Devoir de Programmation : Tries ALGAV

Shiyao CHEN 28707756
Jean-Charles KAING 3808150

Sommaire

1.1 Structure 1 : Patricia-Tries

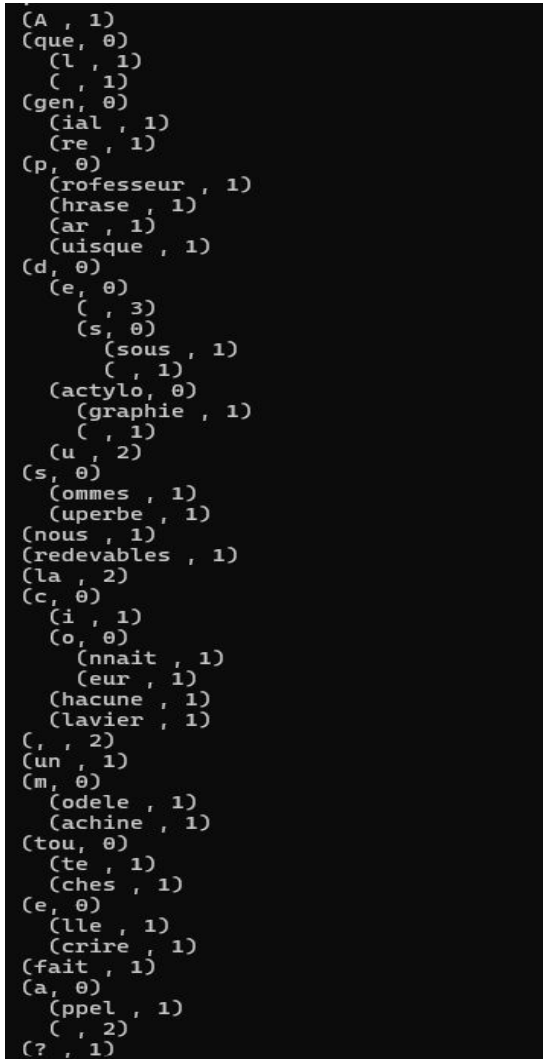
Dans “src/Patricia-Trie/patricia_trie.h” :

```
#define END_OF_WORD "\x20"    // terminateur de mot d'un PAT


typedef struct PAT PAT;

typedef struct _Node{
        char* cle; // le plus long prefixe commun
        int valeur; //nb de cle;
        PAT* fils;
} Node;

typedef struct PAT{
        Node** node;
} PAT;
```

```
/* Fonction pour créer un PAT vide */
PAT* PATVide();

/* renvoie 1 ssi l'arbre PAT est vide, 0 sinon */
int EstVide(PAT* A);

/* Fonction pour créer un arbre avec un seul nœud */
PAT* PATCons(Node* n);

/* renvoie la clé du noeud*/
char* Rac(Node* A);

/* renvoie le nb associe la clé du noeud */
int Val(Node* A)

/* Fonction pour ajouter un enfant à un nœud */
void ajouter_fils(Node* A, Node* fil );

/*Fonction pour ajouter une racine*/
void ajouter_racine(PAT** P, Node* r);

/*Fonction pour insérer un mot dans l'arbre*/
void PATinsertion(PAT** A, char* m) ;

/*Foction pour verifier c est préfixe de m, si oui renvoie 1, sinon 0*/
int estPrefixe (char*c , char* m);

/*Foction pour renvoyer la longeur de préfixe commun de c et m*/
int prefixe(char* c, char* m);
```

L’arbre PAT avec l’exemple

A quel genial professeur de dactylographie sommes nous redevables de la superbe phrase ci dessous, un modele du genre, que toute dactylo connait par coeur puisque elle fait appel a chacune des touches du clavier de la machine a ecrire ?



2 Fonctions avancées

```
/* recherche un mot dans PAT, si le mot présent renvoie 1, sinon 0*/
bool recherchePAT(PAT* A, char* mot);  // O(L*k)

/* compte les mots présents dans le PAT*/
int ComptageMotsdansPAT(PAT* A);  // O(n)

/* liste les mots du PAT dans l'ordre alphabétique */
char** ListeMotsdansPAT(PAT* A); // O( n · Lmax)

/* compte les pointeurs vers Nil */
int ComptageNildansPAT(PAT* A); // O(n)

/* calcule la hauteur de le PAT */
int HauteurPAT(PAT* A);  //O(n)

/* calcule la profondeur moyenne des feuilles de PAT */
int ProfondeurMoyennePAT(PAT* A); // O(n)

/*compter les mots du dictionnaire tel que m est préfixe*/
int PrefixedansPAT(PAT* A, char* m);  // O(L* k)

/* supprime le mot dans PAT */
void PATsuppression(PAT** A, char* mot); // O(L*k)
```

Recher un mot dans PAT:
le mot 'des' est dans l'arbre ? = 1

Les mots présents dans le dictionnaire:
il y a 37 mots présents dans le dictionnaire (avec les ponctuations).

