UMONS
Université de Mons
Faculté des Sciences

Rapport final : Analyse du projet d'implémentation d'un Kd-tree

Étudiant: Mohammed KHAMLICHI
Professeure: Véronique BRUYERE
Assistant: Christophe GRANDMONT

19 avril 2025

Table des matières

1 Introduction

Dans le cadre du cours *Structures de Données II* dispensé à l'Université de Mons durant l'année académique 2024–2025, j'ai été amené à réaliser un projet individuel portant sur l'implémentation d'une structure de données avancée : le **Kd-tree** (pour *k-dimensional tree*). Ce rapport propose une analyse complète du travail que j'ai effectué afin de construire et d'exploiter cette structure dans un contexte de *recherches orthogonales* (*Orthogonal Range Searching*).

Le **Kd-tree** est largement employé dans de multiples domaines où les données présentent plusieurs dimensions :

- **Géométrie algorithmique** : pour gérer de manière efficace des interrogations spatiales ;
- **Bases de données géospatiales** : pour organiser et consulter des informations cartographiques ou géolocalisées ;
- **Vision par ordinateur et robotique** : pour trouver rapidement le plus proche voisin ou déterminer quelles zones de l'espace sont pertinentes pour un traitement ;
- **Apprentissage automatique** : dans les algorithmes « plus proches voisins » (`k-NN`), le Kd-tree accélère la recherche de proximité.

L'objectif majeur de ce *projet* est de *stocker* des données numériques bidimensionnelles (par exemple (x, y)) dans un Kd-tree et de *répondre* à des requêtes élémentaires de type SQL. Ces requêtes, fortement inspirées des besoins de la recherche orthogonale, s'écrivent sous la forme d'intervalles portant sur un ou deux critères numériques. Par exemple :

```
SELECT year FROM P WHERE courses >= 5
SELECT year, courses FROM P WHERE courses >= 5 AND year in [2, 3]
```

Pour assurer l'utilité pratique de l'application, d'autres fonctionnalités sont exigées :

- **Chargement et sauvegarde de l'échantillon** depuis un fichier texte (avec lecture ou saisie du nom de fichier) ;
- **Ajout dynamique de points** : l'utilisateur peut insérer de nouveaux objets durant l'exécution, suivis d'une reconstruction automatique de l'arbre ;
- **Recherche de propriétés globales** telles que la valeur *minimale* ou *maximale* sur une dimension (pour illustrer la facilité d'élagage dans un Kd-tree) ;
- **Affichage visuel de la structure** pour observer la hiérarchie du Kd-tree sous forme d'ASCII-art.

Sur le plan pédagogique, ce projet m'a permis de *mettre en pratique* les notions vues en cours à travers :

i) l'implémentation d'un algorithme **récursif** pour construire et parcourir un *arbre binaire* non trivial ;

ii) l'utilisation de **structures de données** spécifiques (listes triées, tas, etc.) afin de réduire la complexité ;

iii) la **gestion de cas particuliers** (doublons de coordonnées, requêtes partielles, formats de fichier) qui se présentent souvent en conditions réelles.

Organisation du rapport. Après cette introduction, la *Section suivante* propose une **description détaillée** de la structure Kd-tree, de sa *construction naïve* et des raisons pour lesquelles elle est si performante en *recherche orthogonale*. J'aborderai ensuite la **construction optimisée**, qui tire parti d'un *présortage* (via un tri par tas, `HeapSort`) pour améliorer la complexité à $O(n \log n)$. Un passage ultérieur traite des **algorithmes de recherche** (*range queries*) et détaille comment on obtient typiquement un temps $O(\sqrt{n}+k)$ en 2D.

Pour enrichir l'étude, je présenterai ensuite la *gestion des points à coordonnées identiques* (reconnaissant que la réalité n'exclut pas des valeurs x identiques) à travers l'usage de l'*espace des nombres composites*. Puis, je décrirai **toutes les étapes** de l'application réalisée, avec un focus sur les méthodes d'insertion, de sauvegarde ou encore d'affichage. Je terminerai par un **exemple d'extension** en trois dimensions, confirmant la généricité de l'approche.

Ce rapport se conclut par une **réflexion globale** sur les résultats, les difficultés rencontrées et les pistes d'amélioration potentielle. L'ensemble a pour vocation de fournir une *analyse complète* aux personnes qui désireraient s'initier ou approfondir la mise en œuvre d'un **Kd-tree** en Java, dans un contexte de *recherches orthogonales* et d'indexation multidimensionnelle.

2 Description détaillée de la structure de données Kd-tree

2.1 Définition d'un Kd-tree

Un **Kd-tree** (pour *k-dimensional tree*) est une structure de données arborescente utilisée pour organiser efficacement un ensemble de points dans un espace à k dimensions. Il s'agit d'un *arbre binaire*, dans lequel chaque nœud contient un point et divise l'espace en deux sous-espaces à l'aide d'un *hyperplan* perpendiculaire à l'un des axes.

La construction du Kd-tree se fait récursivement. À chaque niveau de l'arbre, on choisit une dimension selon un ordre cyclique. Par exemple, en 2D, on alterne entre l'axe x et l'axe y. Le point du nœud courant sert alors à séparer les points restants en deux groupes : ceux ayant une coordonnée inférieure (ou égale) sont insérés dans le sous-arbre gauche, les autres dans le sous-arbre droit.

Ce découpage hiérarchique de l'espace permet de structurer les données de manière à faciliter les recherches ultérieures, tout en limitant le nombre de régions à explorer.

2.2 Intérêt et principes d'utilisation

Le **Kd-tree** est une structure très utilisée pour accélérer les recherches dans des espaces de dimension $k > 1$. Les cas d'emploi typiques sont :

— **Recherche de plage** (*range query*) : extraire tous les points contenus dans une région orthogonale donnée (ex. un rectangle en 2D).

— **Recherche du plus proche voisin** (*nearest-neighbor*) : trouver le point le plus proche d'une requête.

— **Indexation spatiale** : organisation efficace de données géométriques ou géolocalisées.

Le principe est d'élaguer : lors d'une requête de plage, seules les branches dont la *région* intersecte le domaine recherché sont explorées, ce qui évite un balayage exhaustif.

Complexités :

— Pour une requête *ponctuelle* (ou plus généralement en 1D), on obtient en moyenne un temps $\mathcal{O}(\log n)$.

— Pour une requête *rectangulaire* en 2D, la borne classique (pire cas) est $\mathcal{O}\left(\sqrt{n} + k\right)$, où k est le nombre de points rapportés.

Le Kd-tree est donc un choix naturel dans :

— **Vision par ordinateur** : détection de voisins (segmentation, suivi d'objets).

— **Bases de données géospatiales** et SIG.

— **Jeux vidéo** : gestion des collisions ou de l'IA spatiale.

— **Robotique** : navigation et évitement d'obstacles.

— **Apprentissage automatique** : accélération des algorithmes k-NN ou de clustering.

2.3 Caractéristiques principales

2.3.1 Intuition de l'algorithme de construction

La construction d'un *Kd-tree* repose sur un principe simple mais puissant : diviser récursivement l'ensemble des points afin de construire un arbre binaire équilibré, où chaque nœud représente un sous-ensemble de l'espace.

L'idée générale se déroule comme suit :

1. À chaque étape, on sélectionne un **axe de séparation** (X ou Y en 2D), en alternant à chaque niveau de l'arbre.
2. On **trie** les points selon la coordonnée correspondant à cet axe.
3. On choisit le **point médian** dans la liste triée. Ce point devient un nœud de l'arbre, et il définit un *plan de coupure* (vertical ou horizontal) dans l'espace.
4. Les points situés **avant le médian** sont placés dans le sous-arbre gauche, ceux **après** dans le sous-arbre droit.
5. On **répète récursivement** ce processus sur chaque sous-ensemble.

Ce découpage permet d'obtenir un arbre relativement équilibré, garantissant que chaque nœud divise l'espace de manière cohérente selon une des dimensions. La **profondeur de l'arbre** détermine l'axe utilisé pour la séparation à chaque niveau.

Une fois cette intuition bien en tête, il devient possible de formaliser cette démarche à l'aide d'un algorithme précis.

2.3.2 Algorithme de construction du Kd-tree naïf

Algorithm 1 BuildKdTree(P, profondeur)

Require: Un ensemble de points P et la profondeur actuelle profondeur.
Ensure: Le noeud racine d'un Kd-tree stockant P.

if P est vide **then**
    return null
else if $|P| \leq 1$ **then**
    return une feuille stockant le point de P
else
    $k \leftarrow 2$ ▷ Dimension de l'espace
    axe $\leftarrow$ profondeur mod k
    Trier P selon la dimension axe
    indice_médiane $\leftarrow \left\lfloor \frac{|P|}{2} \right\rfloor$
    point_médian $\leftarrow P$[indice_médiane]
    valeur_coupure $\leftarrow$ point_médian[axe]
    $P_{\text{gauche}} \leftarrow P[0 : \text{indice\_médiane}]$
    $P_{\text{droite}} \leftarrow P[\text{indice\_médiane} :]$
    $\text{noeud}_{\text{gauche}} \leftarrow$ BuildKdTree(P_{gauche}, profondeur + 1)
    $\text{noeud}_{\text{droite}} \leftarrow$ BuildKdTree(P_{droite}, profondeur + 1)
    Créer un noeud avec axe, valeur_coupure, $\text{noeud}_{\text{gauche}}$ et $\text{noeud}_{\text{droite}}$.
    return le noeud créé
end if

2.3.3 Construction récursive

La construction d'un Kd-arbre (k-dimensional tree) repose sur un procédé récursif qui divise l'ensemble des points en sous-ensembles de plus en plus petits à chaque niveau de l'arbre. À chaque étape de la construction, l'axe de division est sélectionné en alternant les dimensions, ce qui permet de structurer l'espace de manière équilibrée :

— **Profondeur paire** : la division s'effectue selon l'axe x (ligne verticale).
— **Profondeur impaire** : la division s'effectue selon l'axe y (ligne horizontale).

Pour chaque noeud interne de l'arbre, la valeur de coupure est déterminée par la coordonnée médiane des points le long de l'axe sélectionné. Cette valeur médiane sert à partitionner les points en deux sous-ensembles distincts :

— Les points dont la coordonnée sur l'axe de division est strictement inférieure à la médiane sont placés dans le sous-arbre gauche.
— Les points dont la coordonnée est supérieure ou égale à la médiane sont placés dans le sous-arbre droit, incluant ainsi la médiane.

2.3.4 Alternance des dimensions de division

L'alternance des dimensions à chaque niveau de l'arbre est une stratégie clé pour maintenir un équilibre spatial dans le Kd-arbre. En alternant systématiquement les axes

de division (par exemple, x, puis y, puis z, etc., selon le nombre de dimensions), le Kd-arbre assure que toutes les dimensions sont prises en compte de manière équitable. Cette méthode présente plusieurs avantages :

— **Équilibrage spatial** : En répartissant les divisions sur toutes les dimensions, l'arbre évite une concentration excessive des séparations sur un seul axe, ce qui pourrait déséquilibrer la structure et dégrader les performances.

— **Optimisation des recherches** : Un équilibrage adéquat des divisions permet de réduire le nombre de noeuds à explorer lors des opérations de recherche, améliorant ainsi l'efficacité globale.

— **Flexibilité dimensionnelle** : Cette alternance permet au Kd-arbre de s'adapter à des espaces de dimensions supérieures, en maintenant une structure cohérente et performante.

En résumé, la construction récursive et l'alternance des dimensions de division sont des éléments fondamentaux qui confèrent au Kd-arbre sa capacité à organiser et à interroger efficacement des ensembles de points multidimensionnels.

2.3.5 Analyse de la complexité de construction naïve

Dans l'algorithme ci-dessus, à chaque appel récursif, nous trions l'ensemble des points P selon la dimension courante axe. Le tri d'un ensemble de n points prend $O(n \log n)$ temps. Comme l'algorithme effectue un tri à chaque niveau de récursion, et que la hauteur de l'arbre est $O(\log n)$, la complexité temporelle totale est donc $O(n \log^2 n)$.

Explication détaillée

— **Nombre de niveaux** : La profondeur de l'arbre est $O(\log n)$ car à chaque division, nous séparons l'ensemble des points en deux parties approximativement égales.

— **Coût par niveau** : À chaque niveau, nous effectuons un tri de l'ensemble des points restants. Si nous avons n points au niveau supérieur, nous avons environ $n/2$ points à chaque niveau suivant. L'algorithme naïf a la récurrence suivante :

$$T_{\text{naïf}}(n) = \begin{cases} O(1) & \text{si } n \leq 1, \\ O(n \log n) + 2T_{\text{naïf}}\left(\dfrac{n}{2}\right) & \text{sinon.} \end{cases}$$

Le terme $O(n \log n)$ correspond au tri des n points à chaque appel récursif.

Application du Master Theorem

Pour résoudre la récurrence $T_{\text{naïf}}(n) = 2T\left(\dfrac{n}{2}\right) + O(n \log n)$, nous appliquons le Master Theorem. Ce théorème est utilisé pour déterminer la complexité des récurrences de la forme :

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

où :

— $a = 2$: nombre de sous-problèmes.

— $b = 2$: facteur de réduction de la taille du problème.

— $f(n) = O(n \log n)$: coût supplémentaire en dehors des appels récursifs.

Étapes de l'application du Master Theorem :

1. Calcul de $\log_b a$:

$$\log_b a = \log_2 2 = 1 \quad \Rightarrow \quad n^{\log_b a} = n^1 = n$$

2. Comparaison de $f(n)$ avec $n^{\log_b a}$:

$$f(n) = O(n \log n) = \Theta(n \log n)$$

Ici, $f(n)$ est asymptotiquement plus grand que $n^{\log_b a}$ mais se situe dans la catégorie $\Theta\left(n^{\log_b a} \cdot \log^k n\right)$ avec $k = 1$.

