Rapport projet RN40

Algorithme de tri sur listes chaînées

Gabriel SCHWAB

Automne 2021

Contents

1 Outils et Compilation

Pour réaliser ce projet j'ai utilisé l'outil de versionning **Git**. Cet outil m'a permis de pouvoir gérer les différentes versions de mon projet au fur et à mesure que celui-ci a évolué.

J'ai aussi utilisé **Valgrind** qui est un outil open-source qui m'a permis de m'assurer que mon programme ne comportait pas de fuites mémoires.

Pour compiler ce projet j'ai choisis d'utiliser **CMake**. L'avantage de CMake par rapport à d'autres scripts de compilation comme **makefile** est qu'il permet de générer un script de compilation quelque soit la plateforme utilisée (Windows, Linux, ...). C'est grâce au fichier CMakeLists.txt que le script de compilation va être généré. Pour utiliser CMake il faut :

- CMake
- GNU Make ou d'autres outils comme Ninja ou MinGW Make
- Un compilateur C, j'ai utilisé MinGW 64

Pour compiler le project :

Il est préférable de créer un dossier `build` dans le dossier du projet, voici comment procéder :

```
mkdir build && cd build
cmake .. -G <generateur>
make
```

J'utilise personnellement "MinGW Makefiles" comme générateur. Ensuite il suffit d'executer le programme avec les nombres que l'on veut trier passés en paramètre. Par exemple : `rn40-project 453 22 17 1 321`

2 Type Bucket

Tout d'abord, dans ce projet nous manipulons notre propre type abstrait de données, le type Bucket en référence au type "seaux" du sujet. Cependant, il n'y a quasiment aucune différence avec les liste chaînées vues en cours. Les mots seaux et liste seront donc équivalents dans la suite du rapport. La liste de Bucket est codée sous forme d'un tableau de liste chaînée.

2.1 Définition

Le type $Bucket$ est un type abstrait équivalent à la structure de donnée liste vue en cours. On note B l'ensemble des $Bucket$. On note $empty\_bucket$ le $Bucket$ vide.
Le type $Bucket$ contient des éléments (nombres) de type T, qui dans notre implémentation en C est un chaîne de caractère.
On a donc le type abstrait $Bucket(T)$ qui est définit par les fonctions suivantes :

$create : () \longrightarrow B$, constructeur qui retourne $empty\_bucket$.

$head : B \longrightarrow T$, associe au Bucket son premier élément ou $empty\_bucket$ s'il y en a pas.

$rest : B \longrightarrow B$, retourne l'élément suivant du Bucket en paramèter ou $empty\_bucket$ s'il n'y en a pas.

cons : $B \times T \longrightarrow B$, modifie le Bucket en lui attachant en tant qu'élément suivant l'élément en paramètre .

2.2 Axiomes

Soit B l'ensemble des Bucket et T l'ensemble des éléments que chaque Bucket peut contenir.

$\forall\, b \in B, \forall\, t \in T$:

(i) $head(cons(t, b)) = t$

(ii) $rest(cons(t, b)) = b$

(iii) $cons(t1, b1) = cons(t2, b2) \iff t1 = t2 \; et \; b1 = b2$

2.3 Opérations

2.3.1 is_empty

$is\_empty : B \longrightarrow Booléen$, Vrai si Bucket est vide, Faux sinon.

Algorithm 1: is_empty

Lexique:
- B : Bucket à tester
- R : Vrai si Bucket est vide, Faux sinon

Donées: B : Bucket
Résultat: R : Booléen
si $B = empty\_bucket$ **alors**
  R $\leftarrow Vrai$
sinon
  R $\leftarrow Faux$
fin

2.3.2 insert_tail

$insert\_tail : B \times T \longrightarrow B$, insère un élément en queue du Bucket.

Algorithm 2: insert_tail

Lexique:
- B : Bucket à modifier
- B' : Bucket modifié
- X : Nombre à ajouter

Donées: B : Bucket, X : T
Résultat: B' : Bucket
B' $\leftarrow B$
si *is_empty(B')* **alors**
  B' $\leftarrow create()$
sinon
  Tant que *non(is_empty(rest(B')))* **faire**
    B' $\leftarrow rest(B')$
  fintq
  B' $\leftarrow cons(B', X)$
fin

2.3.3 remove_head

$remove\_head : B \longrightarrow B$, retire l'élément en tête du Bucket.

Algorithm 3: remove_head

Lexique:
- B : Bucket à modifier
- B' : Bucket modifié

Donées: B : Bucket
Résultat: B' : Bucket
B' $\leftarrow B$
si *non(is_empty(B'))* **alors**
| B' $\leftarrow rest(B')$
fin

2.3.4 delete

$delete : B \longrightarrow B$, supprime le Bucket recursivement et retourne un Bucket vide.

Algorithm 4: delete

Lexique: B : Bucket à supprimer
Donées: B : Bucket
Résultat: *empty_bucket* : Bucket
`DELETE(`B`)`
si *non(is_empty(B))* **alors**
| delete(rest(B))
| B ←emty_bucket
fin

2.3.5 print_bucket

print_bucket, affiche le Bucket.

Algorithm 5: print_bucket

Lexique:
-B : Bucket à imprimer
- B' : Bucket temporaire pour parcourir B
Donées: B : Bucket
Résultat:
si *is_empty(B)* **alors**
  print(*** empty bucket ***)
sinon
  B' $\leftarrow B$
  Tant que *non(is_empty(B'))* **faire**
    print(head(B'))
    B' $\leftarrow rest(B')$)
  fintq
fin

2.3.6 print_bucket_list

Algorithm 6: print_bucket_list

Lexique:
- LB : Liste de bucket
- size : taille de LB
- i : entier qui sert de compteur
Donées: LB : Liste(Bucket), size : entier
Résultat:
Pour *i de 0 à size-1* **faire**
  print_bucket(LB[i])
finpour

2.3.7 insert_bucket_list

insert_bucket_list : permet d'insérer un nombre (élément de type T) dans la liste de Bucket.