liste les mots du dictionnaire dans l'ordre alphabétique:
Mots dans le Patricia Trie :
[, ? A a appel chacune ci clavier coeur connait dactylo dactylographie de des dessous du ecrire elle fait genial genre la machine modele nous par phrase professeur puisque que quel redevables sommes superbe touches toute un]

Compte les pointeurs vers Nil:
Nombre total de pointeurs NULL : 52

Calcule la hauteur de l'arbre PAT:
Hauteur du Patricia Trie : 4

Calcule la profondeur moyenne des feuilles de l'arbre PAT:
somme_profondeur: 75,nb_feuille: 37
Profondeur moyenne des feuilles du Patricia Trie : 2

Compter de mots du dictionnaire le mot A est préfixe.
Il y a 2 de mots du dictionnaire le mot 'que' est préfixe.

1.2 Structure 2 : Tries Hybrides

Dans “src/tries_hybrides/tries_hybrides.h” :

```
typedef struct TrieH {
   char l; // lettre
   int v;  // valeur
   struct TrieH *inf; // sous-arbre gauche
   struct TrieH *eq;  // sous-arbre central
   struct TrieH *sup; // sous-arbre droite
   int hauteur; // hauteur (pour la partie 6)
} TrieH;
```

```
/* met à jour la hauteur du noeud (Q 3.8) */
void majHauteur(TrieH* A);

/* renvoie le trie hybride resultant de l'insertion de c dans A */
TrieH* TH_Ajout(char* c, TrieH* A, int v);
```

```
/* construit une trie hybride */
TrieH* TrieHybride(char l, TrieH* inf, TrieH* eq, TrieH* sup, int v);

/* renvoie une trie hybride vide */
TrieH* TH_Vide();

/* renvoie 1 ssi le trie hybride A est vide, 0 sinon */
int EstVide(TrieH* A);

/* renvoie le caractère de la racine du trie hybride */
char Rac(TrieH* A);

/* renvoie l'entier de la racine du trie hybride, -1 sinon */
int Val(TrieH* A);

/* renvoie une copie du sous-arbre gauche de A */
TrieH* Inf(TrieH* A);

/* renvoie une copie du sous-arbre central de A */
TrieH* Eq(TrieH* A);

/* renvoie une copie du sous-arbre droite de A */
TrieH* Sup(TrieH* A);
```

```
/* renvoie la première lettre de la chaine de caractères */
char prem(char* c);

/* renvoie le reste de la chaine de caractères
   privée de la première lettre */
char* reste(char* c);
```

2 Fonctions avancées Tries Hybrides

Dans “src/tries_hybrides/fonctions_avancees.h” :

```
/* recherche un mot dans un dictionnaire
 * renvoie 1 ssi le mot est présent dans le dictionnaire,
 *         0 sinon
 */
int Recherche(TrieH* arbre, char* mot);

/* compte les mots présents dans le dictionnaire */
int ComptageMots(TrieH* arbre);
```

```
typedef struct List {
    char* mot;
    int entier;
    struct List *suiv;
} List;

/* liste les mots du dictionnaire dans l'ordre alphabétique
 * renvoie une liste de mots
 */
List* ListeMots(TrieH* arbre);

/* compte les pointeurs vers Nil */
int ComptageNil(TrieH* arbre);

/* calcule la hauteur de l'arbre */
int Hauteur(TrieH* arbre);

/* calcule la profondeur moyenne des feuilles de l'arbre */
int ProfondeurMoyenne(TrieH* arbre);

/* prend un mot A en arguments
 * et qui indique combien de mots du dictionnaire
 * le mot A est préfixe
 */
int Prefixe(TrieH* arbre, char* mot);

/* prend un mot en argument
 * et qui le supprime de l'arbre s'il y figure
 */
TrieH* Suppression(TrieH* arbre, char* mot);
```

3 Fonctions complexes

$O((n1+n2) \cdot L\_max * k)$

Algorithm 1 PATfusion

Input: Deux arbres PAT : A et B
Output: Arbre PAT fusionné A

```
1  if A est vide then
2  |  return B
3  end
4  else if B est vide then
5  |  return A
6  end
7  chaque nœud bNode dans B cle_b ← clé de bNode
8  if cle_b n'existe pas dans A then
9  |  Ajouter bNode comme racine dans A
10 end
11 else
12 |  chaque nœud aNode dans A cle_a ← clé de aNode
13 |  if cle_a commence par cle_b (ou vice versa) then
14 |  |  if cle_a et cle_b sont identiques then
15 |  |  |  if les deux clés sont des feuilles then
16 |  |  |  |  Additionner les valeurs de aNode et bNode
17 |  |  |  end
18 |  |  |  else
19 |  |  |  |  Fusionner leurs sous-arbres
20 |  |  |  end
21 |  |  end
22 |  |  else
23 |  |  |  len_com ← longueur du préfixe commun entre cle_a et cle_b
24 |  |  |  pref_com ← préfixe commun
25 |  |  |  a_rest ← partie restante de cle_a
26 |  |  |  b_rest ← partie restante de cle_b
27 |  |  |  F ← nouveau nœud avec clé a_rest et sous-arbres de aNode
28 |  |  |  G ← nouveau nœud avec clé b_rest et sous-arbres de bNode
29 |  |  |  Créer un sous-arbre fusionné entre F et G
30 |  |  |  Mettre à jour aNode avec clé pref_com, valeur 0, et sous-arbre fu-
   |  |  |   sionné
31 |  |  end
32 |  end
33 end
34 return A
```

