Université de Liège

Structures de données et algorithmes

Projet 2 : arbres binaires de recherches

2ème bachelier en ingénieur civil

Auteur :
Antoine Louis
20152140

Professeurs :
P. Geurts
J-M. Begon
R. Mormont

Année académique 2016-2017

1 Explication des choix d'implémentation

1.1 Implémentation de l'arbre binaire de recherche

La première chose à faire est de définir la structure d'un arbre binaire de recherche. cette dernière va contenir trois champs : un pointeur `root` vers une structure `Node` dans le but d'accéder à l'arbre par sa racine, une variable `size` définissant la taille de l'arbre en question, c'est-à-dire le nombre de nœuds de l'arbre, et un pointeur `comparison_fn_t` qui nous permettra de comparer tout type de variable, utile par exemple lorsque nous comparerons des structures de données du type `City`. La structure `Node`, quant à elle, contient un pointeur vers son fils gauche, un autre vers son fils droit et un dernier vers son parent. De plus, elle contient deux pointeurs sur tout type. Le premier représente la clé du nœud qui sera soit un réel (latitude ou longitude), soit un entier (pour le code de Morton). L'autre servira à reprendre la ville et ses coordonnées.

Il convient de créer une fonction qui pourra libérer la mémoire allouée pour l'arbre quand on aura fini de l'utiliser. Pour cela, on crée une fonction `freeNode` qui permettra de supprimer un sous-arbre en partant d'un nœud donné. Ici, une approche par récursivité semble efficace. La fonction va se réappeller à gauche et à droite du nœud donné tant que ce dernier n'est pas vide. Une fois que celui-ci sera vide, la fonction remontera progressivement dans l'arbre en supprimant chacun des nœuds de l'arbre ainsi que leur contenu si `freeContent` est vérifié. Il suffira ensuite d'envoyer la racine de l'arbre considéré en argument de `freeNode` pour supprimer ce dernier.

Pour insérer un nouvel élément dans l'arbre, on cherche tout d'abord par une boucle itérative la place de ce nouveau nœud en respectant l'ordre infixe dans lequel sont classées les clés de chaque nœud. Une fois la place de la nouvelle feuille trouvée, on lui assigne son parent. On n'oublie pas d'incrémenter d'une unité la taille de l'arbre en question.

En ce qui concerne la recherche d'un élément de l'arbre par sa clé, il suffit de créer une boucle itérative qui ne s'arrêtera que lorsque la clé recherchée aura été trouvée dans l'arbre ou bien lorsqu'on aura parcouru tout l'arbre sans trouver la clé correspondante. En partant de la racine de l'arbre, il suffira alors de se déplacer vers le fils gauche si la clé recherchée est plus petite ou bien vers le fils droit si elle est plus grande.

La fonction `getInRange` nous permettra d'obtenir une liste liée contenant les villes comprises entre deux clés données. Son implémentation est simple. Après avoir créer une nouvelle liste liée, la fonction `getInRange` fera appel à une autre fonction nommée `InOrderTreeWalk`. Cette fonction récursive parcourra l'arbre binaire dans le sens infixe et intégrera dans la nouvelle liste liée les éléments de l'arbre compris entre les deux clés indiquées.

1.2 Implémentation de la recherche des villes avec un arbre binaire

Le but de cette première approche est de stocker les villes données dans un arbre binaire dont les clés représenteront les latitudes de ces villes, et puis de rechercher les villes appartenant au carreau donné à partir de cet arbre binaire de recherche. En exploitant la liste liée contenant l'ensemble des villes à filtrer, nous allons créer un arbre binaire de recherche et le compléter de manière infixe avec une boucle itérative. Nous utiliserons la

fonction `insertInBST` pour insérer chaque nœud dont la clé sera la latitude de la ville et la valeur représentera la ville elle-même avec son nom, sa latitude et sa longitude.

Une fois cet arbre créé, nous utiliserons la fonction `getInRange` entre les deux latitudes données afin de créer une liste liée comprenant toutes les villes comprises entre ces latitudes.

La dernière étape consistera à filtrer cette liste liée entre les deux longitudes données de façon à obtenir l'ensemble des villes comprises dans le carreau rectangulaire donné. Pour cela, nous reprendrons presque la même implémentation que la fonction `findCitiesList`, seulement nous l'appliquerons entre deux longitudes.

1.3 Implémentation de la recherche des villes avec deux arbres binaires

Dans cette approche, il convient d'implémenter deux arbres binaires : un dont les clés seront les latitudes des villes et l'autre dont les clés seront les longitudes des villes. A partir de la liste liée donnée contenant l'ensemble des villes à filtrer, on remplit progressivement ces deux arbres grâce à `insertInBST`. Une fois les deux arbres créés et remplis, on utilise la fonction `getInRange` sur l'arbre dont les clés sont les latitudes pour créer une première liste liée contenant l'ensemble des villes comprises entre deux latitudes. On réeffectue la même opération sur l'arbre dont les clés sont les longitudes pour obtenir une liste liée des villes comprises entre deux longitudes. Il suffit ensuite de rechercher l'intersection de ces deux listes pour obtenir l'ensemble des villes appartenant au carreau recherché.

