

Arbres binaires de recherche de villes

***Projet à rendre avant le 15 janvier 2018**

Structures de données et algorithmes

1 Objectif

L'objectif du projet est d'écrire un algorithme permettant de retrouver toutes les villes comprises dans un carreau rectangulaire délimité par les points de latitude et longitude $(\phi_1, \lambda_1), (\phi_2, \lambda_1), (\phi_1, \lambda_2), (\phi_2, \lambda_2)$. Pour ce faire nous nous proposons d'étudier les trois approches suivantes :

Un arbre binaire de recherche La première approche, la plus simple, consiste à stocker les villes en utilisant une des coordonnées comme clé (par exemple la latitude). Il s'agira donc de

1. Rechercher S_ϕ , l'ensemble des villes comprises entre deux latitudes ;
2. Filtrer S_ϕ pour ne garder que les villes comprises entre deux longitudes S .

Deux arbres binaires de recherche La seconde approche, consiste à stocker les villes dans deux arbres binaires de recherche. Le premier admet comme clé les latitudes des villes et le second leur longitude. Il s'agira donc de

1. Rechercher S_ϕ , toutes les villes comprises entre deux latitudes ;
2. Rechercher S_λ , toutes les villes comprises entre deux longitudes ;
3. Calculer l'intersection de ces deux ensembles : $S = S_\phi \cap S_\lambda$.

Un arbre binaire de recherche avec Z-score La troisième approche repose sur un seul arbre de recherche qui utilise comme clé une combinaison de la latitude et de la longitude $k = Z(\phi, \lambda)$ qui garantit que $Z(\phi_m, \lambda_m) \leq Z(\phi, \lambda) \leq Z(\phi_M, \lambda_M)$ pour $\phi_m \leq \phi \leq \phi_M \wedge \lambda_m \leq \lambda \leq \lambda_M$. On peut alors retrouver les villes de la manière suivante :

1. On recherche S_Z , toutes les villes dont les clés sont comprises entre $Z(\phi_m, \lambda_m)$ et $Z(\phi_M, \lambda_M)$ où $\phi_m = \min\{\phi_1, \phi_2\}, \lambda_m = \min\{\lambda_1, \lambda_2\}, \phi_M = \max\{\phi_1, \phi_2\}$ et $\lambda_M = \max\{\lambda_1, \lambda_2\}$.
2. On filtre S_Z pour ne garder que l'ensemble S des bonnes villes.

On utilisera le code de Morton comme fonction Z . Celui-ci consiste à entrelacer les bits des coordonnées (figure 1).

2 Implémentation

Afin de réaliser le projet, il est nécessaire d'implémenter un arbre binaire de recherche ainsi que l'algorithme d'intersection. Nous utiliserons des listes chaînées pour contenir les ensembles de villes. L'algorithme général de recherche des villes est quant à lui défini dans le fichier findCities.h.

*Adapté de P. Geurts, J.M. Begon, R. Mormont.

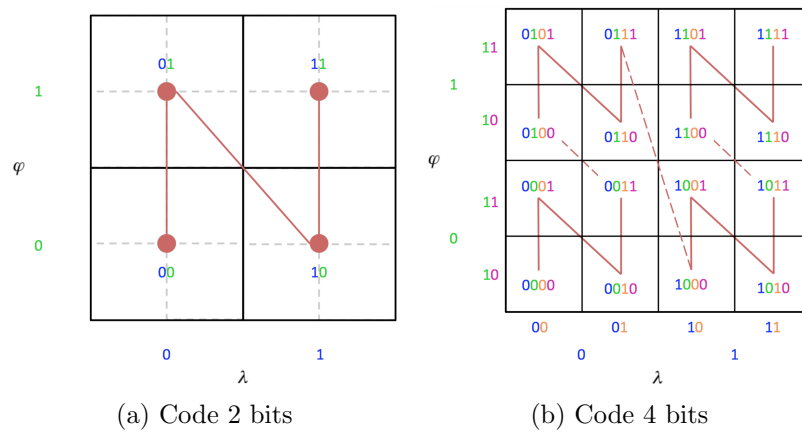


Figure 1: Codes de Morton à différentes échelles.

