Université de Liège
Faculté des Sciences Appliquées

Rapport projet N°2

Dans le cadre du cours de structures des données et algorithmes

Année Académique 2016-2017

Moitroux Olivier
s152122

Étudiant en 2e Bac Ingénieur Civil
Le 21 avril 2017

1 Introduction

L'objectif de ce projet est d'écrire un algorithme permettant de retrouver toutes les villes comprises dans une région rectangulaire délimitée par quatre couples de points ayant une latitude et une longitude attribuée. Dans ce rapport, les fonction seront représentées de la sorte : `fct()`, indépendamment de leurs arguments et type de retour.

2 Analyse théorique et empirique

2.1 Choix d'implémentation

Par soucis de clarté, nous commencerons par `BinarySearchTree.c`. Cette fonction implémente différente opérations applicables à des arbres binaires de recherche. La première étape consiste à déclarer une structure de type *arbre* contenant des pointeurs vers le *nombre d'éléments* contenus dans l'arbre, le *nœud* racine et vers l'adresse d'une fonction de comparaison générique. Il existe également une structure *nœud* contenant des pointeurs vers une clef, une *valeur* à stocker, le fils *gauche*, le fils *droit* et vers le *parent* du nœud. Afin que le code soit le plus versatile possible, les clefs et valeurs seront de type *const void**. La combinaison de ces deux structures permet de représenter de manière assez générale un arbre.

Afin de créer un arbre, il faut utiliser `newBST()` afin de réserver de l'espace mémoire. Le nœud racine est également créé mais ne contient pas d'information utile. C'est à l'utilisateur de fournir cette information grâce à `insertInBST()`.

Naturellement, une fonction `freeBST()` est nécessaire. Elle appelle une autre fonction récursive(qui peut traiter des nœuds contrairement à la fonction appelante) qui va se charger de libérer l'espace mémoire du sous-arbre à gauche et/ou de droite, si nécessaire avec le contenu.

Passons à `insertInBST()`. S'il s'avère que l'arbre ne contient pas d'élément avec de l'information pertinente, on traite à part le cas où l'on doit affecter des valeurs au nœud racine. Dans tous les autres cas, il s'agit de parcourir tous les nœuds en se redirigeant dans le sous-arbre de gauche si la clef recherchée est inférieure à la clef du nœud courant et dans le sous-arbre de droite sinon. Une fois que l'on a atteint la bonne feuille de l'arbre, on lui associe comme fils gauche le nouveau nœud avec la clef fournie en entrée si cette dernière est plus petite que celle de la feuille, sinon ce sera le fils droit.

La fonction `searchBST()` est implémentée de manière itérative afin de rechercher un nœud potentiellement présent dans l'arbre. Il faut parcourir l'arbre tant que la bonne clef n'a pas été trouvée et que l'on a pas dépassé une feuille et se diriger dans l'arbre binaire de recherche grâce à ses propriétés intrinsèques.

`getInRange()` vérifie tout d'abord que les clefs ne sont pas dans un intervalle impossible. Si tel n'est pas le cas et étant donné que l'interface de `getInRange()` n'est pas adaptée à la récursivité (bien pratique dans notre cas), il y a appel à `getInSubRange` pour récursivement sélectionner les clefs comprises dans le bon intervalle. Si tel est le cas, on insère dans la liste liée (créée par la fonction parent) la valeur du nœud qui, dans notre cas, devrait contenir une variable de type *City**.

Une solution pour accélérer la recherche serait d'équilibrer l'arbre à chaque insertion/suppression de nœud. Étant donné que ce seront toujours les mêmes algorithmes (relatifs aux arbres) utilisés avec les même fichiers, la comparaison empirique devrait se faire normalement avec des arbres semblables. N'ayant, de plus, reçu aucune consigne à ce sujet, il n'y aura pas d'équilibrage dans ce projet.

Afin de rechercher les villes, nous utiliserons,les unes après les autres, les trois approches différentes décrites dans l'énoncé :

— La première approche est implémentée dans `find1BST.c`. Il faut tout d'abord créer un arbre de recherche binaire à l'aide de la fonction `newBST` accessible grâce au fichier d'en-tête `BinarySearchTree.h`,

y placer à chaque nœud une ville et sa latitude (clef) associée et enfin renvoyer sous forme de liste chaînée toutes les villes comprises entre deux latitudes grâce à `getInRange`.
Une fois cela fait, on parcourt la liste des villes afin de rajouter dans la liste *filtered*, les bon éléments de *notFiltered*. Notons que la fonction de comparaison se doit d'être implémentée dans ce fichier et compare des *double*.

— La deuxième approche est implémentée dans `find2BST.c`. Le principe (avec deux arbres ici) est le même qu'au point précédent, exception faites bien sûr qu'une fois les deux (latitude-longitude) listes chaînées obtenue grâce à `getInRange`, il faut encore calculer leur intersection. Cette dernière opération est effectuée grâce à `intersect()` qui prendra en plus en argument une fonction de comparaison qui traite des *villes*. Cette dernière utilise un tri lexicographique grâce à la librairie `string.h`. Ainsi, `intersect` s'avèrera être plus général. Cette fonction recopie la première liste chaînée dans un arbre, pour après seulement comparer les clefs de ce dernier avec la deuxième liste chaînée. Cette approche est plus performante que celle où l'on compare les éléments de la liste un par un (en $O(n^2)$).

