Université de Liège

INFO0902 - Structures des Données et Algorithmes

Projet 2 - Arbres binaires de recherche

Auteurs :
Maé Klinkenberg
Louan Robert

Supervision :
Pr. Pierre Geurts

16 octobre 2023

Table des matières

1 Pseudo-code de la fonction BSTRANGESEARCH

```
BSTRANGESEARCH(bst, keymin, keymax)
1  result = NIL
2  BSTTRAVERSERANGESEARCH(bst.root, result, keymin, keymax)
3  return result

BSTTRAVERSERANGESEARCH(node, result, keymin, keymax)
1  if node is not NIL
2      if node.key < keymin
3          BSTTRAVERSERANGESEARCH(node.right, result, keymin, keymax)
4      else if node.key > keymax
5          BSTTRAVERSERANGESEARCH(node.left, result, keymin, keymax)
6      else
7          BSTTRAVERSERANGESEARCH(node.left, result, keymin, keymax) ;
8          INSERT(result, node.value)
9          BSTTRAVERSERANGESEARCH(node.right, result, keymin, keymax) ;
```

2 Analyse de la complexité de BSTRANGESEARCH

2.1 Cas d'un arbre binaire de recherche équilibré

Dans le cas d'un arbre binaire de recherche équilibré, on sait que sa profondeur sera $\log_2(N)$.

Il est simple de constater que le pire cas serait où $k = N$, c'est-à-dire que l'arbre entier serait parcouru. On obtient une complexité $\theta(N)$

Aussi, de façon très triviale, on remarque facilement que le meilleur cas serait où la racine serait le seul nœud dans l'intervalle. On a une complexité $\theta(\log(N)$ car on parcourt une seule branche de chaque nœud et $k = 1$.

Le cas moyen de la recherche dans un intervalle dans un arbre binaire de recherche équilibré est également $\theta(\log(N))$. Cela se produit car l'arbre équilibré répartit uniformément les données dans l'arbre, ce qui permet une recherche plus rapide.

2.2 Cas d'un arbre binaire de recherche non nécessairement équilibré

Dans le cas d'un arbre binaire de recherche non nécessairement équilibré, on sait que sa profondeur maximale est $N - 1$.

On trouve très facilement que le pire cas serait celui où l'on doit parcourir l'entièreté de l'arbre, on a donc $k = N$ et la complexité est $\theta(N)$.

Le meilleur cas serait également celui où la racine est le seul nœud dans l'intervalle, on a $k = 1$ et la complexité est $\theta(\log(N) car on parcourt une seule branche de chaque nœud.$

3 Explication de l'implémentation des fonctions

3.1 pdctCreate et pdctBallSearch dans le cas de l'arbre binaire de recherche simple

pdctCreate : Une vérification sur l'existence des variables données en paramètre est tout d'abord faite ainsi qu'une sur la longueur des deux listes (`lpoints` et `Lvalues`). Ensuite, une structure BST est créée grâce à la fonction bstNew implémentée dans le fichier BST.c. De la mémoire est alors allouée pour la structure `PointDct` et la première structure est placée dans l'unique champ de la seconde.

La fonction parcourt ensuite les deux listes en parallèle et pour chaque paire point-valeur, elle alloue de la mémoire pour un `Tuple` (une structure qui contient la paire clef-valeur) et insère ce tuple ainsi que la clef dans l'arbre en utilisant la fonction bstInsert.

Enfin, la fonction renvoie la structure `PointDct` nouvellement créée contenant le BST avec les clefs et valeurs insérées. Notons, qu'en cas d'erreur lors de l'allocation mémoire, la fonction renvoie `NULL`

pdctBallSearch : Pour commencer, la fonction va créer deux points : un premier qui sera le minimum et un second qui sera le maximum. Pour ce faire, on diminue (resp. augmente) les coordonées (du point autour duquel on fait la recherche) x et y par la valeur du rayon pour le point minimum (resp. maximum).

Ensuite, on utilise ces deux points pour les donner à la fonction bstRangeSearch. Celle-ci nous retournera un premier filtrage, qui correspond au carré formé de médianes de longueur du rayon de recherche et dont le point d'intersection correspond au point de recherche.