2.1 Arbre binaire de recherche

Il s'agit d'implémenter un arbre binaire de recherche générique ; les clés et les valeurs associées sont de type `const void*`. Dans le cadre du projet, les clés seront soit des réels (latitude et longitude), soit des entiers (code de Morton). Les valeurs seront toujours un type reprenant la ville et ses coordonnées (City). En ce qui concerne l'arbre binaire de recherche, nous vous demandons d'implémenter les opérations suivantes :

- `newBST` qui crée un nouvel arbre binaire de recherche vide ;
- `freeBST` qui libère la mémoire allouée par/pour la structure de l'arbre ;
- `sizeOfBST` qui retourne le nombre de villes contenues dans l'arbre ;
- `insertInBST` qui permet d'ajouter une nouvelle paire coordonnée-ville dans l'arbre ;
- `searchBST` qui permet de rechercher une ville sur base de sa clé ;
- `getInRange` qui renvoie une liste chaînée contenant les villes comprises dans l'intervalle spécifié en argument.

Ces opérations sont reprises dans l'interface `BinarySearchTree.h`. Vu la nature du problème, il est nécessaire d'implémenter des arbres qui peuvent contenir plusieurs instances de la même clé.

2.2 Intersection

Le fichier `intersection.h` déclare la fonction `intersect` qui renvoie une liste chaînée contenant l'intersection des deux listes chaînées fournies en entrée. Afin de rester générique, cette fonction prend également en entrée une fonction de comparaison définissant un ordre total sur les éléments des listes.

2.3 Fichiers à rendre

- `BinarySearchTree.c` l'implémentation de l'arbre binaire de recherche ;
- `intersect.c` l'implémentation de l'intersection ;
- `findCities1BST.c` l'implémentation de l'algorithme de recherche des villes utilisant un arbre binaire de recherche sur les latitudes ;
- `findCities2BST.c` l'implémentation de l'algorithme de recherche des villes utilisant deux arbres binaires de recherche ;
- `findCitiesZBST.c` l'implémentation de l'algorithme de recherche des villes utilisant un arbre binaire de recherche sur le code de Morton.

2.4 Fichiers fournis

Afin de vous aider, nous vous fournissons également les fichiers suivants :

- `main.c` un fichier chargé de parser la liste des villes, de prendre les coordonnées du carreau, de créer une liste chaînée contenant les villes et de filtrer celles en dehors du carreau ;
- `City.h` la définition de la structure `City` ;

`LinkedList.*` les fichiers relatifs à la liste chaînée ;
`zscore.*` les fichiers relatifs à l'encodage de Morton ;
`findCitiesList.c` une implémentation de `findCities` sur base des listes chaînées ;
`*.csv` plusieurs fichiers contenant des villes.

3 Analyse théorique et empirique

Dans le rapport, nous vous demandons de répondre aux questions suivantes :

1. Expliquez brièvement vos choix d'implémentation.
2. Donnez le pseudo-code de la fonction `getInRange`.
3. Analysez sa complexité dans le pire et dans le meilleur cas en fonction de N , le nombre de nœuds dans l'arbre.
4. Donnez le pseudo-code de la fonction `intersect`.
5. Analysez sa complexité dans le pire et dans le meilleur cas en fonction de M et N , la taille de la première et deuxième liste respectivement. Sans perte de généralité, on supposera que $M \leq N$.
6. Comparez empiriquement les trois approches. Pour ce faire, procédez comme suit :
 - (a) Répéter K fois :
 - i. Tirer uniformément au hasard deux points délimitant une zone $(\phi_A, \lambda_A), (\phi_B, \lambda_A), (\phi_A, \lambda_B), (\phi_B, \lambda_B)$
 - ii. Mesurer le temps de recherche 1 des villes présentes dans cette zone pour chacune des approches.
 - (b) Calculer la moyenne des K temps de recherche pour chacune des approche.
 - (c) Commenter brièvement ces résultats.

