre "hmap" remplie et prendra en argument un vecteur contenant le chemin de nos différents fichiers.

L'intérieur de la fonction comporte plusieurs parties. On commence par déclarer une "hmap" que l'on nomme table comme ci-dessous:

```
hmap table;
```

NB: C'est cette même table qui sera remplie et retournée.

La boucle pricipale de cette fonction est la suivante:

```
for(std::vector<std::__cxx11::basic_string<char> >::size_type
i = 0; i<fichier.size(); ++i)
```

L'utilisation de cette boucle est nécessaire au parcours de tous les chemins des diffèrents fichiers en appelant le tableau par fichier[i].
NB: ON utilise "std::vector<std::__cxx11::basic_string<char> >::size_type" au dépend d'un entier car le compilateur considère que l'opération "i<fichier.size()" n'est pas sécurisée car on compare deux types différents, il faut donc déclarer "i" du même type que notre tableau.

Une fois dans la boucle, nous allons devoir ouvrir le fichier correspondant au chemin de "fichier[i]", à l'aide de la bibliothèque "fstream". Ensuite, si le fichier en question est ouvert alors on déclare une chaine de caractères "word" oÃ¹ $nousallonsstockerparlasuitelemotquenousseronsentraindetraiter, et, uncaractère"c"quiseral$

```
if(file){
    std::string word;
    char c = file.get();
    //TRAITEMENT
```

```
        file.close();
      }else{
        std::cout << "Erreur lors de l'ouverture du fichier.\n";
    }
```

Pour ce qui est du traitement du fichier en question (si le fichier est ouvert), nous allons se deplacer dans le fichier avec "file.good()" et "c" notre caractère courant.

```
while (file.good()){
    if((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')){
        up_to_low(c);
        word.push_back(c);
    }
    else{
        if(word.empty() == 0){
            table[word]++;
            word.clear();
        }
    }
    c = file.get();
}
file.close();
```

Nous allons décomposer ce traitement en deux parties, le "if" et le "else".

Le "if" sert à la récupèration d'un mot. La condition du "if" vérifie que le caractère courant est bien un caractère de l'alphabet, majuscule ou minuscule confondu. GrÃce à la fonction "up_to_low" nous vérifions que le caractère est minuscule, s'il ne l'est pas, alors on le transforme en minuscule. Voici la fonction "up_to_low":

```
  void up_to_low(char& c){
      if(c >= 'A' && c <= 'Z')
      c += 32;
  }
```

Ensuite, on ajoute notre caractère courant "c" à notre chaine de caractère "word" grÃce à "push_back".
NB: Le contenu de ce "if" reste inchangé selon la méthode utilisée.

Le "else" quand à lui sert à l'insertion du mot "word" dans notre hmap "table". En effet, nous sommes confronté au "else" quand notre caractère courant n'est pas une lettre, ce qui indique que la récupération du mot stocké dans "word" est fini. Dans ce "else", nous allons donc vérifier que le mot n'est pas vide avant toute chose, puis nous allons insérer notre mot "word" à l'aide de l'opérateur [ ] qui prend dans ce cas là une string (car notre type "hmap" est une "unordered_map" prennant des strings et des entiers). Nous pouvons aussi noter, que nous passons par "++" pour incrémenter l'occurence du mot car l'opérateur [ ] est défini tel que "T& operator[ ](const Key key);" et il retourne T& avec pour nous, T un type "int" (source: https://fr.cppreference.com/w/cpp/container/unordered_map/operator_at). Puis finalement nous vidons la string "word", car notre mot a été rajouté à notre hmap "table".

Enfin nous fermons le fichier en question une fois le fichier fini à l'aide de ".close()".

Pour finir, une fois tout les fichiers traités par notre boucle "for", on retourne notre "table" de type hmap.

La Structure Trie

Notre deuxième méthode est basée sur l'utilisation de "stdtrie", un type découlant de la classe trie défini comme ci:

```
using stdtrie = class trie{

private:
    std::unordered_map<char, std::unique_ptr<trie> > next_char_;
    int count_;

public:
    static int& push_trie(std::unique_ptr<trie>& t, std::string
    const& w){
      trie* pt = t.get();
      for(char c : w){
```

```
        if(pt->next_char_[c]==nullptr){
          pt->next_char_[c]=std::make_unique<trie>();
        }
        pt=pt->next_char_[c].get();
      }
      return pt->count_;
    }
};
```

Cette classe est composée de deux éléments privés: une unordered_map next_char_ composée de chars/unique_pointeurs sur "trie", et d'un entier count_ qui sera notre compteur d'occurences.
Cette classe comporte également une fonction statique publique mais nous verrons son fonctionnement au moment de l'insertion d'un mot dans notre trie.

