Université de Mons
Faculté Des Sciences

Structure de données II

Windowing

Professeur:
Véronique Bruyère
Assistants:
Pierre Hauweele

Auteurs:
Cyril Moreau
Arnaud Moreau

Année académique 2022-2023

Contents

1 Questions préliminaires

1.1 Question 1

Si on ne considère que les coordonées en x, voyez-vous que celles-ci sont organisées selon une file à priorité ? Où est la coordonées minimum (maximum) ? Celle file à priorité est-elle un tas ? Justifiez.

Tout d'abord, une fille de priorité est un arbre binaire tel que pour tout noeud, la donnée qui s'y trouve est supérieure ou égale (resp. inférieure ou égale) à celles de ses fils.

En ne considérant que les coordonnées en x, on remarque qu'elles sont arrangées tel que la valeur de chaque noeud est toujours inférieure ou égale à celle de ses fils ce qui correspond bien à la définition d'une file de priorité.

La coordonnée minimum est la racine tandis que la coordonnée maximum est une feuille de l'arbre.

Cependant, ce n'est pas un tas car elle ne respecte pas l'équilibre du tas. En effet, en plus d'avoir la caractéristique d'une file de priorité, un tas doit être équilibré tel que ses niveaux sont complètement remplis de noeuds excepté éventuellement le dernier niveau qui doit commencer à être rempli *par la gauche*. En effet, un arbre de recherche à priorité n'est pas forcément rempli par la gauche.

Par exemple, prenons l'ensemble U = $\{(1,5),(2,3),(3,7),(4,2),(5,8)\}$ qui donnerait un arbre de recherche à priorité comme ceci (le numéro des noeuds représente la coordonnée x du noeud) :

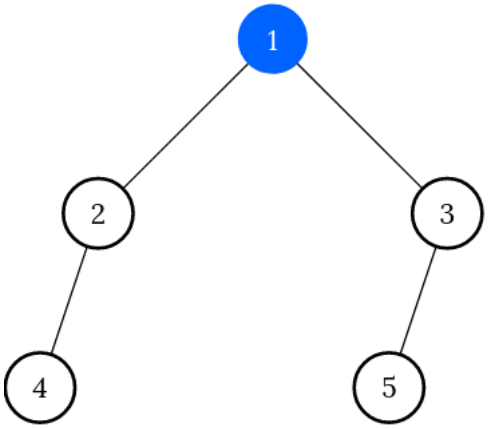

Figure 1: PST construit à partir de l'ensemble U

1.2 Question 2

Si on ne considère que les coordonées en y, voyez-vous que celles-ci sont organisées selon un arbre binaire de recherche ? Où est la coordonée minimum (maximum) ? Justifiez.

Si on ne considère que la coordonnée y, on pourrait penser que ce n'est pas un ABR cepandant la valeur qui est stockée dans chaque noeud est y_{mid} et non simplement y donc il faut regarder par rapport à cela.

La définition d'un ABR est :
Un arbre binaire de recherche est un arbre binaire tel que pour tout noeud x, la donnée qui s'y trouve est

- $<$ aux données du sous-arbre gauche de x
- $>$ aux données du sous-arbre droit de x

Or, dans la définition de la structure de donnée employée, il est dit que l'on note P un ensemble de points et y_{mid} la médiane des coordonnées de tous les points sauf la racine. Aussi, on note

$$P_{below} := \{p \in P \backslash \{p_{min}\} \mid p_y < y_{mid}\}$$
$$P_{above} := \{p \in P \backslash \{p_{min}\} \mid p_y > y_{mid}\}$$

La structure de données est construite telle que la racine possède deux sous-arbres : P_{below} comme sous-arbre gauche et P_{above} comme sous-arbre droit. Il est donc clair que nous avons affaire à un ABR, puisque pour toutes les données d'un noeud x, y_{mid} est supérieur aux données contenues dans son sous-arbre gauche et inférieure aux données contenues dans son sous-arbre droit.

La y_{mid} minimum est la feuille la plus à gauche de l'arbre tandis que le y_{mid} maximum est la feuille la plus à droite de l'arbre.

1.3 Question 3

De quelle façon est équilibré un arbre de recherche à priorité ?

Un arbre de recherche à priorité est équilibré de la même façon qu'un arbre AVL, c'est à dire que la balance de chacun de ses noeuds vaut 0, 1 ou -1. La balance d'un noeud est égale à $h_2 - h_1$ où h_2 (resp. h_1) est la hauteur de son sous-arbre droit (resp. gauche).