3 Algorithme de Tri

3.1 Algorithme

Algorithm 7: Sort

Lexique:
- N : la base numérique
- Max : nombre maximum de chiffre parmis les nombres du Bucket
- B : Bucket à trier
- B' : Bucker trié
- LB : Liste de Bucket

Donées: N : Entier, Max : Entier, B : Bucket
Résultat: B' : Bucket
B' $\leftarrow B$
Pour *i de 0 à Max-1* **faire**
  Tant que $non(is\_empty(B'))$ **faire**
    j $\leftarrow \frac{head(L')}{10_N^i} \pmod{10_N}$
    LB[j] $\leftarrow insert_tail(LB[j], head(B'))$
    B' $\leftarrow remove\_head(B')$
  fintq
  Pour *k de 0 à N-1* **faire**
    Tant que *non(is_empty(LB[k])* **faire**
      B' $\leftarrow insert\_tail(B', head(LB[k])$
      LB[k] $\leftarrow remove\_head(LB[k])$
    fintq
  finpour
finpour

3.2 Explications

Voici la première version de l'algorithme de tri décrite dans le sujet. Cet algorithme est valable dans n'importe quelle base numérique. En effet $\frac{head(B')}{10_N^i}$ troncature le premier nombre de la liste B' au $i^{ème}$ chiffre. Ainsi $\frac{valeur(B')}{10_N^i} \pmod{10_N}$ renvoie le $i^{ème}$ chiffre du premier nombre de B'. On remarque que 10_N est le nombre 10 dans la base numérique choisis. En effet 10 en base 2 est égal à 2 en base 10, 10 en base 8 est égal à 8 en base 10 ...
Ensuite on se sert de la valeur de ce $i^{ème}$ chiffre comme indice pour l'insérer **en queue** dans le seau correspondant dans la liste de seaux.
Enfin on retire l'élément de la liste à trier avec $remove\_head()$ ce qui a aussi pour effet de décaler le $2^{ème}$ élément de la liste à la $1^{ère}$ place donc pas besoin de faire $B' \leftarrow rest(B')$.
Pour finir, on parcourt la liste de seaux et on retire chaque nombre pour l'ajouter **en queue** de B'.

Un détail qui est essentiel pour que cet algorithme de tri fonctionne correctemennt est le fait qu'on ajoute **en queue** à chaque fois qu'on place un nombre dans un seaux ou dans la liste à triée. Cela rend la méthode de tri **stable**.

Nous avons choisis d'utiliser un algorithme itératif pour implémenter cette méthode de trie car cela est plus simple pour trier les nombre en partant du chiffre le plus à droite. Si nous devions trier les nombres par la gauche, un algorithme récursif aurait pu être utilisé facilement.

3.3 Analyse de complexité

Soit une liste contenant n nombres, ayant chacun au plus k chiffres pouvant prendre dans une base b (pouvant prendre donc b valeurs). Le coût pour remplir la liste de seaux est d'au plus n opérations. Le coût pour vider la liste de seaux et remplir la liste à trier est d'au plus n+b opérations (on parcours obligatoirement b seaux et n nombres). On repète cela k fois ce qui donne un complexité temporelle $O(k(n+b))$.
Comme dans notre cas, d est constant et k = $O(n)$, on en déduit que la complexité de notre algorithme de tri est $O(n)$.

Cet algorithme de tri à donc une complexité en temps linéaire ce qui est mieux que la plupart des algorithme de tri par comparaison tel que merge sort ou quicksort qui ont une complexité temporelle quasi linéaire $O(log(n)n)$. Un désavantage pourrai être sa complexité spatiale lorsqu'on travaille avec des bases numériques très grande.

3.4 Algorithme de Tri adapté

Pour des raisons de programmation, nous avons apporté quelques modifications à cet algorithme afin de pouvoir l'implémenter en langage C. Comme notre algorithme de tri fonctionne pour une base numérique quelconque, il faut que nous puissions le programmer de cette façon. Ainsi les nombres sont codés sous forme de chaînes de caractère ce qui s'apparente à une liste d'un point de vu algorithmique. Ainsi nous sommes obligés d'appporter quelques modifcations à notre algorithme de départ, notamment une fonction qui ajoute un nombre au bon index de la liste de seaux.

Cet alorithme est maintenant même valable pour des chaînes de caractère quelconques et ainsi, il peut donc aussi trier des mots par ordre alphabétique par exemple.

Algorithm 8: Sort

Lexique:
- N : la base numérique
- Max : nombre maximum de chiffre parmis les nombres de lu Bucket
- B : Bucket à trier
- B' : Bucker trié
- LB : Liste de Bucket

Donées: N : Entier, Max : Entier, B : Bucket
Résultat: B' : Bucket
B' $\leftarrow B$
Pour *i de Max-1 à 0* **faire**
  Tant que $non(is\_empty(B'))$ **faire**
    insert_bucket_list(LB, head(B'), head(B')[i])
    B' $\leftarrow remove\_head(B')$
  fintq
  Pour *k de 0 à N-1* **faire**
    Tant que *non(is_empty(LB[k])* **faire**
      B' $\leftarrow insert\_head(B', head(LB[k])$
      LB[k] $\leftarrow remove\_head(LB[k])$
    fintq
  finpour
finpour