Cette fois ci, le trie est utilisé dans la fonction "méthode_2" qui renverra un "unique_ptr<stdtrie>" une fois l'exécution de la fonction achevée. Le principe de fonctionnement de "méthode_2" est fortement identique à celui de la méthodes précédentes. Voici les différences entre ces deux fonctions (qui se retrouvent principalement au niveau de la oÃ^1$nousstockonsnosmots, etlorsdel'insertiond'unmot$) :

Tout d'abord, nous ne declarons pas type "hmap" mais "stdtrie" comme suit:

```
std::unique_ptr<stdtrie> pt = std::make_unique<stdtrie>();
```

Ensuite lors de l'insertion, nous utilisons notre fonction statique publique "push_trie", déclarée dans notre classe, pour inserer un mot dans notre trie, de la même manière que nous avons utilisé l'opérateur crochet pour insérer un élément dans la méthode précédente.

```
if(word.empty() == 0){
    ++stdtrie::push_trie(pt,word);
    word.clear();
}
```

On incrémente le mot en question dans le trie par l'opérateur "++" lors de l'appel de la fonction "push_trie".
Maintenant, jettons un coup d'oeil sur le prototype de la fonction:

```
int& push_trie(std::unique_ptr<trie>&, std::string const&);
```

La fonction est de type "int&", ce qui nous permet d'incrémenter la valeur retournée par la fonction, par le biais d'un simple "++" comme nous l'avons vu précédement. De plus elle prend, par référence, un trie "t", oÃ¹ $l'on va stocker l'autre argument, le mo$

$Afin de pouvoir nous déplacer et modifier notre trie passée en argument, nous créeons un alias à cet trie tel$
```
trie* pt = t.get();
```
Cela nous permetra d'ajouter des éléments dans notre tri sans modifier l'adresse de ce dernier.

Ensuite, nous allons parcourir notre string "w", caractère par caractère ("c" le caractère courant) par le biais d'un boucle range-based for. Nous allons tester si next_char_ de notre caractère courant est déja alloué (next_char_[c]), c'est à dire déjà stocké dans notre trie à cet emplacement donné. Si notre caractère n'est pas stocké à cet endroit précis de notre trie, on va donc créer un nouveau trie à l'emplacement de notre caractère dans la "unordered_map" next_char_.
Une fois ce test fait, avant de reboucler, on passe à l'élément suivant de next_char_[c], autrement dit, on se positionne sur la lettre que l'on vient de traiter (dans la condition), en tant que caractère suivant:

```
for(char c : w){
    if(pt->next_char_[c]==nullptr){
        pt->next_char_[c]=std::make_unique<trie>();
    }
    pt=pt->next_char_[c].get();
}
```

Une fois la boucle achevée, on peut retourner le compteur, que l'on prendra soin d'incrémenter lors de l'appel de la fonction!

Pour finir, avec notre fonction "méthode_2", la dernière différence avec la première méthode est qu'il faut retourner notre unique_ptr<stdtrie> "pt" que l'on a déclaré au début de la fonction à la place de notre notre hmap "table".

Implémentation De Ma Méthode D'Indexation

La troisième méthode est ma méthode d'indexation. Elle est fortement inspirée de la deuxième méthode, celle du trie. Voici comment je l'ai implémenté. Tout d'abord, voici la classe "mtrie" (de type "montrie") utilisée pour se faire:

```
using montrie = class mtrie{

private:
    std::unique_ptr<mtrie> next[26];
    int count_;

public:
  static int& push_montrie(std::unique_ptr<mtrie>& t, std::string const& w){
    mtrie* pt = t.get();
    int index;
    for(char c : w){
      index = c - 97;
      if(pt->next[index]==nullptr){
        pt->next[index]=std::make_unique<mtrie>();
      }
      pt=pt->next[index].get();
    }
    return pt->count_;
  }
};
```

Comme dans la méthode 2 nous retrouvons deux declarations en privé: un compteur count_ pour compter le nombre d'occurences de chaque mot et un array de 26 "unique_ptr<mtrie>". L'utilisation d'un tel tableau est utile du point de vu de la mémoire. On déclare un tableau de 26 pointeurs sur "mtrie", oÃ¹ *onpourrastockeraumaximum*26*adresses*(*pour*26*caractèrespossibles*).*Enstockantdesadresses*
Nousverronsunefoisdeplusla fonctiondéclaréenpublicaumomentdel'insertiond'unmotdansnotren

La fonction "*methode_3*" *à pour prototype* :

```
std::unique_ptr<montrie> methode_3(const vector<std::string>);
```

Cette fontion va ressembler trait pour trait à "methode_2", hormis le type du trie initialisé et retourné (ici "montrie"), et la fonction d'insertion utilisée

pour ajouter un mot à notre trie est différente, nous appeleront "push_montrie" à la place de "push_trie". Ci-dessous, le prototype de cette fonction:

```
int& push_montrie(std::unique_ptr<mtrie>&, std::string const&);
```

Comme son nom le laissait présager, c'est une fonction dérivée de "push_trie". On utilisera "t" notre trie, et "w" le mot à insérer.