3. Identification du cas applicable du Master Theorem :

Le Master Theorem comporte trois cas principaux :

— **Cas 1** : Si $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ pour un certain $\epsilon > 0$, alors $T(n) = \Theta\left(n^{\log_b a}\right)$.

— **Cas 2** : Si $f(n) = \Theta\left(n^{\log_b a} \cdot \log^k n\right)$ pour un certain $k \geq 0$, alors $T(n) = \Theta\left(n^{\log_b a} \cdot \log^{k+1} n\right)$.

— **Cas 3** : Si $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ pour un certain $\epsilon > 0$ et si $af\left(\frac{n}{b}\right) \leq cf(n)$ pour un certain $c < 1$ et suffisamment grand n, alors $T(n) = \Theta(f(n))$.

Dans notre cas, $f(n) = \Theta(n \log n)$ correspond au Cas 2 avec $k = 1$.

4. Détermination de la complexité :

Selon le Cas 2 du Master Theorem, si $f(n) = \Theta\left(n^{\log_b a} \cdot \log^k n\right)$, alors :

$$T(n) = \Theta\left(n^{\log_b a} \cdot \log^{k+1} n\right) = \Theta\left(n \cdot \log^2 n\right)$$

Ainsi, la récurrence se résout en :

$$T_{\text{naïf}}(n) = \Theta\left(n \log^2 n\right)$$

2.4 Exemple simple avec 5 points en 2 dimensions

Considérons un ensemble de points P dans le plan, où aucun deux points n'ont la même coordonnée x ou y (comme supposé dans la référence pour simplifier l'explication) :

— $p_1 = (2, 3)$

— $p_2 = (5, 4)$

— $p_3 = (9, 6)$

— $p_4 = (4, 7)$

— $p_5 = (8, 1)$

Nous allons construire un Kd-tree pour cet ensemble en suivant l'algorithme BuildKd-Tree décrit ci-dessous.

2.4.1 Étapes de construction du Kd-tree

Profondeur 0 (racine)

- **Axe de division** : x (profondeur 0).
- Trier P par x : $[p_1, p_4, p_2, p_5, p_3]$ avec x-coordonnées 2, 4, 5, 8, 9.
- indice_médiane $= \left\lfloor \frac{5}{2} \right\rfloor = 2$
- point_médian $= p_2 = (5, 4)$
- valeur_coupure $= 5$
- $P_{\text{gauche}} = [p_1, p_4]$
- $P_{\text{droite}} = [p_2, p_5, p_3]$

Profondeur 1 (sous-arbre gauche)

- **Axe de division** : y (profondeur 1).
- Trier P_{gauche} par y : $[p_1, p_4]$ avec y-coordonnées 3, 7.
- indice_médiane $= \left\lfloor \frac{2}{2} \right\rfloor = 1$
- point_médian $= p_4 = (4, 7)$
- valeur_coupure $= 7$
- $P_{\text{gauche}} = [p_1]$
- $P_{\text{droite}} = [p_4]$

Profondeur 2 (sous-arbre gauche du sous-arbre gauche)

- **Axe de division** : x (profondeur 2).
- $P_{\text{gauche}} = [p_1]$
- Puisque $|P_{\text{gauche}}| \leq 1$, créer une feuille contenant p_1.

Profondeur 2 (sous-arbre droit du sous-arbre gauche)

- $P_{\text{droite}} = [p_4]$
- Puisque $|P_{\text{droite}}| \leq 1$, créer une feuille contenant p_4.

Profondeur 1 (sous-arbre droit)

- **Axe de division** : y (profondeur 1).
- Trier P_{droite} par y : $[p_5, p_2, p_3]$ avec y-coordonnées 1, 4, 6.
- indice_médiane $= \left\lfloor \frac{3}{2} \right\rfloor = 1$
- point_médian $= p_2 = (5, 4)$
- valeur_coupure $= 4$
- $P_{\text{gauche}} = [p_5]$
- $P_{\text{droite}} = [p_2, p_3]$

Profondeur 2 (sous-arbre gauche du sous-arbre droit)

- **Axe de division** : x (profondeur 2).
- $P_{\text{gauche}} = [p_5]$
- Puisque $|P_{\text{gauche}}| \leq 1$, créer une feuille contenant p_5.

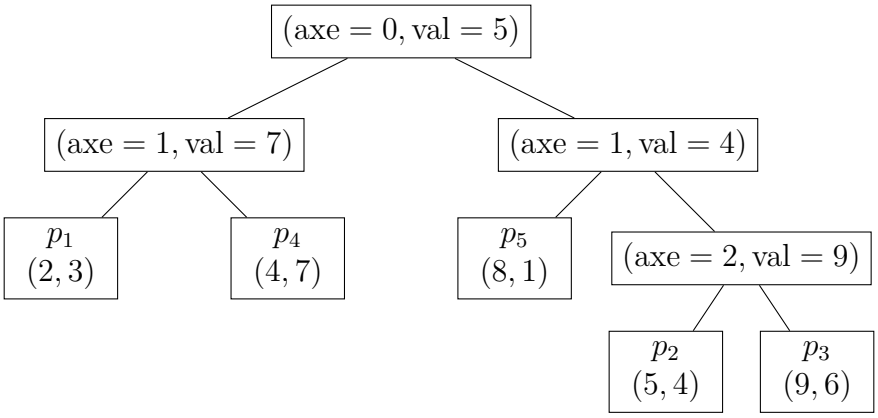
Profondeur 2 (sous-arbre droit du sous-arbre droit)

- $P_{\text{droite}} = [p_2, p_3]$
- **Axe de division** : x (profondeur 2).
- Trier P_{droite} par x : $[p_2, p_3]$ avec x-coordonnées 5, 9.
- indice_médiane $= \left\lfloor \frac{2}{2} \right\rfloor = 1$
- point_médian $= p_3 = (9, 6)$
- valeur_coupure $= 9$
- $P_{\text{gauche}} = [p_2]$
- $P_{\text{droite}} = [p_3]$

Profondeur 3 (feuilles du sous-arbre droit du sous-arbre droit)

- **Sous-arbre gauche** : $P_{\text{gauche}} = [p_2]$, créer une feuille contenant p_2.
- **Sous-arbre droit** : $P_{\text{droite}} = [p_3]$, créer une feuille contenant p_3.

2.4.2 Représentation du Kd-tree



2.4.3 Diagramme des divisions de l'espace

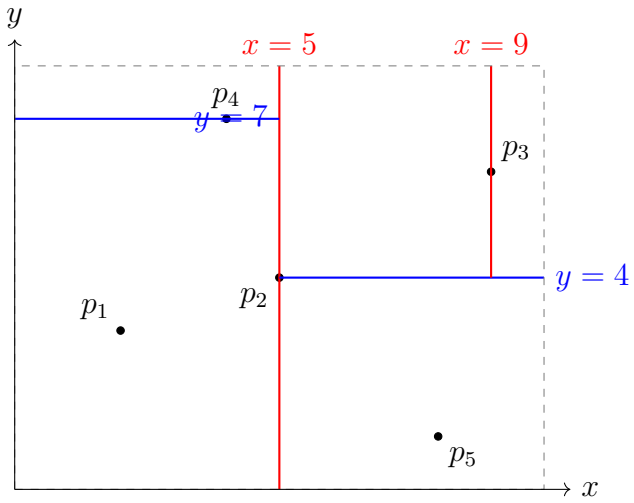


FIGURE 1 – Subdivision du plan lors de la construction du Kd-tree

3 Construction optimisée du Kd-tree

3.1 Motivation de l'optimisation

La construction optimisée d'un Kd-tree est essentielle pour améliorer l'efficacité des opérations de recherche dans des ensembles de données multidimensionnels. L'objectif principal est de réduire le temps de construction tout en garantissant un arbre le plus équilibré possible. Dans le **Chapitre 5 intitulé "Orthogonal Range Searching" du livre de Berg et al.**, il est expliqué que pour construire un Kd-tree efficacement, il est primordial de trouver la médiane des coordonnées (x ou y, selon la profondeur) à chaque étape.

Bien qu'il existe des algorithmes capables de déterminer la médiane en temps linéaire, leur mise en œuvre est souvent complexe et peu pratique. Une approche plus simple et efficace consiste à *pré-trier* l'ensemble des points avant la phase de construction proprement dite.

Dans les sections précédentes, nous avons évoqué un procédé plus *naïf* où, à chaque étape, l'algorithme trie directement les (sous-)ensembles de points suivant l'axe courant, pour extraire la médiane. Bien que conceptuellement clair, ce procédé se révèle coûteux quand l'arbre comporte plusieurs niveaux, car il multiplie les tris successifs. La complexité peut alors atteindre $\mathcal{O}(n \log^2 n)$ ou pire, suivant l'implémentation.

3.2 Présentation de l'algorithme optimisé

Pour remédier à cette redondance, le Chapitre 5 du livre propose une *approche optimisée* qui s'appuie sur l'idée suivante : **présortage** des points selon chacune des coordonnées, puis **partitionnement** en temps linéaire à chaque division.

1. Présortage des points Avant de commencer la construction, on dispose de deux listes pré-triées :

— P_x, la liste de tous les points triés selon leur coordonnée x ;

— P_y, la liste des mêmes points triés selon leur coordonnée y.

Grâce à ce présortage, on peut accéder directement au point médian de chaque liste (en $\mathcal{O}(1)$, par un simple calcul d'indice), *sans avoir à réitérer un tri* à chaque niveau de l'arbre. Cela abaisse la complexité de construction à $\mathcal{O}(n \log n)$.

2. Formation des sous-ensembles Une fois la médiane identifiée (selon l'axe de séparation choisi : x ou y), l'ensemble des points est scindé en deux sous-ensembles :

— **Sous-ensemble gauche** : les points dont la coordonnée est *strictement inférieure* à la valeur médiane.

— **Sous-ensemble droit** : les points dont la coordonnée est *supérieure ou égale* à la médiane.

Pour faciliter cette construction, on *met à jour simultanément* P_x et P_y. De cette façon, chaque sous-arbre dispose de ses propres listes triées, prêtes pour la prochaine division.

3. Gestion des doublons et contraintes Il ne doit exister *aucun doublon* : chaque point de l'ensemble doit donc être *unique*. De plus, dans un premier temps, nous imposons qu'aucun couple de points ne partage la même coordonnée x ou y. Cette restriction sera levée ultérieurement pour traiter des ensembles plus généraux, mais *les doublons restent interdits* afin d'éviter les erreurs de récursion infinie.

Si l'on souhaite lever cette seconde restriction pour intégrer des points partageant la même coordonnée, il faut alors gérer avec soin le cas où plusieurs points ont exactement la même valeur de x ou y. L'algorithme doit en effet déterminer dans quel sous-ensemble placer ces points « identiques » selon l'axe de coupe.

4. Sélection et partitionnement en temps linéaire Puisque les deux listes P_x et P_y sont déjà ordonnées, la sélection de la médiane se fait en temps constant. La création des listes $P_x^{(\text{gauche})}$, $P_x^{(\text{droite})}$ et $P_y^{(\text{gauche})}$, $P_y^{(\text{droite})}$ s'effectue en $\mathcal{O}(n)$, car il suffit de parcourir chaque liste une seule fois pour dispatcher les points.

5. Avantages de l'approche pré-triée

— **Économie sur le tri** : Les tris sont réalisés *une seule fois* au départ (P_x et P_y). À chaque niveau, on accède directement à la médiane, sans devoir re-trier.

— **Complexité améliorée** : La construction globale se fait alors en $\mathcal{O}(n \log n)$, au lieu de $\mathcal{O}(n \log^2 n)$ pour la méthode naïve.

— **Arbre plus équilibré** : En partitionnant toujours selon la médiane, on tend à répartir équitablement les points. Les recherches ultérieures (range queries, nearest neighbor) en sont d'autant plus rapides.

— **Clarté conceptuelle** : Maintenir P_x et P_y est plus simple que d'envisager un "tri 2D unique". L'algorithme sait exactement quelle liste utiliser selon l'axe de division.

Conclusion sur l'algorithme optimisé En pré-triant les points selon chaque coordonnée, on obtient un *Kd-tree* construit efficacement, sans tri redondant à chaque niveau. Le gain de performance est particulièrement significatif pour de grands ensembles de points ou lorsque l'on doit reconstruire l'arbre à plusieurs reprises.

3.2.1 Pseudo-code de l'algorithme optimisé

Algorithm 2 BuildKdTreeOptimisé(P_x, P_y, profondeur)

Require: Deux listes de points P_x et P_y, triées respectivement selon x et y, et la profondeur actuelle *profondeur*.
Ensure: Le noeud racine d'un Kd-tree optimisé contenant P_x et P_y.

if P_x est vide **then**
  return null
else if $|P_x| = 1$ **then**
  return une feuille contenant le point de P_x
else
  $k \leftarrow 2$ ▷ Dimension de l'espace
  $axe \leftarrow$ profondeur mod k
  if $axe = 0$ **then**
    $P_{\text{trié}} \leftarrow P_x$
  else
    $P_{\text{trié}} \leftarrow P_y$
  end if
  indice_médiane $\leftarrow \left\lfloor \frac{|P_{\text{trié}}|}{2} \right\rfloor$
  point_médian $\leftarrow P$[indice_médiane]
  valeur_coupure $\leftarrow$ coordonnée du axe de point_médian
  // **Partitionner P_x et P_y en sous-listes gauche et droite**
  $P_{x,\text{gauche}}$, $P_{x,\text{droite}} \leftarrow$ Partitionner P_x selon valeur_coupure sur l'axe axe
  $P_{y,\text{gauche}}$, $P_{y,\text{droite}} \leftarrow$ Partitionner P_y selon valeur_coupure sur l'axe axe
  noeud$_{\text{gauche}} \leftarrow$ BuildKdTreeOptimisé($P_{x,\text{gauche}}$, $P_{y,\text{gauche}}$, *profondeur* +1)
  $noeud_{\text{droit}} \leftarrow$ BuildKdTreeOptimisé($P_{x,\text{droite}}, P_{y,\text{droite}}, profondeur + 1$)
  Créer un noeud avec axe, valeur_coupure, noeud$_{\text{gauche}}$, $noeud_{\text{droit}}$
  return le noeud créé
end if

3.2.2 Explication détaillée de l'algorithme

L'algorithme optimisé suit les étapes suivantes :

1. **Présortage des points** : Les points sont initialement triés selon les coordonnées x et y, ce qui permet de sélectionner la médiane rapidement.
2. **Choix de l'axe de division** : L'axe est déterminé par profondeur mod k, alternant entre x et y.
3. **Sélection de la médiane** : La médiane est le $\left\lfloor \frac{n}{2} \right\rfloor$-ième élément de la liste triée correspondante.

