Rapport : SDA City Search

Membre du groupe

Nom Prénom	Numéro Etudiant	Pseudonyme
Bayard Emeric	11606611	Dryska
Sheikh Rakib	11502605	NoobZik
Yesli Rayane	11507199	soso7
Belmaati Yacine	11513398	Laser1W

Analyse théorique et empirique

1) Choix d'implémentation

Le code général est implémenté avec le principe de la programmation défensive (d'où la présence des fonctions assert dans les algos et BST). Afin de limiter les problèmes d'accès au mémoire non autorisé et en cas de non-respect de l'écriture des coordonnées par l'utilisateur.

- **BinarySearchTree.c**

La manière dont les arbres de binaire de recherche sont écrites dans le fichier Header n'était pas familier avec ce qu'on a vu en cours. C'est-à-dire une structure contenant un pointeur vers la racine et un deuxième définissant les nœuds.

Du coup, l'implémentation certaines fonctions du BST est spécialement basé sur les cours de l'Institut Montefiore rédigé par Pierre Guerts, en complément de nos propres cours.

Le Header nous informe que nous devons utiliser une structure opaque pour les ABR. En voyant par la suite les prototypes des fonctions, nous avons jugé que c'était logique de mettre tous les champs dans une même structure pour pouvoir gérer efficacement les fonctions récursives.

On a eu des problèmes à faire fonctionner insertInBST. On est parti sur la base du TP effectué en cours. Le programme nous disait que le bst était vide. On a remarqué que le bst créée pour l'ajout remplace le bst passé en paramètre. On a dû régler ce problème en ajoutant un pointeur vers le nœud qui le précède et en créant un *tmp* pour gérer l'ajout.

La fonction getInRange était particulière, on a dû le recoder plusieurs fois. On est tout d'abord partie sur une récursion. On a remarqué assez rapidement que la fonction prenait seulement une liste chaînée, clé minimal et maximum et pas de bst.

On a pensé à faire une nouvelle fonction récursive sur ce bst pour un parcours latéral. Mais les return posent problèmes. On avait un truc du genre :

```
const BinarySearchTree *inorderBST(BinarySearchTree *bst) {
  if (!bst) return NULL;
  return inorderBST(bst->left);
  return bst;
  return inorderBST(bst->right);
}
```

Chose qui est impossible.

Du coup, on est parti sur une version itérative avec un parcours latérale. On pense que c'était la seule solution disponible. On a dû rajouter une fonction supplémentaire pour gérer les extractions d'une liste sous forme de file temporairement.

- **intersect.c**

On s'est entièrement basé sur les paramètres du prototype donnée en fichier d'entête pour rédiger cette fonction. La première approche est totalement naïve. On prend 1 élément de la liste A et on regarde s'il est dans la liste B. On admet alors que la complexité de cet algorithme est très médiocre. θ(4*N) + θ(P*Q)

Après avoir testé sur des bases de données plus grandes, on a très vite remarqué que cette première approche était très lente.

On a alors réfléchi sur une deuxième approche qui est de faire un Tri-fusion de la liste A et B (qui ne coûte que θ(n*log(n)) pour ensuite faire son intersection.

Ce qui ramène a faire :

- getInRange de Liste A
- getInRange de Liste B
- Tri fusion de Liste A et B
- calcule de l'intersection de deux listes triées.

La complexité estimée sera alors de θ(2(p+q) + θ((p*log(p) + q*log(q)) + θ(p+q) (Largement mieux et plus rapide).

On avait eu des problèmes de Stack Overflow sur la récursion du SortedMerge. Il est sous investigation sur ce lien (StackOverflow).

Du coup, on est passé en version itérative (qui n'a pas de pertes sur la complexité générale) le temps qu'on comprend comment c'est possible. Le code de la récursion est laissé en commentaire.

- **findCities (1 / 2 / Z) BST.c**

La manière dont les algorithmes 1 / 2 / Z sont entièrement basés sur l'algorithme par liste chaînée afin de garder une forme assez similaire.