Afin d'avoir des situations comparables, assurez-vous que les arbres de recherche des différentes méthodes ont des formes similaires.

4 (Bonus) Partitionnement de l'espace

Le but de cet exercice est d'utiliser une structure de données qui permet un partitionnement hiérarchique de l'espace, tel que les *arbres k-d*, les *tétrabres (quadtree)*, les *arbres BSP* (partition binaire de l'espace) ou autre, pour améliorer la complexité de la recherche des villes.

En particulier, un *arbre 2d* permet de stocker n points en divisant l'espace récursivement en deux demi-plans à l'insertion de chaque point (figure 2).

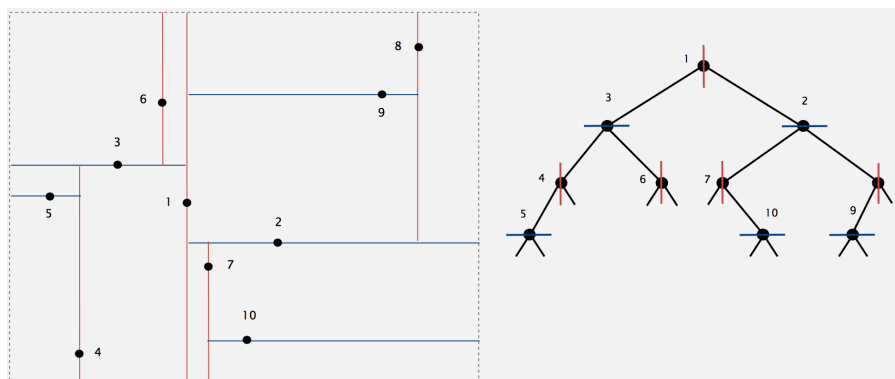


Figure 2: Construction d'un arbre 2d.

1. Donner l'algorithme de construction de l'arbre 2d.
2. Dessiner l'arbre 2d résultat de l'algorithme de construction sur les villes du fichier `cities.10.csv`.
3. Donner l'algorithme de recherche par rectangle 2d dans un arbre 2d.
4. Quelle est la complexité de la construction et de la recherche ?

5 Deadline et soumission

Le projet est à réaliser individuellement ou en groupes de 3 étudiants maximum pour le 15 janvier 2018, 23h59 au plus tard. Il doit être soumis via l'ENT, sur la page du cours :

<https://moodle.univ-paris13.fr/course/view.php?id=963>

Le projet doit être rendu sous la forme d'une archive `tar.gz` contenant :

- Votre rapport (7 pages maximum) au format PDF. Soyez bref mais précis et respectez bien la numération des (sous-)questions.
- Les fichiers mentionnés à la section 2.3.

Vos fichiers seront évalués avec les commandes :

```
gcc main.c LinkedList.c BinarySearchTree.c findCities1BST.c --std=c99 --pedantic -Wall -Wextra -Wmissing-prototypes -o boxsearch

gcc main.c LinkedList.c BinarySearchTree.c intersect.c findCities2BST.c --std=c99 --pedantic -Wall -Wextra -Wmissing-prototypes -o boxsearch

gcc main.c LinkedList.c BinarySearchTree.c zscore.c findCitiesZBST.c --std=c99 --pedantic -Wall -Wextra -Wmissing-prototypes -o boxsearch
```

en substituant adéquatement l'algorithme de recherche. Ceci implique que :

- Les noms des fichiers doivent être respectés.
- Le projet doit être réalisé dans le standard C99.
- La présence de warnings impactera négativement la cote finale.
- Un projet qui ne compile pas avec ces commandes recevra une note nulle (pour la partie code du projet).

Un projet non rendu à temps recevra une pénalité. En cas de plagiat avéré, l'étudiant se verra affecter une note nulle.