4. **Partitionnement des points** : Les listes P_x et P_y sont partitionnées en sous-listes gauche et droite en temps linéaire.
5. **Appels récursifs** : L'algorithme est appelé récursivement sur les sous-listes pour construire les sous-arbres.

Cette méthode assure que l'arbre est équilibré et que le temps de construction est optimisé.

3.2.3 Analyse de la complexité

Complexité de l'algorithme optimisé En utilisant cette approche de *pré-tri* des points, la complexité temporelle totale de la construction du Kd-tree s'écrit comme suit :

$$T_{\text{total}}(n) \;=\; O(n \log n) \;\; (\text{présortage}) \;+\; T_{\text{construction}}(n).$$

Le temps de construction récursif, noté $T_{\text{construction}}(n)$, peut s'exprimer par la récurrence suivante :

$$T_{\text{construction}}(n) \;=\; \begin{cases} O(1), & \text{si } n \leq 1, \\ O(n) \;+\; 2\,T_{\text{construction}}\left(\frac{n}{2}\right), & \text{sinon.} \end{cases}$$

Comme vu précédemment dans l'analyse de la méthode naïve, nous pouvons appliquer le Master Theorem à la partie récursive $2\,T(\frac{n}{2}) + O(n)$. Sans reprendre en détail tous les cas, on obtient :

$$T_{\text{construction}}(n) = O(n \log n).$$

Conclusion En additionnant les deux contributions, on obtient donc :

$$T_{\text{total}}(n) \;=\; O(n \log n) \;+\; O(n \log n) \;=\; O(n \log n).$$

L'algorithme optimisé de construction d'un Kd-tree est ainsi *asymptotiquement optimal* en $\mathcal{O}(n \log n)$ temps, et l'espace mémoire requis demeure $\mathcal{O}(n)$.

3.2.4 Conclusion de l'optimisation

Grâce à cette optimisation, nous pouvons construire le Kd-tree en temps $O(n \log n)$. Le présortage initial des points selon les coordonnées x et y permet de trouver la médiane à chaque étape en temps constant et de partitionner les points en temps linéaire, ce qui améliore significativement la complexité de construction.

Lemme 1. *Un Kd-tree pour un ensemble de n points utilise $O(n)$ espace et peut être construit en $O(n \log n)$ temps.*

4 Utilisation du tri par tas dans la construction d'un Kd-tree

Lorsque l'on souhaite construire un *Kd-tree* de manière optimale, la première étape consiste à **trier l'ensemble des points** selon leurs coordonnées, avant même d'entrer dans le procédé récursif de partitionnement. Dans cette optique, le **tri par tas** (*HeapSort*) est un

choix pertinent, car il permet d'exécuter cette phase de prétraitement avec un *coût global modéré* et une implémentation relativement robuste. Cette section détaille **pourquoi** et **comment** le tri par tas intervient dans la construction d'un Kd-tree, ainsi que les implications en termes de complexité.

4.1 Principes du tri par tas (HeapSort)

Le tri par tas repose sur la notion de *tas binaire* (*binary heap*), une structure ordonnée permettant d'extraire l'élément le plus grand (ou le plus petit) en temps logarithmique. L'algorithme *HeapSort* utilise en général un *tas maximal* :

— Il débute par *construire* un tas à partir des n éléments à trier ; cette opération (build-heap) s'effectue en $\mathcal{O}(n)$ temps si l'on dispose déjà de tous les éléments.

— Il *répète* ensuite, pour i allant de n à 2, les actions suivantes :

1. **Échanger** la racine (l'élément maximum) avec l'élément à la position i.
2. **Diminuer** la taille logique du tas (on "supprime" le dernier élément qui est déjà correctement positionné en fin de tableau).
3. **Réparer** la structure de tas (heapify) pour que la nouvelle racine soit à sa position légitime, ce qui prend $\mathcal{O}(\log i)$.

Au terme de ces itérations, les éléments sont triés dans le tableau initial.

4.2 Rôle dans la construction d'un Kd-tree

Dans la construction optimisée d'un *Kd-tree*, on opte en général pour un *présortage global* sur chaque axe concerné (ex. x et y en 2D). On obtient ainsi deux listes :

— *Points triés selon l'axe x* (liste que l'on appellera P_x) ;

— *Points triés selon l'axe y* (liste P_y).

Grâce à ces deux listes, on peut choisir le **point médian** sur l'axe approprié sans devoir refaire de tri partiel à chaque étape de partitionnement. Or, pour ordonner deux fois n points (la première selon x, la seconde selon y), un **tri par tas** est particulièrement approprié, car :

— Il assure une *complexité* en $\mathcal{O}(n \log n)$ pour chacun des deux tris, maintenant ainsi le coût total du présortage dans cette borne ;

— Sa mise en place est *modulaire*, puisqu'on peut simplement changer le critère de comparaison pour distinguer le tri par x du tri par y ;

— Il n'exige pas de structure supplémentaire conséquente : un *tas* peut être représenté en place dans un tableau, évitant des allocations coûteuses.

Par la suite, l'algorithme de construction *récursive* emploie ces listes triées pour récupérer rapidement la médiane, effectuer la séparation gauche/droite, et construire récursivement les sous-arbres. Ainsi, le **tri par tas** facilite un *accès direct* aux points clés (médianes) et **abolit la nécessité de réitérer des tris locaux**.

4.3 Impact sur la complexité

En l'absence de présortage, l'algorithme "naïf" construit le Kd-tree en effectuant un tri à chaque niveau de profondeur, conduisant à une complexité typiquement en $\mathcal{O}(n \log^2 n)$. À l'inverse, lorsque l'on présort *une fois* sur chacun des axes (via HeapSort ou un équivalent), la *construction* s'effectue en $\mathcal{O}(n \log n)$, synthétisée comme suit :

1. **Présortage global :** deux passages de HeapSort ($\mathcal{O}(n \log n)$ chacun), menant au total à $\mathcal{O}(n \log n)$, puisqu'il s'agit d'une constante multipliée par $n \log n$;
2. **Partitionnement récursif :** l'extraction de la médiane est ensuite triviale ($\mathcal{O}(1)$ pour l'accès), et la répartition *gauche / droite* se fait en temps linéaire. Sur la profondeur $O(\log n)$ de l'arbre, l'ensemble de ces étapes demeure en $\mathcal{O}(n \log n)$.

Le *tri par tas* est donc un ingrédient majeur pour atteindre cette borne *quasi-optimale* et construire le Kd-tree de façon efficace, tout en conservant l'avantage d'une implémentation relativement simple et peu consommatrice en mémoire.

Au final, le **présortage par le tri par tas** constitue un *pilier* de la *construction optimisée* d'un Kd-tree, car il garantit la complétion de la phase initiale de tri en temps $\mathcal{O}(n \log n)$, ce qui, combiné au partitionnement "linéaire" à chaque niveau, permet de maintenir la **construction globale** de l'arbre dans ces mêmes bornes de complexité.

5 Algorithme SearchKdTree

5.1 Description de l'algorithme

L'algorithme SearchKdTree est conçu pour rapporter tous les points dans un Kd-tree qui se trouvent dans une plage de requête R. Il fonctionne récursivement sur le sous-arbre enraciné en un noeud ν.

5.2 Intuition et vocabulaire préalable

Avant de décrire l'algorithme SearchKdTree en détails, clarifions quelques notions clés :

— **Plage de requête R** : Dans ce document, nous cherchons tous les points qui se trouvent dans un *rectangle aligné sur les axes* (ou «intervalle» si vous préférez). Concrètement, si on appelle ce rectangle R, il peut être décrit par :

$$R = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}],$$

ce qui signifie que nous ne voulons rapporter que les points (x, y) satisfaisant

$$x_{\min} \leq x \leq x_{\max} \quad \text{et} \quad y_{\min} \leq y \leq y_{\max}.$$

Nous appellerons ce rectangle la «*plage de requête*».

— **Région associée à un nœud** : Chaque nœud ν d'un kd-tree correspond à *une portion du plan* (qu'on appellera region(ν)). Quand on descend dans l'arbre, on se «restreint» à la partie de l'espace correspondant au sous-arbre exploré. Par exemple,

si à la racine on coupe selon x=5, la region(ν) côté gauche comprendra tous les points $\{p : p.x < 5\}$, et la région droite ceux pour lesquels $p.x \geq 5$. En plus de la coupure de la racine, on accumule celles des niveaux suivants (coupures sur y, puis x, etc.), d'où la notion de «sous-arbre» correspond à un sous-domaine plus précis.

— **Parcours ciblé (SearchKdTree)** : L'algorithme de *recherche par plage* doit, idéalement, éviter de parcourir *tous* les points quand la plage de requête R est petite. Au lieu de tout vérifier un par un, on va examiner la region(ν) associée à chaque nœud ν.
  — Si region(ν) est **entièrement** à l'extérieur de R, on peut ignorer ce nœud et tout son sous-arbre (ils ne contiennent aucun point d'intérêt).
  — Si region(ν) est **complètement** incluse dans R, on sait que *tous* les points de ce sous-arbre satisfont la requête, sans vérifier chaque feuille individuellement.
  — Sinon, si region(ν) chevauche partiellement la requête R, on descend dans les nœuds fils pour chercher les points effectivement à l'intérieur.

Cet algorithme repose donc sur deux notions fondamentales : la *détection d'inclusion* (savoir qu'un sous-arbre est entièrement dans la requête) et la *détection d'exclusion* (savoir qu'un sous-arbre est entièrement hors de la requête). Ainsi, on n'explore en profondeur que les régions qui s'avèrent potentiellement utiles. C'est ce qui va nous permettre de répondre aux requêtes beaucoup plus vite que si on parcourait tous les points un par un.

Nous pouvons maintenant préciser la logique de l'algorithme SearchKdTree :

5.2.1 Pseudo-code

Algorithm 3 SearchKdTree(ν, R)

Require: Le noeud ν (racine d'un sous-arbre du Kd-tree), et une plage de requête R.
Ensure: Tous les points dans les feuilles sous ν qui se trouvent dans R.

```
1:  if ν est une feuille then
2:      Rapporter le point stocké en ν s'il se trouve dans R.
3:  else
4:      if region(gauche(ν)) est entièrement contenue dans R then
5:          ReportSubtree(gauche(ν))
6:      else if region(gauche(ν)) intersecte R then
7:          SearchKdTree(gauche(ν), R)
8:      end if
9:      if region(droit(ν)) est entièrement contenue dans R then
10:         ReportSubtree(droit(ν))
11:     else if region(droit(ν)) intersecte R then
12:         SearchKdTree(droit(ν), R)
13:     end if
14: end if
```

5.2.2 Explication détaillée

Région associée à un nœud. À chaque nœud ν est attachée une portion du plan (appelée region(ν)) : c'est la zone géométrique *réellement couverte* par ce nœud et ses

descendants. Autrement dit, si un point (x, y) ne se trouve pas dans region(ν), ce point ne pourra pas être dans le sous-arbre de ν.

1. Feuille. Ligne 1 du pseudo-code :

— Si ν est un nœud feuille, alors il ne stocke qu'un unique point (appelons-le p). On vérifie si p est dans la plage R. Si oui, on l'ajoute aux résultats.

— Après cela, on a fini d'explorer ν (il n'y a pas de fils gauche ou droit).

2. Nœud interne. Si ν n'est pas une feuille, il dispose de deux sous-arbres : gauche(ν) et droit(ν). L'idée est de décider, pour chacun de ces sous-arbres, s'il est *totalement en dehors* de la requête, *totalement à l'intérieur*, ou *partiellement recouvert* par R.

— region(gauche(ν)) $\subseteq R$: signifie que tout le sous-arbre gauche est dans la zone de requête. Dans ce cas, on peut immédiatement rapporter l'intégralité des points de ce sous-arbre (c'est la fonction ReportSubtree). On gagne du temps, car on n'a plus besoin de vérifier feuille par feuille.

— region(gauche(ν)) intersecte R : cela veut dire qu'une partie du sous-arbre gauche est dans R, l'autre partie est hors de R. On doit donc explorer récursivement : SearchKdTree(gauche(ν), R).

— Si la région gauche n'intersecte pas R du tout, on ne fait rien pour ce sous-arbre (il ne pourra contenir aucun point satisfaisant).

— On applique exactement le même raisonnement pour la region(droit(ν)).