Un arbre de recherche à priorité respecte la condition sur les balances car, à chaque noeud, on calcule la médiane des coordonnées y et on divise l'ensemble en 2. Par définition de la médiane, il y aura toujours autant d'élément à gauche qu'à droite sauf au niveau des feuilles car il ne pourrait rester qu'un seul noeud à placer. Son père aurait donc une balance de 1 (resp. -1) ce qui respecte toujours la condition de balance.

1.4 Question 4

La construction d'un arbre de recherche à priorité peut se faire en $O(n \log_2 n)$ si n est le nombre de points de $\mathbb{R}^2$ contenus dans l'arbre. Expliquez comment on peut y parvenir et comment le prouver. Il faut sans doute utiliser une autre structure de données qui permet de calculer efficacement la médiane.

L'algorithme est facilement trouvé en suivant simplement la définition d'un arbre de recherche à priorité mais est-il bien construit en $O(n \log_2 n)$?

Algorithm 1 createPST(p)

Input : P a set of n point $\in \mathbb{R}^2$
Output : The root of the PST

```
1:  if length(data) = 1 then
2:      return node(data)
3:  else if length(data) = 0 then
4:      return null
5:  else
6:      p_min ← findMinX(data)
7:      P* ← removeRootFromData(data,p_min)
8:      y_mid ← findMedian(newDataSet)
9:      res ← node(p_min, y_mid)
10:     P_below = {p ∈ P* | p_y < p_mid}
11:     P_above = {p ∈ P* | p_y > p_mid}
12:     leftChild(res) ← createPST(P_below)
13:     rightChild(res) ← createPST(P_above)
14:     return res
15: end if
```

Les lignes 1 à 4 et 14 sont en temps constant tandis que findMinX(), removeRootFromData() et la crétation de P_{below} et P_{above} se font en $O(n)$.

Finalement, on peut montrer grâce au master theorem que l'algorithme est en $O(n \log_2 n)$. Pour simplifier les calculs, prenons que n est un multiple de 2. On a donc

$$T(n) = 2 * T(n/2) + O(?)$$

Afin d'avoir un algorithme en $O(n \log_2 n)$, il faut que le calcul de la médiane se fasse au plus en $O(n)$. Si les données sont triées selon les y, le calcul de la médiane se fait en temps constant. On peut donc trier les données en $O(n \log n)$ avec un tris par tas et ensuite créer le PST. On a donc

$$T(n) = 2 * T(n/2) + O(n)$$

Par le master theorem, on a bien que createPST() est en $O(n \log_2 n)$
car $O(n \log_2 n)$ *(tri)*+$O(n \log_2 n)$ *(création arbre)* $= O(n \log_2 n)$.

1.5 Question 5

La structure d'arbre de recherche à priorité permet d'obtenir efficacement l'ensemble des points de T qui sont contenus dans une fenêtre donnée de la forme $(-\infty : x'] \times (y : y']$. Expliquer comment adapter l'algorithme proposé pour traiter une fenêtre de la forme $[x : +\infty] \times [y : y']$, $[x : x'] \times (-\infty : y]$ ou $[x : x'] \times [y : +\infty)$ ou $[x : x'] \times [y : y']$.

Pour une fenêtre de la forme $[x : +\infty) \times [y : y']$, il faudra adapter l'intervalle passé en paramètre à QueryPrioritySearchTree. Il faudra aussi adapter la méthode ReportInSubtree afin qu'elle retourne tous les noeuds dont la coordonnée x est supérieure ou égale à la valeur passée en paramètre. Voici une telle adaptation de la méthode :

Algorithm 2 ReportInSubtree2(v, q_x)

Input : The root of a subtree of a priority search tree and a value q_x.
Output : All points in the subtree with x-coordinate at least q_x.

```
    if (p(v))_x ≥ q_x then
2:      Report p(v)
    end if
4:  ReportInSubtree(lc, q_x)
    ReportInSubtree(rc, q_x)
```

Pour une fenêtre de la forme $[x : x'] \times [y : +\infty]$ (resp.$[x : x'] \times (-\infty : y']$), la recherche de y (resp. y') consistera à toujours descendre dans le fils gauche (resp. droit) afin de "longer" l'extrémité gauche (resp. droite) de l'arbre. Il faudra aussi évidemment adapter l'intervalle passé en paramètre à QueryPrioritySearchTree ainsi que la méthode ReportInSubtree en réalisant une combinaison de la version originale et celle présentée ci-dessus.