La réelle diffèrence entre "push_trie" et "push_montrie", est que "push_trie" utilise "stdtrie" qui possède une unordered_map avec char/int, il faut donc utiliser l'opérateur [ ] avec un caractère pour accéder à notre élément courant, tandis que pour "push_montrie", c'est un tableau, nous accédons donc aux éléments par leurs indexs. Sinon, elles sont identiques (hormis le type du trie):

```
int& push_montrie(std::unique_ptr<mtrie>& t, std::string const& w){
  mtrie* pt = t.get();
  int index;
  for(char c : w){
    index = c - 97;
    if(pt->next[index]==nullptr){
      pt->next[index]=std::make_unique<mtrie>();
    }
    pt=pt->next[index].get();
  }
  return pt->count_;
}
```

Ici, la variable "index" correpond à chaque case du tableau. En effet, si on essaie d'accéder à un élément par son caractère courant en index, cela ne fonctionnera pas car il lira simplement sa table ASCII. Il faut donc soutraire la valeur ASCII du caractère 'a' (97), pour pouvoir caler les valeurs de nos caractères sur celle du tableau.

Difficultés & Problèmes

Lors de ce projet, j'ai été confronté face à un problème, qui a engendré une perte de temps. Avant d'arriver à un résultat concluant pour la troisième

méthode, j'ai implémenté une table de hachage à partir d'un vecteur de structure tel que:

```
struct Cellule{
  std::string word;
  int count;
  int key;
};
typedef Cellule cellule;
```

Pour chaque mot, la fonction hash retournait une clef.

```
int hachage(std::string word){
  int key=0;
  for(auto i=0; word[i] != '\0'; ++i){
    key += word[i] * 500;
  }
  return key %= 20000;
}
```

Pour ajouter une cellule au vecteur, on utilisait la clef du mot à insérer comme index, si cette cellule était déja occupée, on incrémentait le compteur pour gérer les collisions.

```
void push_cell(std::string word, vector<cellule>& v){
  int k = hachage(word);
  for(std::vector<Cellule>::size_type i=k; i<v.size(); ++i){
    if(v[i].word.empty()){
      v[i].key = i;
      v[i].word = word;
      v[i].count = 1;
      return;
    } else if((v[i].word == word) && v[i].key == (int)i){
      ++v[i].count;
      return;
    }
  }
}
```

Mais cette méthode a un gros soucis: le temps d'éxécution. En effet, on est obligé de parcourir des bouts de vecteurs qui peuvent parfois être très long (en cas de collisions successives, nous devrons parcourir notre vecteur tant qu'une cellule n'est pas vide).
De plus, contrairement à la dernière méthode du projet, on stock 3 données dans une struct cellule, qui plus est, sera stockée dans un vecteur, ce qui est vraiment lourd. Je laisse tout de même cette méthode à disposition, mais séparée dans le fichier "methode3old.cpp". Ce fichier est compilé par le makefile, nous pouvons donc la tester dans le fonction "main" du fichier "main.cpp", mais seulement avec 1 ou 2 fichiers maximum, sinon l'attente dépassera les 3 mins...

Conclusion

Pour conclure, voici une trace d'exécution du programme:

```
[etienne@Knuckles Projet]$ ./ProjetAS1
Temps pour la m\'ethode 1: 12521 millisecondes
Temps pour la m\'ethode 2: 25144 millisecondes
Temps pour la m\'ethode 3: 8618 millisecondes
```

Notre troisième méthode utilisée est la plus performante parmis les trois (voir quatre) implémentées dans le projet.

Ce projet m'a permis de me rendre compte de l'importance qu'a un programme à être optimisé. Plus il y a de données à traiter, plus cela mettra de temps. Il est donc important de chercher l'optimisation pour, un flux massif d'information à traiter, ou encore un programme gourmant en mémoire.

Mentions

Sites qui m'ont permis de mieux comprendre le sujet:

https://en.cppreference.com/w/ (Particulièrement pour unordered_map)

https://fr.wikipedia.org/wiki/Table_de_hachage

https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c/19978-les-tables-de-hachage

Personnes qui m'ont permis de mieux comprendre le sujet sur les débuts:

Mr. Jean-NoÃl VITTAUT (Professeur de la matière AS1)

Camarades de TD