5.3 Calcul des régions enfants et tests d'inclusion/intersection

Lorsqu'on descend dans le kd-tree, chaque nœud ν stocke une *ligne de coupure* $\ell(\nu)$, soit verticale ($x = \alpha$) soit horizontale ($y = \beta$). Cette coupure partitionne la region(ν) en deux sous-régions :

$$\text{region}(\text{gauche}(\nu)) = \text{region}(\nu) \cap \ell(\nu)_{\text{gauche}}, \qquad \text{region}(\text{droit}(\nu)) = \text{region}(\nu) \cap \ell(\nu)_{\text{droit}}.$$

Ici, $\ell(\nu)_{\text{gauche}}$ et $\ell(\nu)_{\text{droit}}$ désignent respectivement le demi-plan (ou demi-espace) à gauche/droite de la coupure si $\ell(\nu)$ est verticale, ou en dessous/au-dessus si elle est horizontale.

Maintien de la région courante. Pour savoir si region(gauche(ν)) (ou region(droit(ν))) est *entièrement incluse* dans une requête R ou la *chevauche*, on peut :

1. **Recaculer à la volée** la région courante en fonction des coupures traversées (on ne stocke pas explicitement region(ν) dans l'arbre).
2. **Pré-calculer et mémoriser** la région de chaque nœud au moment de la construction du kd-tree (chaque nœud connaît alors ses bornes en x et y).

Dans les deux cas, on dispose ensuite de bornes min/max qui permettent les tests décrits ci-après.

Tests d'inclusion et d'intersection. Dans le pseudo-code de SearchKdTree, on distingue notamment :

$$\text{region}(\nu) \subseteq R \quad \text{(inclusion totale)} \quad \text{et} \quad \text{region}(\nu) \cap R \neq \varnothing \quad \text{(intersection partielle)}.$$

Soit $R = [x_{\min} : x_{\max}] \times [y_{\min} : y_{\max}]$. Appelons $\text{region}(\nu)$ le rectangle $[X_{\min}, X_{\max}] \times [Y_{\min}, Y_{\max}]$ associé à ν. Alors :

- **Inclusion totale :**
  On dit que $\text{region}(\nu) \subseteq R$ s'il vérifie : $X_{\min} \geq x_{\min}$, $X_{\max} \leq x_{\max}$, $Y_{\min} \geq y_{\min}$, $Y_{\max} \leq y_{\max}$. Dans ce cas, on peut appeler ReportSubtree(ν) directement, sans vérifier chaque point individuellement.
- **Intersection partielle :**
  On dit que $\text{region}(\nu)$ intersecte R s'il vérifie : $X_{\max} \geq x_{\min}$, $X_{\min} \leq x_{\max}$, $Y_{\max} \geq y_{\min}$, $Y_{\min} \leq y_{\max}$. Tant que ces conditions sont satisfaites, la région chevauche R. L'algorithme descend alors récursivement pour examiner plus finement les fils de ν.
- **Exclusion totale :**
  Si la région de ν ne satisfait pas ces conditions (par exemple, si elle est entièrement au-dessus de $y_{\max}$), on ignore le sous-arbre de ν sans effectuer d'autres vérifications.

Exemple rapide.

- *Inclusion* : si $\text{region}(\nu)$ se trouve dans $[3, 7] \times [2, 5]$, on appelle ReportSubtree(ν).
- *Exclusion* : si $\text{region}(\nu)$ est entièrement au-dessus de $y = 5$, on saute ce sous-arbre.
- *Intersection* : dans les autres cas, on applique SearchKdTree récursivement.

En procédant ainsi, l'algorithme parcourt *uniquement* les sous-arbres où la région a un minimum de recouvrement avec la requête R, ce qui accélère grandement la recherche.

5.4 Analyse de la complexité en deux dimensions : Kd-trees

Dans la référence, la Section 5.2 prolonge l'idée de la recherche d'intervalle 1D (qui se fait en $O(\log n + k)$) au cas 2D, via une structure appelée *Kd-tree*. Elle permet de répondre aux requêtes rectangulaires $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ en évitant un parcours exhaustif de tous les points.

5.4.1 Passage de 1D à 2D : principe de l'alternance des coupes

Rappel en une dimension (Théorème 5.2). La référence montre qu'en 1D, on peut stocker n points dans un arbre binaire équilibré. Pour une requête $[x_{\min}, x_{\max}]$, on obtient un temps

$$O(\log n + k),$$

où k est le nombre de points effectivement rapportés (on appelle cela *sortie-sensible* car plus il y a de points à l'intérieur de la requête, plus on passera de temps à les énumérer).

Kd-tree pour 2D. En deux dimensions, on souhaite organiser les points (x, y) de façon à « couper » l'espace tantôt verticalement (niveau pair : $x = \alpha$), tantôt horizontalement (niveau impair : $y = \beta$).

— Au niveau 0 (racine), on sépare environ la moitié des points dans le *sous-arbre gauche* ($x < \alpha$) et l'autre moitié dans le *sous-arbre droit* ($x \geq \alpha$).

— Au niveau 1, on coupe horizontalement ($y = \beta$), puis au niveau 2, à nouveau vertical, etc.

Chaque nœud ν se voit ainsi associer une « région » region(ν) du plan : les coupures héritées des ancêtres de ν délimitent un rectangle (parfois non borné), contenant exactement les points du sous-arbre de ν. On peut construire cet arbre en $O(n \log n)$ si on présort les points selon (x, y) comme dans la version construction optimisée et l'espace reste linéaire en n.

5.4.2 Algorithme de requête rectangulaire $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$

Pour extraire tous les points (x, y) satisfaisant $x_{\min} \leq x \leq x_{\max}$ et $y_{\min} \leq y \leq y_{\max}$, on utilise SearchKdTree(ν, R). L'idée est de *n'explorer* que les nœuds ν dont la région region(ν) intersecte le rectangle R, en élaguant :

— **Exclusion** : si region(ν) est hors du rectangle, on ignore tout le sous-arbre ν.

— **Inclusion** : si region(ν) est complètement dedans, on appelle ReportSubtree(ν) pour rapporter ses points d'un bloc.

— **Intersection partielle** : sinon, on descend récursivement dans les sous-arbres gauche et/ou droit pour affiner la recherche.

5.4.3 Pourquoi le temps est $O(\sqrt{n} + k)$?

La référence (Lemme 5.4) prouve qu'on obtient au final un temps

$$O\Big(\sqrt{n} + k\Big),$$

où k représente le nombre de points à l'intérieur du rectangle.

Partie en k. Chaque fois qu'un sous-arbre ν est *entièrement* dans la requête, ReportSubtree(ν) parcourt ses points en temps proportionnel à leur nombre (disons k_ν). En additionnant sur tous les sous-arbres inclus, on rapporte exactement les k points finaux, d'où un coût total $O(k)$.

Partie en $\sqrt{n}$. Reste à borner le nombre de nœuds ν dits *partiels*, c'est-à-dire dont region(ν) recoupe seulement *une partie* du rectangle. On doit descendre dans ces nœuds pour déterminer quels points sont dedans.

L'analyse consiste à compter combien de « régions » peut couper la *frontière* du rectangle. D'abord, on regarde une ligne verticale $x = c$. Du fait de l'alternance (niveau pair : coupure verticale, niveau impair : horizontale), il faut descendre deux niveaux pour retrouver la même orientation (verticale). En deux niveaux, on divise l'ensemble des points par 4, d'où une récurrence

$$Q(n) \;=\; 2 \;+\; 2\,Q\Big(\tfrac{n}{4}\Big),$$

se résolvant en $O(\sqrt{n})$. Chaque côté vertical du rectangle se comporte comme une telle ligne. Les côtés horizontaux se traitent de façon similaire. Au total, la **frontière** du rectangle coupe donc $O(\sqrt{n})$ régions *partielles*. On en déduit un coût supplémentaire $O(\sqrt{n})$ en plus de $O(k)$.

Conclusion. La recherche s'effectue donc en

$$O\Big(\sqrt{n} + k\Big).$$

5.4.4 Remarques pratiques et extensions

Analyse parfois pessimiste. La preuve se base sur l'idée qu'on prolonge chaque côté du rectangle en *ligne entière*. Si le rectangle est petit, la frontière réelle est plus courte, et la pratique peut être plus favorable que $\sqrt{n}$. D'autre part, si la requête se résume à un point unique (x_0, y_0), on fait souvent un chemin unique dans l'arbre, coûtant $O(\log n)$.

Généralisations. Le Kd-tree se construit en $O(n \log n)$ avec un stockage linéaire $O(n)$. Il fournit un temps $O(\sqrt{n}+k)$ pour les requêtes rectangulaires 2D. On peut aussi l'étendre en plus de deux dimensions, le temps devenant alors $O(n^{1-\frac{1}{d}} + k)$. Pour de meilleurs temps (type $\log^d n$), d'autres structures existent (*range trees*, etc.), mais exigent davantage d'espace et d'implémentation plus complexe.

5.5 Exemple illustratif

5.5.1 Kd-tree utilisé

Pour illustrer la recherche, considérons un exemple où :

— Au niveau 0 (axe = 0), la valeur de coupure est 5. Nous décidons que x $<$ 5 va dans le sous-arbre gauche, et x $\geq$ 5 va dans le sous-arbre droit (conformément à la convention « les points inférieurs vont à gauche, les supérieurs ou égaux vont à droite »).

— Au niveau 1 (axe = 1), on fait la coupure sur y. Par exemple, si la valeur de coupure est 7, alors y $<$ 7 ira à gauche, et y $\geq$ 7 à droite.

— Idem au niveau suivant (axe = 1, val = 4), etc.

Le petit Kd-tree résultant pourrait être (schéma ci-dessous) :

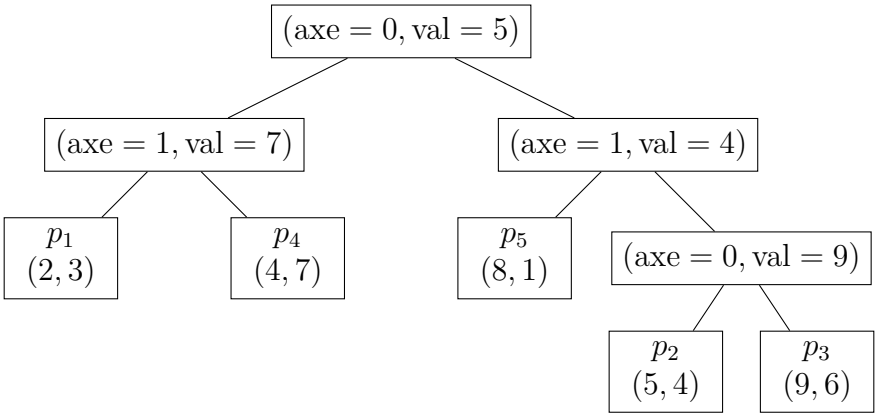


FIGURE 2 – Un exemple de Kd-tree avec la convention x $<$ val $\rightarrow$ gauche, x $\geq$ val $\rightarrow$ droit, etc.

5.5.2 Plage de requête

On se donne une requête rectangulaire R décrite par

$$x \in [3, 7] \quad \text{et} \quad y \in [2, 5].$$

5.5.3 Étapes de la recherche

Appliquons l'algorithme SEARCHKDTREE (décrit précédemment) :

1. **Noeud racine** (axe = 0, val = 5) :
   — Sous-arbre *gauche* : correspond à $x < 5$. Or $[3, 7]$ intersecte partiellement la zone $x < 5$ (puisque la plage commence à 3). On explore donc ce sous-arbre.
   — Sous-arbre *droit* : correspond à $x \geq 5$. La plage $[3, 7]$ intersecte aussi la zone $x \geq 5$ (du moins pour $5 \leq x \leq 7$). On explore également ce sous-arbre.
2. **Sous-arbre gauche** (axe = 1, val = 7) :

— Sous-arbre gauche : $y < 7$. Ici, $[2, 5]$ intersecte clairement $y < 7$, on descend donc dans ce fils.
— Sous-arbre droit : $y \geq 7$. Mais la requête $y \in [2, 5]$ ne touche pas le domaine $y \geq 7$. On l'ignore donc.

3. **Feuille** $p_1 = (2, 3)$:
   — p_1 a $x = 2$, $y = 3$. Or la plage requise est $x \in [3, 7]$, $y \in [2, 5]$.
   — Comme $2 < 3$, p_1 n'est pas dans R.
4. **Sous-arbre droit** $(\text{axe} = 1, \text{val} = 4)$:
   — Sous-arbre gauche : $y < 4$. La requête $[2, 5]$ sur y intersecte partiellement $y < 4$ (plus exactement la portion $2 \leq y < 4$), donc on explore ce fils.
   — Sous-arbre droit : $y \geq 4$. Là aussi, la requête $[2, 5]$ intersecte la portion $4 \leq y \leq 5$. On l'explore donc également.
5. **Feuille** $p_5 = (8, 1)$:
   — p_5 a $x = 8 > 7$, donc il sort de la plage $[3, 7]$. On ne le rapporte pas.
6. **Noeud** $(\text{axe} = 0, \text{val} = 9)$:
   — Sous-arbre gauche : $x < 9$. Comme la requête sur x est $[3, 7]$, elle intersecte la zone $x < 9$ (pour $3 \leq x \leq 7$). On descend.
   — Sous-arbre droit : $x \geq 9$. Cela ne touche pas $[3, 7]$, donc on ignore ce fils.
7. **Feuille** $p_2 = (5, 4)$:
   — p_2 a $x = 5$, $y = 4$. Or $x = 5 \in [3, 7]$ et $y = 4 \in [2, 5]$, donc p_2 est *dans* la requête R. On le rapporte.
8. **Feuille** $p_3 = (9, 6)$:
   — p_3 a $x = 9 > 7$. Il n'est pas dans R.

5.5.4 Résultat final

Le seul point rapporté est $p_2 = (5, 4)$.