Chacun des trois algorithmes ont la même boucle while pour l'insertion. Cependant, le deuxième algorithme contient une deuxième boucle de l'insertion vu que c'est sur deux arbres binaires de recherche.

2) Pseudo-code de getInRange

```
# Prends en paramètre : BinarySearchTree *, void *keyMin, void *keyMax
# 2 LinkedList (LL and File)
# LL for returning a filtered city
# file for Level Order Transversal BinarySearchTree

WHILE temp exist
  if minimum < temp->key AND temp->key < maximum
    if insertInLinkedList(LL, temp->value) == false;
      ret NULL;
    if temp->left exist
      if insertInLinkedList(File, temp->left) == false;
        ret NULL
    if temp->right exist
      if insertInLinkedList(File, temp->right) == false;
        ret NULL;
  temp = extractFile(File);
freeLinkedList (File);
```

3) Analyse de complexité

- insertInLinkedList à une complexité de *θ(1)*, on l'ajoute directement à la fin de la liste
- extractFile à une complexité de *θ(1)* aussi, elle retire le premier élément de la liste.
- Les instructions if sont de complexité *θ(1)*.
- **En répétant N fois cette boucle, on a donc une complexité de *θ(n)***.
- Dans le meilleur cas : *θ(1)*.
- Dans le pire cas : *θ(n)*

4) Pseudo-code de intersect

- Première approche :

```
tmpA = listA->head;
tmpB = listB->head;
while tmpA exist
  while tmpB exist
    if tmpA->value == tmpB->value
      if insertInLinkedList(listC, tmpA->value) == false;
        ret NULL;
    tmpB = tmpB->next
  tmpA = tmpA->next;
  tmpB = listB->head;
ret listC;
```

- Deuxième approche :

```
/*intersect.c*/
LinkedList *intersect (LinkedList *listA,
                       LinkedList *listB,
                       int cmp_foo(const void*, const void*))
    MergeSort(listA);
    MergeSort(listB);
    listC = intersect_list(listA, listB);
    ret ListC
```

5) Analyse de complexité

- **Pour la première approche**

Dans le pire cas, soit M = N, on aura une complexité de $\theta(N^2)$ On peut dire que la manière dont l'algorithme est écrit est très lente.

Dans le meilleur cas, on aura θ(1).

- **Pour la deuxième approche**

Description de l'avancé du tri-fusion

On (Rakib et Florian) avait un très gros problème de complexité en temps sur l'intersect, ce qui est logique avec une approche naïve. On a alors pensé à une autre méthode de l'intersection qui doit être largement plus rapide que n^2.

Pour ma part, j'avais deux sous-approche :

- Approche tout en récursive.
- Approche partiellement en récursive (La fusion + comparaison est en itératif).

L'approche récursive est la plus simple visuellement. Mais d'après la documentation sur les différents problèmes d'une liste chaînée de l'université de Stanford, l'utilisation de la méthode récursive est déconseillée en production de code. En effet, la profondeur de la récursion est proportionnel à la taille des listes. L'espace utilisé par cette récursion est le Stack. On peut très facilement atteindre la limite du stack alloué par le système et provoquer dans le pire cas, une erreur de segmentation.

Après avoir testé cette implémentation, la récursion sur une base de 1 000 000 de ville a échoué (Erreur de segmentation par suite d'un dépassement de mémoire alloué du stack). Mais elle fonctionne correctement sur des bases de ville plus petite.

L'approche partiellement récursive est de créer un pointeur temporaire qui va permettre de déplacer des nœuds d'une liste à l'autre le tout de manière itératif par boucle while. La division de la liste reste quant à elle récursif. On garde un pointeur vers le dernier nœud de la liste trié. C'est au moment de la comparaison que la fonction fusion décide de changer l'emplacement du nœud a ou le nœud b. L'avantage de cette approche est qu'on économise de l'espace occupé de la mémoire du stack.

Dans ces deux sous cas, on devrait garder une complexité égale, donc le choix de l'implémentation ne sera pas un facteur majeur affectant le temps.