5 Format de rendu du code

```
/* Ecrit le patricia dans le fichier format JSON  */
int ecrire_patricia(char* namefile, PAT* arbre);

/*Convertir le PAT dans le format JSON*/
cJSON* node_to_json(Node* node);
cJSON* pat_to_json(PAT* pat);

/* Construit le patricia depuis le format JSON */
PAT* json_to_pat(cJSON* json_node);
cJSON* node_to_json(Node* node);
```

Pour le format de rendu du code, on a écrit des scripts bash qui appellent des scripts en C.

Pour gérer les fichiers JSON pour les Patricia-tries, on a utilisé la bibliothèque cJSON.h.

Pour gérer les fichiers JSON avec les tries hybrides, on a écrit les fonctions suivantes dans “src/tries_hybrides/ecriture_lecture.h” :

```
/* Ecrit le trie hybride dans le format JSON */
int ecrire_trie(FILE* file, TrieH* arbre, int tabulation);

/* Modifie la valeur de cpt */
void setCpt(int valeur);

/* Construit le trie hybride depuis le format JSON */
TrieH* charger_trie(char *content, int *index);
```

Un exemple de script bash de listeMots en C

```
#!/bin/bash

# Check if exactly two arguments are provided
if [ "$#" -ne 2 ]; then
        echo "Usage: $0 <x> <y>"
        exit 1
fi

x=$1
y=$2

# Check if x is either 0 or 1
if [ "$x" -eq 0 ]; then
        echo "Running liste_mots_patricia.c with argument $y"
        ./liste_mots_patricia "$y"
elif [ "$x" -eq 1 ]; then
        echo "Running liste_mots_trie.c with argument $y"
        ./liste_mots_trie "$y"
else
        echo "Error: x must be 0 or 1"
        exit 1
fi
```

Tester:

En patricia_trie
./listeMots.sh 0 pat.json

En trie_hybride
./listeMots.sh 1 trie.json

→ retourne un fichier mot.txt

6 Etudes expérimentales

```
typedef struct words{
        char* data;
        struct words* suiv;
}Words;
```

```
/*Lit les fichiers contenus dans un dossier spécifié et extrait les mots.*/
Words* read_Files_Shakespeare(char* nomDossier);

/*Écrit les mots d'une structure Words dans un fichier*/
void eciture_words(Words* words);

/* Ouvre un fichier spécifié et extrait les mots contenus dans celui-ci.*/
Words* read_ouvre_Shakespeare(char* nomFichier);

/*Mesure le temps d'exécution pour l'ajout d'une seule clé dans une structure PAT.*/
double measureTime_ajout_un_seul_PAT(void (*function)(PAT**, char*),PAT** pat, char* cle);

/*Mesure le temps d'exécution pour la suppression d'une clé dans une structure PAT.*/
double measureTime_supp_PAT(void (*function)(PAT**, char*),PAT** pat, char* cle);

/*Mesure le temps d'exécution pour la fusion de deux structures PAT.*/
double measureTime_fusion_PAT(PAT* (*function)(PAT*, PAT*),PAT* a, PAT* b);
```

Resultat

// DANS PAT

temps de construction PAT est :0.3994350

temps d'ajout d'un nouveau mot dans PAT: 0.0000030

somme_profondeur: 130337,nb_feuille: 23087

Hauteur de PAT est 11, Profondeur de PAT est 5

Temps de la suppression d'un ensemble de mots dans PAT: 0.1040390

temps de fusion les PAT est : 0.0802520

Pour un petit nombre de clés, l'insertion est plus rapide ;

Pour des données à grande échelle et de nombreux préfixes partagés, la fusion peut être plus rapide.

// dansTrieH, exécutable ./main_etude_trie

temps de construction TrieH équilibré est : 0.0596500

temps d'ajout d'un nouveau mot dans TrieH equilibré est 0.000000

Hauteur de TrieH équilibré est 1980392558, Profondeur de TrieH équilibré est 17

Temps de la suppression d'un ensemble de mots dans TrieH équilibré: 0.0875000

temps de construction TrieH non équilibré est : 0.0649510

temps d'ajout d'un nouveau mot dans TrieH non equilibré est 0.000080

Hauteur de TrieH équilibré est 1986289939, Profondeur de TrieH non équilibré est 19

Temps de la suppression d'un ensemble de mots dans TrieH non équilibré: 0.0904770

Conclusion

En conclusion , ce projet a permis de réaliser une comparaison complète des deux structures de tries et d'étudier leur performance.

Merci pour Votre attention ! !