5.6 Conclusion

Grâce à l'inclusion ou à l'exclusion rapide de sous-arbres entiers, on parcourt seulement les zones du Kd-tree potentiellement utiles. Dans cet exemple, les nœuds concernant p_1, p_5, et p_3 ont été visités, mais rejetés après comparaison, tandis que p_2 a été rapporté. Cette démarche confirme que la recherche prend un temps $O(\sqrt{n} + k)$ en deux dimensions, où k est le nombre de points effectivement extraits.

6 Gestion des cas particuliers

6.1 Problématique

Dans le domaine des structures de données géométriques, telles que les **kd-arbres** (kd-trees) ou les **arbres de plage** (range trees), il est courant de travailler avec des ensembles de points dans le plan. Ces structures sont essentielles pour effectuer des requêtes efficaces, comme la recherche de tous les points dans une certaine plage rectangulaire.

Cependant, jusqu'à présent, nous avons imposé une restriction importante : aucun couple de points ne doit avoir la même coordonnée x ou la même coordonnée y. Cette hypothèse simplificatrice facilite la construction et l'analyse des structures de données, mais elle est peu réaliste. En pratique, il est fréquent que plusieurs points partagent la même abscisse ou la même ordonnée. Par exemple, si les points représentent des données telles que les salaires ou le nombre d'enfants des employés, il est probable que plusieurs employés aient le même salaire ou le même nombre d'enfants.

Cette situation pose des défis lors de la construction des structures de données :

— **Ambiguïtés dans le partitionnement** : Les algorithmes de construction des kd-arbres et des arbres de plage s'appuient sur la recherche d'un *pivot* (par exemple la médiane) selon la coordonnée courante (x ou y). Or, si plusieurs points possèdent exactement la même valeur $x = \alpha$ (ou $y = \beta$), on ne sait plus clairement lequel de ces points doit servir de pivot. Typiquement, on veut séparer l'ensemble en deux moitiés d'à peu près la même taille, mais si dix points ont $x = 5$, on n'a plus un point de division "unique" : on risque de mettre tous ces points dans la même branche, ce qui peut déséquilibrer la structure et nuire aux performances lors des requêtes.

— **Comparaisons ambiguës** : Les opérations de comparaison des coordonnées pour naviguer dans la structure peuvent être ambiguës si les coordonnées ne sont pas distinctes. Lorsqu'on compare la coordonnée du point actuel à la coordonnée du pivot, on ne sait plus clairement s'il faut aller à gauche ou à droite quand il y a égalité (ou plusieurs égalités successives).

Il est donc essentiel de trouver une méthode pour gérer ces cas particuliers tout en maintenant les performances et l'efficacité des structures de données.

6.2 Solution proposée : Utilisation de l'espace des nombres composites

La solution proposée repose sur une observation clé : nous n'avons pas besoin que les valeurs des coordonnées soient des nombres réels spécifiques. Il suffit qu'elles appartiennent à un ensemble totalement ordonné, ce qui nous permet de comparer n'importe quelles deux coordonnées et de calculer des médianes.

6.2.1 Définition de l'espace des nombres composites

Nous allons introduire un nouvel ensemble, appelé **espace des nombres composites**, constitué de paires de nombres réels. Un **nombre composite** est défini comme une paire $(a \mid b)$, où a et b sont des nombres réels.

Nous définissons un **ordre total** sur cet espace en utilisant l'ordre lexicographique. Plus précisément, pour deux nombres composites $(a \mid b)$ et $(a' \mid b')$, nous définissons :

$$(a \mid b) < (a' \mid b') \quad \text{si et seulement si} \quad \begin{cases} a < a', & \text{ou} \\ a = a' \quad \text{et} \quad b < b'. \end{cases}$$

Cet ordre total garantit que tous les nombres composites sont comparables, ce qui est essentiel pour les structures de données qui nécessitent un tri et des comparaisons de clés.

6.2.2 Transformation des points

Supposons que nous avons un ensemble P de n points dans le plan, où les points sont distincts, mais où plusieurs points peuvent avoir la même coordonnée x ou la même coordonnée y.

Nous allons transformer chaque point $p = (p_x, p_y)$ en un nouveau point $\hat{p}$ avec des coordonnées composites :

$$\hat{p} = ((p_x \mid p_y),\ (p_y \mid p_x))$$

Ainsi, nous obtenons un nouvel ensemble $\hat{P}$ de n points avec les propriétés suivantes :

— Les premières coordonnées $(p_x \mid p_y)$ de tous les points $\hat{p}$ sont distinctes.

— Les deuxièmes coordonnées $(p_y \mid p_x)$ de tous les points $\hat{p}$ sont également distinctes.

Justification :

— Si deux points p et q ont la même coordonnée x (donc $p_x = q_x$) mais des coordonnées y différentes (puisque les points sont distincts), alors leurs premières coordonnées composites $(p_x \mid p_y)$ et $(q_x \mid q_y)$ seront distinctes, car $p_y \neq q_y$ et $p_x = q_x$.

— De même, si deux points ont la même coordonnée y mais des coordonnées x différentes, leurs deuxièmes coordonnées composites seront distinctes.

6.2.3 Construction des structures de données

Avec l'ensemble transformé $\hat{P}$, nous pouvons construire des kd-arbres ou des arbres de plage en utilisant les algorithmes standard, car les coordonnées composites permettent d'éviter les ambiguïtés liées aux coordonnées identiques.

— **Kd-arbres** : Lors de la construction du kd-arbre, nous alternons les divisions selon les coordonnées composites. À chaque nœud, nous choisissons la médiane des coordonnées composites correspondantes pour diviser l'ensemble des points, ce qui assure un arbre équilibré.

— **Arbres de plage** : De même, les arbres de plage peuvent être construits en triant les points selon les coordonnées composites et en créant les structures associées.

6.2.4 Transformation des requêtes

Lorsque nous effectuons une requête pour trouver tous les points de P qui se trouvent dans une plage rectangulaire $R = [x_{\min} : x_{\max}] \times [y_{\min} : y_{\max}]$, nous devons également transformer cette plage pour qu'elle corresponde à l'espace des nombres composites.

La plage transformée $\hat{R}$ est définie comme suit :

$$\hat{R} = [(x_{\min} \mid -\infty) : (x_{\max} \mid +\infty)] \times [(y_{\min} \mid -\infty) : (y_{\max} \mid +\infty)]$$

Ici, $-\infty$ et $+\infty$ sont utilisés pour étendre la plage de recherche, garantissant que tous les points pertinents sont inclus.

6.2.5 Preuve de la correction de l'approche

Nous devons nous assurer que cette transformation préserve la relation d'appartenance des points aux plages de requête. Autrement dit, nous voulons montrer que :

$$p \in R \quad \Leftrightarrow \quad \hat{p} \in \hat{R}$$

Lemme 5.10 : Soit p un point et R une plage rectangulaire définie comme ci-dessus. Alors :

$$p \in R \quad \Leftrightarrow \quad \hat{p} \in \hat{R}$$

Preuve :

— **Sens direct ($\Rightarrow$)** : Supposons que $p \in R$. Alors :

$$x_{\min} \leq p_x \leq x_{\max} \quad \text{et} \quad y_{\min} \leq p_y \leq y_{\max}$$

Considérons la première coordonnée composite $(p_x \mid p_y)$:

- Puisque $x_{\min} \leq p_x \leq x_{\max}$, nous avons $x_{\min} \leq p_x \leq x_{\max}$.
- Pour toute valeur de p_y, $-\infty < p_y < +\infty$.
- Donc, selon l'ordre lexicographique :

$$(x_{\min} \mid -\infty) \leq (p_x \mid p_y) \leq (x_{\max} \mid +\infty)$$

De même pour la deuxième coordonnée composite $(p_y \mid p_x)$:

$$(y_{\min} \mid -\infty) \leq (p_y \mid p_x) \leq (y_{\max} \mid +\infty)$$

Ainsi, $\hat{p} \in \hat{R}$.

— **Sens réciproque ($\Leftarrow$)** : Supposons que $\hat{p} \in \hat{R}$. Alors :

$$(x_{\min} \mid -\infty) \leq (p_x \mid p_y) \leq (x_{\max} \mid +\infty)$$

Selon l'ordre lexicographique, cela signifie que :

- $x_{\min} < p_x$, ou
- $x_{\min} = p_x$ et $-\infty \leq p_y$ (toujours vrai).

De même, pour la borne supérieure :

— $p_x < x_{\max}$, ou
— $p_x = x_{\max}$ et $p_y \leq +\infty$ (toujours vrai).

Donc, $x_{\min} \leq p_x \leq x_{\max}$.
De même pour la deuxième coordonnée :

$$(y_{\min} \mid -\infty) \leq (p_y \mid p_x) \leq (y_{\max} \mid +\infty)$$

Ce qui implique que $y_{\min} \leq p_y \leq y_{\max}$.
Par conséquent, $p \in R$.

Cette preuve démontre que la transformation préserve la relation d'appartenance des points aux plages de requête. Ainsi, interroger l'arbre construit avec les points transformés $\hat{P}$ en utilisant la plage $\hat{R}$ nous donne exactement les mêmes points que si nous avions interrogé l'ensemble original P avec la plage R.

6.3 Conclusion

En utilisant l'espace des nombres composites, nous avons résolu le problème des coordonnées identiques dans les ensembles de points. Cette méthode nous permet de :

— Éviter les ambiguïtés lors du partitionnement des points dans les structures de données géométriques.
— Maintenir les propriétés d'équilibre et d'efficacité des kd-arbres et des arbres de plage.
— Conserver la validité et la précision des requêtes effectuées sur ces structures.

Cette approche est élégante et efficace, car elle repose sur une modification minimale des algorithmes existants. En adaptant simplement la manière dont nous comparons les coordonnées, nous pouvons gérer les cas particuliers sans augmenter la complexité des structures de données ni leur coût en termes de stockage.

Elle est particulièrement utile dans les applications pratiques où les coordonnées identiques sont fréquentes, garantissant ainsi des performances optimales et une utilisation efficace des structures de données géométriques.

7 Description complète des étapes du programme

Ce projet a pour objectif de développer un programme capable de gérer efficacement des requêtes SQL basiques sur un ensemble de données numériques multidimensionnelles, en utilisant les **Kd-trees** pour optimiser les opérations de recherche. Le programme doit également être en mesure de gérer les cas particuliers où plusieurs points peuvent avoir la même coordonnée x ou y. Cette section détaille les différentes étapes du programme, en mettant en évidence l'utilisation des structures de données appropriées à chaque étape.

7.1 Étape 1 : Chargement et organisation des données

La première étape consiste à lire et organiser les données provenant d'un fichier texte ou d'une entrée utilisateur. Le format du fichier est spécifié de manière à inclure :

- Le nombre M de critères (entier).
- Les noms des M critères (chaînes de caractères).
- Le nombre N d'éléments dans l'échantillon (entier).
- Pour chaque élément :
  - Les deux premiers critères sont des nombres réels (utilisés pour le Kd-tree).
  - Les critères suivants sont des chaînes de caractères (informations supplémentaires).

Structures de données utilisées :

- `Listes` ou `tableaux` pour stocker les points de l'échantillon.
- `Classe Point` pour représenter chaque élément, avec des attributs pour les critères numériques et non numériques.

Détails de l'étape :

1. Lire le fichier ligne par ligne, extraire les noms des critères et les valeurs associées.
2. Pour chaque élément, créer une instance de la classe `Point` en initialisant les attributs correspondants.
3. Stocker tous les points dans une liste principale pour un accès facile lors de la construction du Kd-tree.

7.2 Étape 2 : Construction optimisée du Kd-tree

Nous allons construire pas à pas un *kd-tree* en 2D pour l'ensemble de points suivant, en insistant sur l'utilisation de **CompositePoint** lors du choix des pivots (split) et du partitionnement :

$$p_1 = (2, 3), \quad p_2 = (5, 4), \quad p_3 = (9, 6), \quad p_4 = (4, 7), \quad p_5 = (8, 1).$$

Dans notre code Java, la comparaison entre deux points se fait en fonction de l'**axe** courant :

- Pour `axis = 0` (séparation par x) :

$$\text{CompositePoint}(x, y, \texttt{axis} = 0) : \begin{cases} \text{primary} = [x, y], \\ \text{secondary} = [y, x]. \end{cases}$$

  On compare d'abord la composante x (primary), puis y (secondary) en cas d'égalité.
- Pour `axis = 1` (séparation par y) :

$$\text{CompositePoint}(x, y, \texttt{axis} = 1) : \begin{cases} \text{primary} = [y, x], \\ \text{secondary} = [x, y]. \end{cases}$$

  On compare d'abord y (primary), puis x (secondary) en cas d'égalité.

Dans cette étape, l'algorithme de construction du kd-tree est optimisé pour atteindre une complexité de $O(n \log n)$ grâce à deux phases essentielles : un pré-tri des points et une construction récursive reposant sur la création d'un nœud (KdTreeNode) à partir des données.

Pré-tri des points : Avant de construire l'arbre, les points sont triés selon leurs coordonnées x et y. On constitue ainsi deux listes :

— `Px` : liste des points triés en utilisant la clé définie par CompositePoint($x, y,$ `axis` $= 0$) (où la valeur « point » correspond aux coordonnées du point).

— `Py` : liste des points triés en utilisant la clé définie par CompositePoint($x, y,$ `axis` $= 1$).