Algorithm 3 ReportInSubtree3(v, q_x, q'_x)

Input : The root of a subtree of a priority search tree, a value q_x and a value q'_x.
Output : All points in the subtree with x-coordinate at least q_x and at most q'_x.

```
    if (p(v))_x ≤ q'_x then
        if (p(v))_x ≥ q_x then
3:          Report p(v)
        end if
        ReportInSubtree(lc, q_x, q'_x)
6:      ReportInSubtree(rc, q_x, q'_x)
    end if
```

Pour une fenêtre de la forme $[x : x'] \times [y : y']$, on utilisera ReportInSub-

TREE3 afin de borner les valeurs de x entre x et x'. Pour borner les valeurs de y entre y et y', on suivra simplement l'algorithme initial puisqu'il est appliqué à une fenêtre de la forme $[x : +\infty) \times [y : y']$. Donc, la seule différence avec l'algorithme initial est qu'il faut borner les x supérieurement, ce qui est fait en utilisant REPORTINSUBTREE3 plutôt que REPORTINSUBTREE.

1.6 Question 6

Les auteurs du Chapitre 10 font l'hypothèse que tous les points ont des coordonées bien distinctes en x et en y. Expliquer pourquoi.

Les auteurs font l'hypothèse que tous les points ont des coordonées bien disctinctes en x et en y car si ce n'était pas le cas, la construction de l'arbre de recherche à priorité ne serait pas possible avec cette définition.
En effet, prenons 2 points (6, 9) et (6, 10), leurs coordonnées en x sont égales et la définition de l'arbre nous dit que le point avec la coordonnée minimum en x doit se trouver à la racine. Dans notre exemple, on ne sait pas les distinguer sur leur coordonée en x donc on ne peut pas savoir quel point serait la racine et lequel sera le fils.

1.7 Question 7

Une technique est présentée à la page 111. Celle-ci permet de simuler des coordonnées distinctes pour un ensemble de points quelconques et une requête. Expliquez comment.

La technique est de remplacer les coordonnées x et y de chaque point par des paires notées $(x \mid y)$ pour avoir un nouveau point $((x \mid y), (y \mid x))$. L'ordre sur ces nouvelles coordonées est défini tel que pour deux paires $(a \mid b)$ et $(c \mid d)$, on a

$$(a \mid b) < (c \mid d) \iff a < c \vee (a = c \wedge b < d)$$

Avec cet ordre, on a que les coordonées sont distinctes pour un ensemble de points quelconques.

Imaginons maintenant que l'on souhaite récupérer les points d'un ensemble P qui se trouve dans la fenêtre $R = [x : x'] \times [y : y']$ en utilisant la technique vue ci-dessus. On va donc construire $\hat{\text{P}}$ l'ensemble P où chaque point $p = (p_x, p_y)$ est remplacé par $((p_x \mid p_y), (p_y \mid p_x))$.

Nous devons également modifier la fenêtre R pour qu'elle soit compatible avec la nouvelle représentation des points. on a donc :

$$\hat{\text{R}} = [(x \mid -\infty) : (x' - +\infty)] \times [(y \mid -\infty) : (y' - +\infty)]$$

1.8 Question 8

Dans le chapitre, la technique présentée est adaptée à des points. Quelles différences importantes aurait-on avec des segments de droite ? Comment peut-on adapter la technique ?

Nous devons former un seul point décrivant un segment et contenant suffisemment d'informations que pour différencier chaque segments. De cette façon, on pourra organiser ces points en un *priority search tree*. Ce point pourrait être un couple de couples définit comme suit :

Soit $S \in \mathbb{R}^2$ le segment ayant pour extrémités $E_1 = (x, y)$ et $E_2 = (x', y')$

$$P = (P_x, P_y) \text{ avec } \begin{cases} P_x = (x, x') \\ P_y = (y, y') \end{cases}$$

P aurait donc la forme $P = ((x, x'), (y, y'))$. De cette façon, à partir d'un ensemble de segments dont chacune des paires possède au plus une extrémité commune, on obtient un ensemble de segments dont chacune des paires ont des composantes distinctes.