Pour finir, on effectue un nouveau filtrage pour ne garder que les points qui se trouvent dans le disque autour du point de recherche.

bstTraverseRangeSearch : Si elle existe, la clef de la node est comparée aux clefs minimale et maximale. Si elle est dans l'intervalle, la valeur est insérée dans la liste et la fonction est ré-appelée pour les branches de gauche et de droite. Si la clef est plus petite, la fonction récursive est appelée pour la branche de droite ms si elle est plus grande, la fonction est appelée pour la branche de gauche.

3.2 bst2dBallSearch dans le cas de l'arbre binaire de recherche 2D

bst2dBallSearch : Premièrement, si l'arbre ne contient rien, la fonction retourne une liste vide au préalablement créée. Si l'arbre existe bel et bien, une seconde fonction est appelée. Celle-ci prend en paramètre un nœud, la liste, le point, le rayon et la profondeur de l'arbre. C'est la fonction récrusive bst2dTraverseBallSearch qui parcourra l'arbre et qui insérera les valeurs dans la liste.

bst2dTraverseBallSearch : La fonction vérifie d'abord si la node passée en paramètre existe. Elle insère ensuite la valeur du nœud dans la liste si le point qui sert de clef se situe bien dans le cercle donné par le point et le rayon des paramètres.

Si la profondeur est paire, la comparaison des points se fera sur leur coordonée X alors que si elle est impaire, cela se fera sur la coordonée Y. Si la coordonée du point donné en paramètre additionnée (resp. soustraite) du rayon est inférieure (resp. supérieure) à la coordonée du point

courant de l'arbre, alors la fonction est ré-appelée seulement pour la branche gauche (resp. droite) de l'arbre. Autrement, la fonction est ré-appelée pour les deux branches.

3.3 pdctCreate dans le cas de l'arbre binaire de recherche 2D

L'implémentation du dictionnaire pour l'arbre 2D est similaire à celle du dictionnaire pour l'arbre 1D mise à part la structure `Tuple`. Une vérification sur l'existence des variables données en paramètre est tout d'abord faite ainsi qu'une sur la longueur des deux listes (`lpoints` et `Lvalues`). Ensuite, une structure BST2D est créée grâce à la fonction bst2dNew implémentée dans le fichier BST2D.C. De la mémoire est alors allouée pour la structure `PointDct` et la première structure est placée dans l'unique champ de la seconde.

La fonction parcourt ensuite les deux listes en parallèle et chaque paire point-valeur est insérée dans l'arbre en utilisant la fonction bst2dInsert.

Enfin, la fonction renvoie la structure `PointDct` nouvellement créée contenant le BST avec les clefs et valeurs insérées. Notons, qu'en cas d'erreur lors de l'allocation mémoire, la fonction renvoie `NULL`

4 Profondeur moyenne des nœuds

4.1 Résultats

Nombre de points	10^4	10^5	10^6
BST	15.406100	20.274800	25.377456
BST2d	15.373200	19.526500	24.611424

4.2 Discussion des résultats

Plus le nombre de points dans le dictionnaire augmente, plus la profondeur moyenne des nœuds augmente également. Celle-ci augmente de ± 5 à chaque fois que le nombre de points est multiplié par 10. La profondeur moyenne des nœuds est légèrement plus grande pour un arbre binaire de recherche 1D que pour un 2D.

5 Comparaison empirique des temps de calcul des trois approches

5.1 Première expérience

5.1.1 Résultats

Nombre de recherches : 10^4 Rayon : 0.01

Opération	Génération du dictionnaire			Recherche dans une boule		
Nombre de points	10^4	10^5	10^6	10^4	10^5	10^6
Liste	0.000003	0.000002	0.000002	0.619352	7.050791	76.348466
BST	0.003980	0.080320	1.784153	0.205553	4.136310	64.389051
BST2d	0.003326	0.099867	1.617555	0.015100	0.189362	1.487918

Évolution du temps pris par opération en fonction du temps.