Par exemple, pour nos 5 points, on obtient :

$$\texttt{pointsSortedX} = \{p_1(2,3),\ p_4(4,7),\ p_2(5,4),\ p_5(8,1),\ p_3(9,6)\},$$

$$\texttt{pointsSortedY} = \{p_5(8,1),\ p_1(2,3),\ p_2(5,4),\ p_3(9,6),\ p_4(4,7)\}.$$

Principe de l'algorithme optimisé : L'algorithme s'appuie sur ces listes triées pour déterminer rapidement la médiane (le pivot, c'est-à-dire le `split`) et partitionner l'ensemble :

1. **Sélection de l'axe de division :** L'axe utilisé est défini par la profondeur actuelle (`depth`) :
$$\texttt{axis} = (\texttt{depth} \mod 2)$$
   — Si `axis` $= 0$, on effectue la division selon x.
   — Si `axis` $= 1$, la division se fait selon y.
2. **Sélection de la médiane (le pivot/split) :** Le point médian est choisi à partir de la liste triée correspondante (`Px` ou `Py`). Ce point, transformé en `CompositePoint` via la méthode `compositeKey`, fournit la valeur de coupure (`split`) qui permettra de partitionner l'ensemble.
3. **Partitionnement des points :** Les points sont séparés en deux sous-ensembles en temps linéaire :
   — `left` : les points dont la valeur sur l'axe est strictement inférieure à la valeur de coupure.
   — `right` : les points dont la valeur sur l'axe est supérieure ou égale à la valeur de coupure.
4. **Appels récursifs et création des nœuds :** L'algorithme est appelé récursivement sur les sous-ensembles `left` et `right`. Pour chaque appel, la profondeur est incrémentée (ce qui permet d'alterner l'axe de division) et un nouveau nœud (KdTreeNode) est créé avec les champs suivants :
   — `axis` : l'axe utilisé pour la séparation.
   — `split` : le `CompositePoint` déterminé à partir du pivot.
   — `left` et `right` : les références aux sous-arbres construits récursivement.
   — Pour une feuille, le champ `point` est non nul (et les `extraCriteria` associés sont stockés), tandis que pour un nœud interne, le champ `point` est `null`.

Gestion des cas particuliers : Pour éviter les problèmes liés aux points ayant des coordonnées identiques (ce qui pourrait provoquer une récursion infinie), l'algorithme utilise l'**espace des nombres composites**. Ainsi, la comparaison se fait d'abord sur la valeur principale (du `CompositePoint`) puis, en cas d'égalité, sur la valeur secondaire.

Complexité de l'algorithme :

— Le pré-tri initial s'effectue en $O(n \log n)$.

— Le partitionnement à chaque niveau se fait en $O(n)$.

— La profondeur de l'arbre étant $O(\log n)$, la complexité totale demeure en $O(n \log n)$.

Construction récursive et exemple détaillé : Le paramètre `depth` débute à 0 et l'axe courant est défini par `axis` = (`depth` mod 2). À chaque nœud, on procède comme suit :

1. Choisir la médiane dans la liste triée correspondant à l'axe.
2. Créer le `CompositePoint` pivot (`split`).
3. Partitionner l'ensemble en sous-listes `left` et `right` en se basant sur la comparaison des clés composites.
4. Appeler récursivement la construction sur les sous-ensembles.

Exemple pour l'ensemble de points donné :

Niveau 0 (racine, `axis` = 0) Avec `pointsSortedX` = $(2,3)$, $(4,7)$, $\underline{(5,4)}$, $(8,1)$, $(9,6)$:

— Médiane : $(5,4)$.

— Le pivot (split) est défini par CompositePoint$(5, 4, \texttt{axis} = 0)$ avec primary = $[5,4]$ et secondary = $[4,5]$.

— Partitionnement :

  — `left` : $\{(2,3),(4,7)\}$ (car $x < 5$).

  — `right` : $\{(5,4),(8,1),(9,6)\}$ (car $x \geq 5$).

Niveau 1 (branche gauche, `axis` = 1) Pour les points $\{(2,3),(4,7)\}$ triés par y :

— Médiane : $(4,7)$.

— Le pivot est défini par CompositePoint$(4, 7, \texttt{axis} = 1)$ avec primary = $[7,4]$ et secondary = $[4,7]$.

— Partitionnement :

  — `left` : $\{(2,3)\}$ (car $y = 3 < 7$).

  — `right` : $\{(4,7)\}$.

Niveau 1 (branche droite, `axis` = 1) Pour les points $\{(5,4),(8,1),(9,6)\}$ triés par y (ordre : $(8,1),(5,4),(9,6)$) :

— Médiane : $(5,4)$.

— Le pivot est défini par CompositePoint$(5, 4, \texttt{axis} = 1)$ avec primary = $[4,5]$ et secondary = $[5,4]$.

— Partitionnement :

  — `left` : $\{(8,1)\}$ (car $y = 1 < 4$).

— `right` : $\{(5,4),(9,6)\}$ (car $(5,4)$ est le pivot et $(9,6)$ a $y = 6 > 4$).

Niveau 2 (dans la branche droite-droite, `axis` $= 0$) Pour les points $\{(5,4),(9,6)\}$:

— Médiane : $(9,6)$ (liste triée par x).

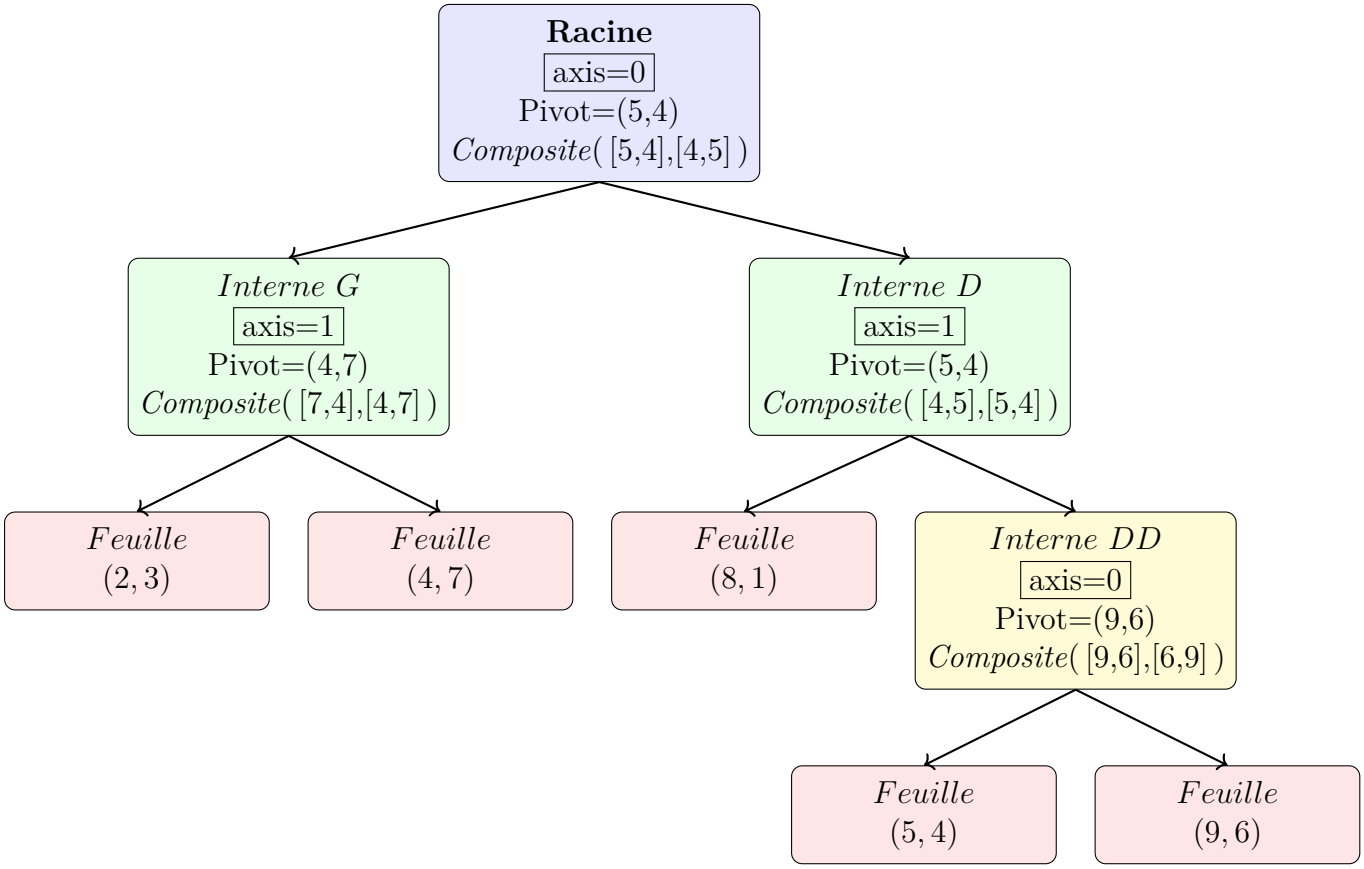
— Le pivot est défini par CompositePoint$(9, 6,$ `axis` $= 0)$ avec primary $= [9,6]$ et secondary $= [6,9]$.

— Partitionnement :

— `left` : $\{(5,4)\}$ (car $5 < 9$).

— `right` : $\{(9,6)\}$.

Représentation finale de l'arbre :



Remarques :

— À la racine (`axis` $= 0$) : le pivot (split) est $(5,4)$ avec `primary` = [5,4] et `secondary` = [4,5].

— Au niveau suivant (`axis` $= 1$) : par exemple, le pivot $(4,7)$ se traduit par `primary` = [7,4].

— La branche Droite-Droite repasse en `axis` $= 0$ avec le pivot $(9,6)$.

— Les feuilles contiennent le `point` et les `extraCriteria` associés ; elles n'ont pas de valeur dans `split`.

Conclusion : La méthode présentée permet de construire efficacement un kd-tree en 2D en utilisant des `CompositePoint` pour gérer la comparaison en fonction de l'axe courant. Chaque nœud interne (KdTreeNode) mémorise :

— l'`axis` (0 ou 1),

— la valeur pivot, c'est-à-dire le `split` (issu de la transformation du point médian en `CompositePoint`),

— et ses références aux sous-arbres `left` et `right`.

7.3 Étape 3 : Traitement des requêtes

Le programme doit être capable de traiter des requêtes SQL basiques, incluant des conditions sur des critères numériques, par exemple :

```
SELECT year FROM P WHERE courses >= 5
SELECT year, courses FROM P WHERE courses >= 5 AND year IN [2,3]
```

Algorithme de recherche dans le Kd-tree :

1. **Démarrage à la racine :** La recherche commence à la racine du Kd-tree.
2. **Parcours récursif :** À chaque nœud, les étapes suivantes sont réalisées :
   — **Vérification de la région :** Déterminer si la région associée au nœud intersecte la plage de la requête pour l'axe de division.
   — **Élagage des branches :**
      — Si la région est complètement en dehors de la plage, on n'explore pas ce sous-arbre.
      — Si la région est entièrement contenue dans la plage, on rapporte tous les points du sous-arbre sans vérification supplémentaire.
      — Si la région intersecte partiellement la plage, on poursuit la recherche récursive dans les sous-arbres gauche et droit.
   — **Vérification des feuilles :** Quand des feuilles sont atteintes, on vérifie si les points satisfont les conditions de la requête et on les ajoute aux résultats le cas échéant.
3. **Compilation des résultats :** Les informations demandées par la requête sont collectées et retournées à l'utilisateur.

Complexité de la recherche : La complexité dépend du nombre de points rapportés (k). Pour un Kd-tree équilibré en 2D, on obtient souvent $O(\sqrt{n} + k)$.

Détails de l'implémentation du traitement des requêtes Pour illustrer la recherche d'intervalle, considérons une requête sur un ensemble de points P avec les conditions numériques suivantes :

$$R = [2, 5] \times [3, 7].$$

Autrement dit, nous souhaitons sélectionner tous les points dont x est compris entre 2 et 5 et y entre 3 et 7.

1. Transformation des bornes en espace composite Le code transforme à la fois les points et les bornes de la requête dans un espace « composite » afin de gérer correctement les cas où plusieurs points partagent une même valeur pour un axe donné.

— **Pour chaque point** (p_x, p_y) :

$$\texttt{compositeKey}\Big((p_x, p_y), \text{axis}\Big)$$

construit un *CompositePoint* dont la partie *primary* et *secondary* changent selon l'axe (0 pour x, 1 pour y).

— **Pour les bornes de la requête** :

$$\texttt{createCompositeBound}(v, \text{borneInf})$$

crée un *CompositePoint* où la première coordonnée est fixée à v et la seconde à $\pm\infty$ selon qu'il s'agit d'une borne inférieure ou supérieure.

Ainsi, $[2, 5]$ pour x devient $\Big[(2 \mid -\infty), (5 \mid +\infty)\Big]$, et de même pour $[3, 7]$ sur y.

2. Parcours récursif du Kd-tree et calcul des régions Le parcours commence via la méthode

$$\texttt{searchKDTreeCompositeNodes}(\text{root}, \dots).$$

On part d'une région initiale $(-\infty, +\infty) \times (-\infty, +\infty)$. À chaque nœud interne :

1. On décide de l'axe `axis` (0 ou 1), mémorisé dans le nœud.
2. On met à jour la borne supérieure ou inférieure pour cet axe avec `node.getSplit()`, ce qui découpe la région en deux sous-régions (gauche et droite).
3. **Élagage** :
   — `regionFullyContained` : si la sous-région est entièrement dans R, on ajoute directement tous les points de ce sous-arbre.
   — `regionIntersects` : si la sous-région intersecte partiellement R, on continue la recherche récursive.
   — Sinon, on n'explore pas ce sous-arbre (région hors de R).
4. **Feuilles** : si on atteint un nœud feuille (qui contient effectivement un point), on vérifie s'il satisfait l'intervalle R.

La méthode renvoie enfin la liste de tous les points trouvés.