Cette dernière étape sera implémentée dans une fonction `intersect` qui se basera sur deux boucles *while* imbriquées. En effet, pour chaque élément de la première liste, on parcourra l'entièreté de la seconde liste afin de trouver un élément en commun, c'est-à-dire un élément ayant exactement les mêmes coordonnées géographiques. Si un élément de la seconde liste est en commun avec l'élément courant de la première liste, nous utiliserons l'instruction *break* pour éviter de continuer à comparer cet élément avec le reste de la seconde liste. Nous sortirons ainsi de la seconde boucle et passerons directement à l'élément suivant de la première liste.

1.4 Implémentation de la recherche des villes avec un arbre binaire utilisant Z-score

La dernière approche consiste à utiliser un arbre binaire de recherche dont les clés sont cette fois une combinaison de la latitude et de la longitude. Pour créer une telle combinaison, on utilisera le code Morton qui consiste à entrelacer les bits de coordonnées. Ici encore, la première étape consiste à créer l'arbre binaire à partir de la liste liée contenant l'ensemble des villes. De la même manière, on utilise une boucle itérative qui, grâce à la fonction `insertInBST`, insérera dans l'arbre les éléments de la liste. On utilisera ici un tableau de clés qui contiendra chacune des combinaisons "latitude-longitude". On créera ensuite avec `getInRange` une liste liée contenant les villes dont les clés sont comprises la combinaison des latitude et longitude minimales et la combinaison des latitude et longitude maximales. On finira alors par filtrer cette liste pour les coordonnées géographiques du carreau donné. Nous utiliserons pour cela la même implémentation que la fonction `findCitiesList`.

2 Pseudo-code de la fonction `getInRange`

```
1  getInRange(bst, keyMin, keyMax)
2    list = newLinkedList()
3    InOrderTreeWalk(bst.root, list, keyMin, keyMax, bst.comparison)
4    return list
```

```
1  InOrderTreeWalk (node, list, keyMin, keyMax, comparison)
2    if node==NIL
3      return
4    if(comparison(node.key, keyMin) > 0)
5      InOrderTreeWalk(node.left, list, keyMin, keyMax, comparison)
6    if(comparison(node.key, keyMin)>=0 && comparison(node.key, keyMax)<=0)
7      insertInLinkedList(list, node.value)
8    if(comparison(node.key, keyMax) <= 0)
9      InOrderTreeWalk(node.right, list, keyMin, keyMax, comparison)
10   return
```

3 Analyse de la complexité de `getInRange`

Pire cas Le pire cas correspond à un tri entre la clé minimale et la clé maximale d'un arbre binaire de recherche quelconque. Il faudra alors parcourir les N nœuds de l'arbre et les insérer un à un dans la liste liée. La complexité résultante est par conséquent $\Theta(N)$.

Meilleur cas Dans le meilleur cas, l'arbre est équilibré. Les recherches des clés *keyMin* et *keyMax* ont alors chacune une complexité $\Theta(logN)$, à laquelle il faut ajouter l'insertion des k points appartenant à l'intervalle correspondant. La complexité résultante est alors $\Theta(logN + k)$, que l'on peut simplifier en $\Theta(logN)$.

4 Pseudo-code de la fonction `intersect`

```
1  intersect(listA, listB, comparison)
2    intersection = newLinkedList()
3    currentA = listA.head
4    currentB = listB.head
5    while(currentA ≠ NIL)
6        currentB = listB.head
7        while(currentB ≠ NIL)
8            if(comparison(currentA.value, currentB.value)==0)
9                insertInLinkedList(intersection, currentA.value)
10               break
11           currentB = currentB.next
12       currentA = currentA.next
13   return intersection
```

5 Analyse de la complexité de `intersect`

Analysons à présent la complexité en temps de cette fonction en tenant compte des tailles respectives des deux listes. On fait l'hypothèse que $M \leq N$, où M est la taille de la première liste et N celle de la seconde. Nous allons étudier cette complexité dans

le pire et le meilleur cas. Notons que dans tous les cas, il faudra au moins effectuer M comparaisons.

Pire cas Dans le pire cas, la première liste n'a aucun élément en commun avec la seconde liste. La fonction va alors comparer N fois chacun des éléments de la première liste. La complexité résultante sera $\Theta(M * N)$.