— Enfin, l'implémentation de `findCitiesZBST.c` utilise un arbre binaire de recherche basé sur le code de Norton. Plus ou moins le même principe qu'avant, mais ici, il faudra utiliser `zEncode()` avec les coordonnées de chaque ville pour placer les résultats dans les clefs des nœuds. On trie une première fois avec `getInRange` (en "zcodant" les coordonnées d'abord), puis l'on trie la liste chaînée obtenue comme au point 2.

2.2 Pseudo-code de getInRange

```
getInRange(tree, keyMin, keyMax)
1  if keyMin > tree.keyMax or keyMax < tree.keyMin
2        return NIL
3
4  ll = newLinkedList()
5  getInSubRange(tree, tree.root, keyMin, keyMax, ll)
6  return ll

getInSubRange(tree, node, keyMin, keyMax, ll)
 1  if node ≠ NIL
 2
 3        if keyMin < node.key
 4              getInSubRange(tree, node.left, keyMin, keyMax)
 5
 6        if keyMin <= node.key <= keyMax
 7              insertInLinkedList(ll, node.value)
 8
 9        if node.key <= keyMax
10              getInSubRange(tree, node.right, keyMin, keyMax)
```

2.3 Analyse de la complexité de getInRange

Le pire cas :

Le pire cas correspond à une situation où il faut comparer la clef de chacun des nœuds pour vérifier si elle appartient bien au bon intervalle, c'est-à-dire quand toutes les clefs de l'arbre vérifient la condition. Il n'y a alors pas moyen de prendre de raccourci et de ne pas aller vérifier un sous-arbre.

La création de la nouvelle liste liée se fait en $O(1)$ car elle ne dépend pas de la taille de la structure de donnée. Par contre, vu qu'il faut "visiter" tous les nœuds, la fonction `getInSubRange` sera en$O(N)$ tout

comme l'insertion dans la liste chaînée. Ainsi la fonction parent devrait être en $O(N) + O(N) + O(1) = O(N)$.

Le meilleur cas cas :
Le meilleur cas devrait être en $O(log_2 N)$. En effet, si les deux clefs fournies ne sont pas dans le bon intervalle, seule une des deux conditions (gauche ou droite) est remplie et la complexité devrait être bornée par la hauteur de l'arbre complet.

2.4 Pseudo-code de intersect

```
INTERSECT(listA, listB)
 1  tree = NEWBST()
 2  for listA.head to listA.last
 3       INSERTINBST(tree, listA.city, listA.city)
 4
 5  intersection = NEWLINKEDLIST()
 6  for listB.head to listB.last
 7       if SEARCHBST(tree, listB.city) == found
 8            INSERTINLINKEDLIST(intersection, searchResult)
 9
10  return intersection
```

2.5 Analyse de la complexité de intersect

Le pire cas :
Dans le pire cas, il est nécessaire d'aller chercher systématiquement tout en bas de l'arbre de recherche binaire dégénéré pour trouver une clef identique à la ville stockée dans la deuxième liste chaînée. Ainsi, il est nécessaire de parcourir tout les nœuds de cette liste et de parcourir pour chacun d'eux tous les nœuds de l'arbre jusqu'au dernier. Nous avons ainsi une complexité en $O(M.N)$.

Le meilleur cas cas :
Le meilleur cas devrait être en $O(Mlog(N))$. Il faut dans tous les cas parcourir l'entièreté de la liste chaînée B. Mais dans le meilleur cas, l'arbre est équilibré donc la rechercher est de complexité logarithmique.

2.6 Comparaison empirique des différentes méthodes

Le tableau ci-dessous reprend la moyenne des temps d'exécution sur 20 mesures sans considérer le temps d'exécution nécessaire à la construction des arbres.

Fonction	`findCities1BST()`	`findCities2BST()`	`findCitiesZBST()`
Temps d'exécution	0,155	3,3745	0,1325

FIGURE 1 – Moyenne des temps d'exécution en fonction de l'approche considérée pour la résolution du problème

On remarque ainsi que c'est la troisième approche qui est la plus performante. L'utilisation du code de Morton pour l'encodage des clefs semble donc être la meilleure solution pour la recherche de villes. À l'inverse, on évitera d'utiliser deux arbres binaires comme il est proposé dans la deuxième approche. En effet, il faut appliquer deux fois l'algorithme de `getInRange` pour ensuite seulement calculer l'intersection des deux structures de données, qui elle même va devoir reconstruire un arbre et le comparer à une liste chaînée. On remarquera également que le temps d'exécution devient plus conséquent lorsque le nombre de villes dans le bon intervalle est plus important. Cela semble logique quand on pense qu'il faut rajouter tous ces résultats également dans des listes liées.