Exemple complet : requête SQL Prenons cinq points en 2D (plus d'éventuels critères texte non représentés) :

$$p_1 = (2, 3), \quad p_2 = (5, 4), \quad p_3 = (9, 6), \quad p_4 = (4, 7), \quad p_5 = (8, 1).$$

Requête SQL et interprétation des résultats Pour récupérer les points satisfaisant $x \in [2, 5]$ et $y \in [3, 7]$, le code :

```
SELECT x, y FROM P WHERE x in [2,5] AND y in [3,7];
```

sera converti dans l'espace composite en :

$$R' = \Big[(2 \mid -\infty), (5 \mid +\infty)\Big] \times \Big[(3 \mid -\infty), (7 \mid +\infty)\Big].$$

La méthode `rangeSearchCompositeNodes` vérifiera récursivement chaque branche du Kd-tree :

— *Élagage* si la région d'un sous-arbre est hors de R'.

— *Ajout direct* si la région est entièrement dans R'.

— *Descente récursive* sinon, jusqu'aux feuilles.

Tous les points respectant $2 \leq x \leq 5$ et $3 \leq y \leq 7$ sont alors renvoyés. Dans notre exemple, cela inclut notamment $p_1(2, 3)$, $p_2(5, 4)$ et éventuellement $p_4(4, 7)$ (car on considère l'intervalle inclusif).

Conclusion : Grâce à la transformation en espace composite et à l'algorithme de partitionnement décrit, le code construit effectivement un arbre où le *pivot* de chaque niveau réapparaît souvent dans le sous-arbre droit comme feuille. Bien que ce comportement soit inhabituel dans certains tutoriaux sur les Kd-trees, il fonctionne correctement pour les requêtes d'intervalle et répond aux besoins du projet. Les bornes composites permettent en outre de gérer précisément l'égalité sur un axe.

7.4 Étape 4 : Gestion des cas particuliers (Points avec coordonnées identiques)

Application dans l'algorithme :

1. **Tri des listes :** Les listes `Px` et `Py` sont triées en utilisant les coordonnées composites.
2. **Comparaisons :** Lors du partitionnement et des comparaisons, les coordonnées composites sont utilisées pour assurer un ordre total.
3. **Transformation des requêtes :** Les plages de requêtes sont adaptées pour correspondre à l'espace des nombres composites.

Logique et rôle de `compositeKey` : Dans le code, la méthode `compositeKey(point, axis)` est la clé pour gérer la présence éventuelle de valeurs identiques sur l'axe de coupe. L'implémentation procède ainsi :

1. **Création de deux tableaux : primary et secondary.** Pour un point (x, y), si l'axe est 0, alors

   $$\texttt{primary} = (x,\, y), \quad \texttt{secondary} = (y,\, x).$$

   Et si l'axe est 1, on inverse le rôle de x et y. On obtient ainsi un "ordre principal" et un "ordre secondaire" pour chaque point.

2. **Comparaison lexicographique dans `CompositePoint`.** La classe `CompositePoint` compare d'abord toutes les valeurs de `primary` ; en cas d'égalité, elle compare `secondary`. Cela garantit un *ordre total* : deux points (x_1, y_1) et (x_2, y_2) ne peuvent jamais être considérés comme "égaux" si l'un des deux axes diffère.
3. **Pourquoi c'est crucial ?** Si plusieurs points ont la même valeur de x (ou de y), l'algorithme a besoin d'un moyen pour les distinguer. Sinon, on risquerait de ne pas savoir dans quel sous-arbre les placer. Ici, `compositeKey` apporte un "tie-break" fiable, prévenant tout blocage ou arbre dégénéré.
4. **Au moment des requêtes par intervalle.** Dans `rangeSearchCompositeNodes`, les bornes $[x_{\min}, x_{\max}]$ et $[y_{\min}, y_{\max}]$ sont elles aussi converties en coordonnées composites (avec $\pm\infty$ pour la partie secondaire). Comparer un point à $(x_{\min} \mid -\infty)$ revient à tester $x \geq x_{\min}$ (et si $x = x_{\min}$, alors on tranche avec y). C'est essentiel pour décider l'inclusion ou l'exclusion d'un sous-arbre.

7.5 Étape 5 : Fonctions supplémentaires

Le programme supporte également plusieurs fonctionnalités additionnelles qui permettent de gérer l'échantillon et l'arbre Kd-tree de façon complète. Ces fonctionnalités sont implémentées principalement dans la classe `KdTreeManager` et s'appuient sur les méthodes de la classe `KdTree`. Voici comment se fait, en détail, le chargement d'un échantillon :

Chargement d'un échantillon

Pour charger un échantillon, la méthode `\loadSample(Scanner scanner)` de `KdTreeManager` demande à l'utilisateur le nom d'un fichier. Ce fichier doit respecter un format précis, et le processus se déroule comme suit :

Lecture du format du fichier

— La première ligne contient le nombre total de critères (M). Ce nombre détermine combien de lignes suivantes contiendront les noms des critères.

— Les M lignes suivantes contiennent chacune un nom de critère. Les deux premiers critères correspondent aux dimensions numériques (par exemple, x et y) qui serviront à la construction du Kd-tree, tandis que les éventuels critères supplémentaires fourniront des informations complémentaires pour chaque point.

— La ligne suivante indique le nombre de points (N).

— Enfin, les N lignes suivantes décrivent chaque point. Chaque ligne commence par deux valeurs numériques correspondant aux coordonnées x et y, suivies, le cas échéant, de valeurs pour les critères supplémentaires.

Lecture et initialisation des données

— La méthode `loadFromFile(String filename)` de la classe `KdTree` ouvre le fichier (en utilisant par exemple un `BufferedReader`) et lit la première ligne pour obtenir

M. Si $M < 2$, une exception est levée car au moins deux critères sont nécessaires pour définir les dimensions numériques.

— Ensuite, les M lignes suivantes sont lues et stockées dans le tableau `criteriaNames`. Ce tableau est utilisé par la suite pour nommer les colonnes et pour effectuer les comparaisons lors de la construction de l'arbre.

— La ligne indiquant le nombre de points N est ensuite lue. Puis, pour chaque point, le fichier est lu ligne par ligne. Chaque ligne est découpée par les espaces : les deux premières valeurs sont converties en `double` et mises dans un tableau représentant les coordonnées numériques, et les valeurs restantes (si présentes) sont stockées dans un tableau de `String` correspondant aux critères supplémentaires.

— Chaque point est ajouté à la liste interne `samplePoints` sous forme d'une paire associant les coordonnées numériques et les critères supplémentaires.

Construction de l'arbre

— Une fois l'ensemble des points chargé dans `samplePoints`, la méthode `loadFromFile` termine sa lecture.

— La méthode `rebuildTree()` est alors appelée. Celle-ci effectue un tri des points selon la dimension x (axe 0) et la dimension y (axe 1) en utilisant l'algorithme `HeapSort` et la fonction `compositeKey`.

— Ensuite, la méthode `buildTree(...)` est appelée de manière récursive pour construire le Kd-tree en partitionnant les points en sous-groupes, jusqu'à atteindre le cas de base (feuille avec un seul point).

— La liste `nodeList` est également mise à jour pour contenir l'ensemble des feuilles de l'arbre.

Sauvegarde de l'échantillon

La méthode `saveSample(Scanner scanner)` de la classe `KdTreeManager` permet d'exporter l'échantillon actuel dans un fichier texte. Le processus se déroule comme suit :

Vérification préalable

Avant de lancer l'opération de sauvegarde, la méthode vérifie que l'arbre Kd-tree n'est pas vide en appelant la fonction `isEmpty()` de la classe `KdTree`. Si l'échantillon est vide, une exception est levée et la sauvegarde est interrompue.

Saisie du nom de fichier

L'utilisateur est invité à saisir le nom du fichier où l'échantillon doit être sauvegardé. Ce nom est récupéré à l'aide d'un `Scanner` dans la méthode `saveSample(Scanner scanner)`.

Préparation du répertoire de sauvegarde

La méthode `saveToFile(String filename)` de la classe `KdTree` détermine le chemin absolu du projet, puis construit le chemin vers le répertoire `src/main/resources`.

— Si le répertoire n'existe pas, le programme tente de le créer.
— En cas d'échec de la création du répertoire, une `IOException` est levée.

Écriture du fichier

Un `BufferedWriter` est utilisé pour écrire dans le fichier. La méthode `writeSample(BufferedWriter bw)` est alors appelée pour écrire l'échantillon en respectant le format suivant :

1. **Nombre total de critères :** La première ligne du fichier contient le nombre total de critères (M).
2. **Noms des critères :** Les M lignes suivantes contiennent chacune le nom d'un critère. Les deux premières lignes correspondent aux dimensions numériques (par exemple, x et y), et les lignes suivantes (si elles existent) correspondent aux critères supplémentaires.
3. **Nombre de points :** La ligne suivante indique le nombre de points (N).
4. **Données des points :** Pour chaque point, une ligne est écrite contenant :
   — Les deux premières valeurs numériques représentant les coordonnées x et y.
   — Le cas échéant, les valeurs des critères supplémentaires, séparées par un espace.

Ce format garantit que le fichier sauvegardé sera structuré de manière cohérente, ce qui permettra de le recharger ultérieurement sans ambiguïté.

Ajout de points

L'ajout de nouveaux points se fait via la méthode `addPoint(Scanner scanner)` de la classe `KdTreeManager`. Le processus se déroule en plusieurs étapes détaillées ci-dessous :

Saisie des données par l'utilisateur

L'utilisateur est invité à saisir :
— Les coordonnées numériques x et y (les deux premières dimensions du Kd-tree).
— Les valeurs des critères supplémentaires, si le nombre total de critères M est strictement supérieur à 2. Ces valeurs correspondent aux informations additionnelles associées à chaque point et sont lues pour chacun des $M - 2$ critères supplémentaires.

Cette saisie se fait par le biais de boucles qui demandent la conversion des entrées en nombres (pour x et y) et la lecture de chaînes de caractères pour les autres critères.

Vérification de l'unicité du point

Avant d'ajouter le nouveau point à l'échantillon, le programme vérifie que ce point n'existe pas déjà dans la liste `samplePoints`.

— La comparaison se fait en examinant les deux premières coordonnées (x et y) de chaque point déjà présent.
— Si un point avec les mêmes coordonnées x et y est détecté, une exception est levée (en utilisant par exemple une `IllegalArgumentException`), ce qui empêche l'insertion de doublons.

Ajout du point et reconstruction de l'arbre

Une fois la validation effectuée, la méthode `addPointAndRebuild(double[] point, String[] extraCriteria)` de la classe `KdTree` est appelée. Cette méthode :

- Ajoute le nouveau point (associé à ses critères supplémentaires) à la liste interne `samplePoints`.
- Appelle la méthode `rebuildTree()` pour reconstruire l'intégralité du Kd-tree à partir de la liste mise à jour. Cela implique :
  - Un tri des points selon la dimension x et selon la dimension y à l'aide de l'algorithme `HeapSort`.
  - La reconstruction récursive de l'arbre via la méthode `buildTree(...)` qui partitionne les points en utilisant la fonction `compositeKey` pour définir les clés de séparation.

Affichage du Kd-tree

La représentation visuelle du Kd-tree est obtenue grâce à la méthode `printTree()` de la classe `KdTree`, appelée via `printTree()` de `KdTreeManager`. Ce mécanisme fonctionne comme suit :

Génération de la représentation en ASCII-art

La méthode `printTree()` vérifie d'abord que l'arbre n'est pas vide, puis appelle une fonction récursive, `buildTreeString()`, pour parcourir l'arbre et générer une chaîne de caractères pour chaque nœud. Cette chaîne représente la hiérarchie de l'arbre en utilisant des symboles comme └──, ┌── ou │ pour indiquer les relations de parenté entre les nœuds.

Fonction récursive `buildTreeString()`

Cette fonction est chargée de parcourir l'arbre et de construire les lignes de la représentation textuelle. Son fonctionnement détaillé est le suivant :

1. **Parcours du sous-arbre droit :** Si le nœud courant possède un sous-arbre droit, `buildTreeString()` est appelée récursivement sur ce sous-arbre avec un préfixe d'indentation adapté (par exemple, en ajoutant │ ou). Cela permet d'afficher d'abord les nœuds situés à droite.
2. **Construction de la ligne de représentation :** Pour le nœud courant, la fonction construit une chaîne de caractères qui dépend de son type :
   - **Nœud interne :** Si le nœud ne contient pas de point (c'est-à-dire qu'il s'agit d'un nœud interne), la fonction affiche la valeur de séparation sous la forme « Split x = 5 » ou « Split y = 7 », en fonction de l'axe de séparation stocké dans le nœud.
   - **Feuille :** Si le nœud est une feuille (c'est-à-dire qu'il contient un point), la chaîne affichée présente directement les coordonnées du point ainsi que les critères supplémentaires associés.
3. **Parcours du sous-arbre gauche :** Si le nœud courant possède un sous-arbre gauche, la fonction est appelée récursivement sur ce sous-arbre, en ajustant de nouveau le préfixe d'indentation pour refléter correctement la structure hiérarchique.

Affichage final dans la console

Une fois que toutes les lignes ont été générées et stockées dans une liste, la méthode `printTree()` parcourt cette liste et imprime chaque ligne dans la console. Le résultat est une représentation en ASCII-art qui permet de visualiser clairement la structure du Kd-tree, facilitant ainsi la compréhension et le débogage de l'arbre.

Calcul des valeurs extrêmes (min / max)

Pour déterminer le point ayant la valeur minimale ou maximale sur une dimension donnée (*dim*, par exemple 0 pour x ou 1 pour y), la classe `KdTree` fournit deux méthodes optimisées :

— `findMin(dim)` – Cette méthode s'appuie sur la fonction récursive optimisée `findMinRecOptimized(node, dim)` pour identifier le point minimum.