M :

a_1	a_2	a_3	a_4	a_5	a_6	a_7	...	a_M

N :

b_1	b_2	b_3	b_4	b_5	b_6	b_7	...	...	...	b_N

en supposant que $a_i \neq b_j,\ \forall i \in [1, M]$ et $\forall j \in [1, N]$, avec $M \leq N$.

Meilleur cas Dans le meilleur cas, tous les éléments de la première liste se retrouvent au début de la seconde, dans le même ordre de parcours. Il faut alors faire au plus m^2 comparaisons et insérer M éléments dans la liste. La complexité résultante sera alors $\Theta(M^2 + M)$ et donc $\Theta(M^2)$.

M :

e_1	e_2	e_3	e_4	e_5	e_6	e_7	...	e_M

N :

e_1	e_2	e_3	e_4	e_5	e_6	e_7	...	e_M	...	...	e_N

Notons que tant dans le pire et que dans le meilleur cas, la complexité de la fonction est très mauvaise et le temps d'exécution de celle-ci sera d'autant plus conséquent lorsqu'il s'agira de filtrer des villes parmi un très grand ensemble. Pour améliorer cette dernière, il aurait été judicieux de trier au préalable les listes de villes à l'aide d'un algorithme de tri et puis d'en rechercher les éléments communs.

6 Comparaison empirique des trois approches

6.1 Mesure des temps de recherche pour différentes zones

En nous basant sur les différents fichiers de villes donnés, nous allons à présent comparer au mieux les différentes approches. Pour cela, nous allons tirer 10 fois deux points complètement aléatoires délimitant une zone de recherche.

	Point A	Point B
Zone 1	(22°, -68 °)	(85°, 44 °)
Zone 2	(-77°, 15 °)	(39°, 93 °)
Zone 3	(6°, -37 °)	(52°, -17 °)
Zone 4	(-78°, 46 °)	(63°, 172 °)
Zone 5	(-12°, -123 °)	(14°, -19 °)
Zone 6	(62°, 112 °)	(88°, 178 °)
Zone 7	(-26°, 7 °)	(42°, 12 °)
Zone 8	(14°, 31 °)	(27°, 154 °)
Zone 9	(68°, -115 °)	(86°, 9 °)
Zone 10	(61°, 134 °)	(73°, 147 °)

Nous allons à présent mesurer le temps de recherche des villes dans ces zones pour chacune des approches. Commençons par exemple par les temps de recherche pris pour filtrer les villes parmi un ensemble de 100000 villes.

	`findCities1BST`	`findCities2BST`	`findCitiesZBST`	N^{bre} de villes filtrées
Zone 1	0,02 s	141,49 s	0,02 s	31856
Zone 2	0,02 s	150,67 s	0,03 s	24961
Zone 3	0,02 s	0,12 s	0,00 s	63
Zone 4	0,05 s	201,75 s	0,02 s	43567
Zone 5	0,01 s	5,18 s	0,01 s	2420
Zone 6	0,00 s	0,46 s	0,00 s	117
Zone 7	0,02 s	3,85 s	0,01 s	1264
Zone 8	0,01 s	29,1 s	0,02 s	9229
Zone 9	0,00 s	0,27 s	0,02 s	17
Zone 10	0,00 s	0,02 s	0,00 s	29

6.2 Moyenne des temps de recherche pour chaque approche

Nous allons effectuer la même chose pour un ensemble de 10, 1000, 10000 et 1000000 de villes et calculer pour chacun la moyenne des temps de recherche de chaque méthode. Nous reportons nos résultats dans le tableau ci-dessous :

	10	1000	10000	100000	1000000
`findCities1BST`	0,000 s	0,000 s	0,000 s	0,015 s	0,305 s
`findCities2BST`	0,000 s	0,001 s	0,107 s	53,291 s	$\sim$ 8470 s
`findCitiesZBST`	0,000 s	0,000 s	0,000 s	0,013 s	0,285 s

6.3 Conclusions

En analysant nos résultats, on se rend compte que les temps d'exécution pris par les fonctions `findCities1BST` et `findCitiesZBST` sont assez similaires. Cette observation est assez cohérente puisque ces deux fonctions se basent sur le même algorithme, à savoir une recherche des villes au moyen d'un unique arbre binaire qui, par la suite, sera filtré sous la forme d'une liste liée pour donner les villes appartenant au carreau. On note toutefois que `findCitiesZBST` est sensiblement plus rapide pour un grand ensemble de villes à trier (supérieur à 100000).

Sans surprise, la fonction `findCities2BST` est la moins performante au niveau du temps d'exécution puisque sa complexité théorique est de l'ordre de N^2. On remarque également que plus il y a des villes appartenant au carreau recherché, plus le temps d'exécution augmente.

En conclusion, c'est la fonction `findCitiesZBST` qui, avec nos choix d'implémentation, répond au mieux à la tâche demandée.