Filtrer une liste chaînée par l'algorithme de tri fusion devrait prendre θ(n log(n)). Etant donné que les deux listes chaînées seront linéaires. On fait une comparaison linéaire de ces deux listes. Ce qui nous donne une complexité de θ(n+m) pour cette comparaison linéaire.

La complexité finale sera alors estimée à θ(nlog(n) + mlog(m)). Ce qui devrait être largement plus rapide que la première approche.

Le meilleur cas devrait être θ(1).

6) Comparaison des 3 approches

Pour calculer le temps on décide de le faire sur 1 000 villes et 1 000 000 villes avec les commandes suivantes :

Testé sur un i5 4200H Dual Core 2.8 GHz / 3.4 GHz Turbo.

```
$ time ./boxsearch cities_1000.csv -1 -1 1 1
$ time ./boxsearch cities_10000.csv -1 -1 1 1
$ time ./boxsearch cities_100000.csv -1 -1 1 1
$ time ./boxsearch cities_1000000.csv -1 -1 1 1
```

Pour latitude (-1;1) et longitude (-180;180) pour les villes suivantes

	1 000	10 000	100 000	1 000 000
Algorithme 1	0.004s	0.17s	0.166s	4,067s
Algorithme 2	0.003s	0.018s	0.268s	7,78s
Algorithme 3	0.002s	0.017s	0.135s	2,05s

Pour latitude (-90;90) et longitude (-180;180) pour les villes suivantes

	1 000	10 000	100 000	1 000 000
Algorithme 1	0.002s	0.019s	0.187s	4,078s
Algorithme 2	0.008s	0.348s	2m44,94s	Supérieur à 30min*
Algorithme 3	0.002s	0.016s	0.152s	2,238s

> *La recherche a été abandonnée par* `CTRL+C` *suite à la lenteur de l'algorithme.*

A partir de ce tableau comparatif entre ces trois algorithmes, il est évident que le deuxième algorithme prend le plus de temps à faire la recherche des villes. Ceci est clairement dû à l'implémentation médiocre au terme de complexité de la fonction intersection. On rappelle qu'il y a deux boucles whiles. On prend un élément et on parcourt toute la liste si elle y est. Sinon on passe au suivant. La complexité est alors de θ(4xg) + θ(N*M) ce qui est très lent.

En ce qui concerne l'algorithme 1 et 3, on peut affirmer que l'algorithme 3 est environ 2x plus rapide que le premier algorithme, bien qu'on ait une complexité supplémentaire de θ(N) pour refiltrer les villes qui ont échappé au filtre.

Nouveau tableau comparatif avec tri-fusion

Pour latitude (-1;1) et longitude (-180;180) pour les villes suivantes Pour l'algorithme 3, une optimisation de l'espace a été effectuée, les résultats peuvent différer.

	1 000	10 000	100 000	1 000 000
Algorithme 1	0.002s	0.022s	0.166s	4,067s
Algorithme 2	0.002s	0.023s	0.346s	7,63s
Algorithme 3	0.002s	0.017s	0.135s	2,05s

Pour latitude (-90;90) et longitude (-180;180) pour les villes suivantes

	1 000	10 000	100 000	1 000 000
Algorithme 1	0.002s	0.010s	0.187s	4,078s
Algorithme 2	0.002s	0.028s	0,412s	8,48s
Algorithme 3	0.002s	0.016s	0.152s	2,238s

Avec la recherche en tri-fusion, on a largement diminué la complexité en temps dans le pire cas. Mais on remarque aussi que le temps passé est équivalent à la première approche. On pourrait penser que le fait d'utiliser deux arbres binaires nuit de manière significatif la complexité en temps mais aussi en espace (ou bien que ce soit une mauvaise implémentation, ça reste à voir).

Mais la durée de l'exécution est toujours trois à quatre fois supérieures par rapport à une liste chaînée, 1 bst et du code Morton.

En tout cas ce qui est sûr, l'algorithme sur le code Morton est deux à trois fois plus rapide que ce soit dans la moyenne ou le pire des cas de complexité.