Opération	Recherche exacte		
Nombre de points	10^4	10^5	10^6
Liste	0.409344 | 0.499994	4.520404 | 6.041154	48.792421 | 66.911415
BST	0.004286 | 0.001219	0.011361 | 0.001621	0.020300 | 0.001960
BST2d	0.004014 | 0.002004	0.012919 | 0.002714	0.020813 | 0.003301

Évolution du temps pris par opération en fonction du temps. Les résultats à gauche sont pour une recherche positive et à droite pour une recherche négative.

5.1.2 Impact du nombre de points stockés dans la structure

Pour la création du dictionnaire sous forme de liste, le temps reste constant en fonction du nombre de points et est minime par rapport aux deux autres formes. Par contre, pour celle des arbres binaires de recherches 1D et 2D, le temps de génération du dictionnaire n'est pas constant et augmente fortement. Le temps de création du dictionnaire de BST2d est légèrement plus rapide que pour un dictionnaire BST.

Comparé aux deux autres implémentations, l'implémentation de la recherche dans une boule pour un arbre binaire de recherche 2D est très efficace. La recherche dans une boule est très inefficace dans le cas d'une liste et d'un arbre binaire, même si ce dernier s'en sort un peu mieux. De plus, pour BST2d, l'augmentation du temps est minime en fonction du nombre de points.

La recherche exacte pour des recherches négatives dans le cas d'une liste prend plus de temps que pour des recherches positives. À l'inverse, dans le cas des arbres binaires, la recherche exacte pour des recherches positives prend plus de temps que pour des recherches négatives.

5.2 Deuxième expérience

5.2.1 Résultats

Nombre de recherches : 10^4 Nombre de points : 10^5

Opération	Génération du dictionnaire			Recherche dans une boule		
Rayon	0.01	0.05	0.1	0.01	0.05	0.1
Liste	0.000002	0.000006	0.000002	7.050791	7.555664	9.082125
BST	0.080320	0.077327s	0.072176	4.136310	19.278685	44.274460
BST2d	0.099867	0.065990	0.071652	0.189362	2.478183	7.116571

Évolution du temps pris par opération en fonction du rayon de recherche dans une boule.

Opération	Recherche exacte		
Rayon	0.01	0.05	0.1
Liste	4.520404 | 6.041154	4.431629 | 6.105892	4.308002 | 6.249223
BST	0.011361 | 0.001621	0.011507 | 0.001340	0.008823 | 0.001321
BST2d	0.012919 | 0.002714	0.009249 | 0.002179	0.010443 | 0.002252

Évolution du temps pris par opération en fonction du rayon de recherche dans une boule. Les résultats à gauche sont pour une recherche positive et à droite pour une recherche négative.

Les résultats de la génération du dictionnaire et de la recherche exacte ont été redonnées à titre indicatif. Ces résultats ne seront pas discutés puisqu'ils n'ont pas d'intérêt dans notre

expérience (voir sous-section 5.1).

5.2.2 Impact du rayon de la recherche dans une boule

Le temps reste constant pour la recherche dans une boule pour le cas d'une liste. Pour l'arbre binaire de recherche, le temps augmente très fortement lorsque le rayon augmente. Celui-ci devient pire que le cas d'une liste dès que le rayon dépasse 0.01. Pour l'arbre binaire de recherche 2D, le temps de recherche est plus faible que les deux autres formes. Cependant, il augmente plus fortement que dans le cas d'une liste.

6 Conclusion quant à l'intérêt relatif des trois approches

L'implémentation avec l'arbre binaire de recherche 2D est vraisemblablement la plus efficace. En effet, la profondeur des nœuds est légèrement plus petite que celle de l'arbre binaire 1D. De plus, la majorité des opérations sont plus efficaces comparés aux deux autres implémentations, à l'exception de la recherche négative qui est légèrement moins efficace que l'arbre de recherche binaire. Un arbre binaire de recherche 2D serait donc le bon choix lorsqu'il est nécessaire de faire des recherches dans une boule. Si ce n'est pas nécessaire, un arbre binaire de recherche simple serait un meilleur choix.

7 Position ayant la densité de taxis la plus élevée à Porto

Le nombre de taxis maximal à Porto est 94049 et la position géographique sortie du programme est la longitude -8.585478 et la latitude 41.148732. Cette position correspond à la gare Campanha.