— `findMax(dim)` – Cette méthode utilise la fonction récursive optimisée `findMaxRecOptimized(node, dim)` pour trouver le point maximum.

Principe d'élagage en fonction de l'axe

Dans un kd-tree, chaque nœud interne possède un *axe de séparation* (accessible via `node.getAxis()`) qui détermine la dimension selon laquelle l'espace est divisé (par exemple, 0 pour x ou 1 pour y). Cet axe permet d'orienter la recherche de la valeur extrême :

— **Recherche du minimum (`findMinRecOptimized`) :**

1. **Nœud feuille :** Si `node.getPoint()` $\neq$ *null*, le nœud est une feuille et contient directement un point. Ce point est retourné immédiatement.
2. **Nœud interne avec `node.getAxis()` = *dim* :** Lorsque l'axe de séparation du nœud est égal à la dimension recherchée, le pivot (défini dans `node.getSplit()`) est utilisé pour partitionner l'espace. Dans ce cas, tous les points du sous-arbre droit sont supposés avoir une valeur plus élevée sur l'axe *dim* que le pivot. Ainsi, seule l'exploration du sous-arbre gauche et la comparaison avec le pivot permettent de trouver la valeur minimale.
3. **Nœud interne avec `node.getAxis()` $\neq$ *dim* :** Dans ce cas, il n'est pas possible d'élaguer un sous-arbre entier. La recherche se fait alors dans les deux sous-arbres (gauche et droit), puis on compare leurs minimums respectifs avec le pivot pour déterminer le candidat minimal.

— **Recherche du maximum (`findMaxRecOptimized`) :**

1. **Nœud feuille :** Comme pour le minimum, si `node.getPoint()` $\neq$ *null*, le point contenu dans le nœud est retourné directement.
2. **Nœud interne avec `node.getAxis()` = *dim* :** Ici, le pivot et le sous-arbre droit sont examinés, car tous les points du sous-arbre gauche devraient avoir des valeurs inférieures sur la dimension *dim* par rapport au pivot. La recherche se limite donc au sous-arbre droit et à la comparaison avec le pivot.
3. **Nœud interne avec `node.getAxis()` $\neq$ *dim* :** On explore les deux sous-arbres, puis on compare leurs maximums avec le pivot pour déterminer le point maximal.

Comparaison via `compositeKey`

Pour comparer deux points, par exemple (x_1, y_1) et (x_2, y_2), la méthode `compositeKey` est utilisée avec l'argument `axis = dim`. Cela signifie que la comparaison porte d'abord sur la coordonnée correspondant à *dim* (la coordonnée principale) et, en cas d'égalité, sur l'autre coordonnée (la secondaire).

Complexité

— **Cas favorable :** Dans un kd-tree équilibré, lorsque l'axe du nœud correspond à la dimension recherchée, l'élagage permet d'ignorer systématiquement un des sous-arbres. Dans ce cas, la recherche se rapproche d'une complexité de $O(\log n)$.

— **Cas défavorable :** Si l'arbre est déséquilibré ou si la répartition des données ne permet pas d'élaguer efficacement les branches, il peut être nécessaire d'explorer la quasi-totalité des nœuds, ce qui conduit à une complexité maximale de $O(n)$, où n représente le nombre total de points.

8 Construction d'un Kd-tree en trois dimensions : Exemple complet pas à pas

Note pédagogique – non implémenté. L'arbre 3D présenté ci-dessous illustre la généralisation théorique d'un Kd-tree ; *le code fourni reste limité à deux dimensions.*

Dans cette section, nous illustrons la **généralisation des Kd-trees en 3D** à partir de l'exemple donné dans l'énoncé (Section 3.3). Nous disposons de 15 points, chacun caractérisé par

$$(\texttt{year}, \texttt{height}, \texttt{courses}, \texttt{name}),$$

dont **seuls les trois premiers** (`year`, `height`, `courses`) sont utilisés comme dimensions numériques du Kd-tree. À chaque profondeur d, on choisit l'axe (d mod 3) :

$$0 \mapsto \texttt{year}, \quad 1 \mapsto \texttt{height}, \quad 2 \mapsto \texttt{courses}.$$

Convention de partition :

$$\begin{cases} \text{sous-arbre gauche} = \{\, p \mid \text{coord}(p) < \text{pivot} \}, \\ \text{sous-arbre droit} = \{\, p \mid \text{coord}(p) \geq \text{pivot} \}. \end{cases}$$

Le pivot (médiane) se retrouve donc dans le *sous-arbre droit*.

Rappel de l'exemple

Les 15 points (pour mémoire) sont :

year	height	courses	name
3	161.50	7	Alexandra
5	175.00	11	Bob
2	163.75	9	Charlotte
4	178.20	13	David
1	168.90	6	Emma
3	172.30	8	Frank
4	166.70	10	Grace
2	183.40	12	Henry
5	162.80	5	Irene
1	170.20	13	Jack
4	172.60	8	Kate
3	175.10	9	Liam
2	165.00	10	Mathieu
5	180.90	4	Noah
1	168.30	10	Olivia

Axes en 3D :

- Profondeur 0 : `year` (*axe 0*).
- Profondeur 1 : `height` (*axe 1*).
- Profondeur 2 : `courses` (*axe 2*).
- Profondeur 3 : retour à `year` (*axe 0*), etc.

Le *pivot* à chaque niveau est la *médiane* (selon l'axe courant) dans l'ensemble de points considéré.

Profondeur 0 : Séparation selon `year` (axe = 0)

1. **Trier par `year`.** On classe les 15 points en ordre croissant de `year`.
2. **Médiane (pivot).** Supposons qu'avec 15 éléments, la médiane se trouve être `(3,172.30,8,Frank)`. Ainsi, pivot = 3.
3. **Sous-arbres** :

$$\text{sous-arbre gauche} = \{\, p \mid \texttt{year}(p) < 3\}, \quad \text{sous-arbre droit} = \{\, p \mid \texttt{year}(p) \geq 3\}.$$

Au *nœud racine*, nous stockons donc : $(\text{axe} = 0, \text{split} = 3)$. Le pivot (Frank) se retrouve dans le *sous-arbre droit* par la règle "$\geq$ pivot".

Profondeur 1 : Séparation selon `height` (axe = 1)

Branche gauche : ce sont tous les points dont `year` < 3. On les trie par `height`, puis on prend la médiane (split). Tout point dont `height` $<$ (pivot) ira à gauche, tout point dont `height` $\geq$ (pivot) ira à droite.
Branche droite : ce sont tous les points dont `year` ≥ 3. On les trie par `height` de la même façon, prend la médiane, etc.

Profondeur 2 : Séparation selon `courses` (axe = 2)

Pour chacun des sous-ensembles restants, on trie selon `courses`. On sépare `courses` $<$ pivot (gauche) / `courses` $\geq$ pivot (droit). On poursuit la récursion jusqu'à ne plus avoir qu'un seul point par sous-ensemble, faisant de celui-ci une *feuille*.

Structure finale du Kd-tree 3D

En fin de parcours, on obtient un **Kd-tree en 3 dimensions** où :

— Chaque *nœud interne* mémorise :

$$(\text{axe} \in \{0,1,2\},\ \text{split} = (\text{valeur pivot}),\ \text{sous-arbre gauche},\ \text{sous-arbre droit}).$$

— Chaque *feuille* contient un unique point $(year, height, courses, name)$.

Le pivot est toujours *dans* le sous-arbre droit, conformément à la règle coord $\geq$ pivot. Chacun des 15 points aboutit donc dans une feuille distincte.

Exemple d'une forme finale (schématique) :

$$(\text{axe} = 0, \text{split} = 3)\ :\quad \begin{cases} \text{Gauche} : (\text{axe} = 1, \text{split} = 168.30) \\ \quad \begin{cases} \text{Gauche} : (\text{axe} = 2, \text{split} = 9) \ \to \dots \\ \text{Droit} : (\text{axe} = 2, \text{split} = 10) \ \to \dots \end{cases} \\ \text{Droit} : (\text{axe} = 1, \text{split} = 170.00) \ \to \dots \end{cases}$$

et ainsi de suite, jusqu'à 15 feuilles.

Complexité & conclusion

D'après la référence :

— **Construction** : $O(n \log n)$ si on présort l'ensemble en `year`, `height`, `courses`.

— **Espace** : $O(n)$, car l'arbre est binaire avec 15 feuilles.

— **Requêtes orthogonales 3D** : le temps se majorant par $O(n^{1-\frac{1}{3}} + k)$, soit $O(n^{2/3} + k)$.

Le **Kd-tree 3D** décrit dans cet exemple est donc une simple extension du principe 2D : on alterne les *3* axes $(year, height, courses)$ et on applique la convention "coord $<$ pivot" à gauche, "coord $\geq$ pivot" à droite.

9 Diagramme de classes

Pour mieux appréhender l'architecture logicielle gérant l'implémentation du Kd-tree et ses diverses fonctionnalités, il est utile de s'appuyer sur un **diagramme de classes UML**. Celui-ci offre une représentation visuelle des entités principales du projet (par exemple, les classes `KdTree`, `KdTreeNode`, `CompositePoint`, `KdTreeManager`), de leurs attributs et méthodes, ainsi que des liens unissant ces entités (associations, héritages, dépendances, etc.).

Dans notre code, on retrouve notamment :

— `KdTreeManager`, qui coordonne les opérations de chargement, sauvegarde et gestion de l'arbre ;

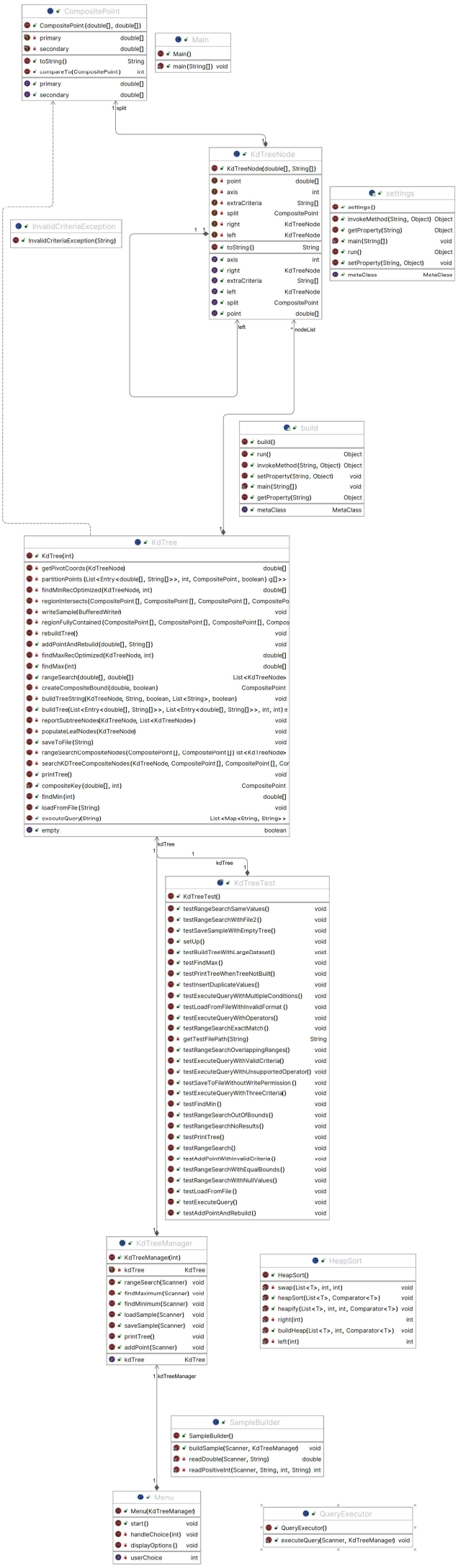
— `KdTree`, qui construit et manipule la structure du Kd-tree (noeuds internes, feuilles, etc.) ;

— `KdTreeNode`, qui représente chaque noeud (interne ou feuille) de l'arbre ;

— `CompositePoint`, utilisé pour créer des clés de comparaison (primary /secondary) et gérer les axes de séparation ;

— Plusieurs classes utilitaires comme `HeapSort` (pour le tri), `Menu` (pour l'interface console), ou `QueryExecutor` (pour interpréter des requêtes SQL simples).

Grâce à ce diagramme, on perçoit rapidement la répartition des responsabilités dans le code et la façon dont les différentes classes collaborent pour fournir des opérations telles que la *construction* du Kd-tree, la *recherche par plage*, l'*ajout dynamique* de points ou la *visualisation* de la structure. En plus de faciliter la lecture et la maintenance de l'application, ce diagramme constitue un atout pédagogique pour toute personne qui découvrirait ou reprendrait le projet.


CompositePoint
CompositePoint(double[], double[])
primary double[]
secondary double[]
toString() String
compareTo(CompositePoint) int
primary double[]
secondary double[]
Main
Main()
main(String[]) void
1 split
KdTreeNode
KdTreeNode(double[], String[])
point double[]
axis int
extraCriteria String[]
split CompositePoint
right KdTreeNode
left KdTreeNode
toString() String
axis int
right KdTreeNode
extraCriteria String[]
left KdTreeNode
split CompositePoint
point double[]
settings
settings()
invokeMethod(String, Object) Object
getProperty(String) Object
main(String[]) void
run() Object
setProperty(String, Object) void
metaClass MetaClass
InvalidCriteriaException
InvalidCriteriaException(String)
1 1
left
* nodeList
build
build()
run() Object
invokeMethod(String, Object) Object
setProperty(String, Object) void
main(String[]) void
getProperty(String) Object
metaClass MetaClass
1
KdTree
KdTree(int)
getPivotCoords(KdTreeNode) double[]
partitionPoints(List<Entry<double[], String[]>>, int, CompositePoint, boolean) g[]>>
findMinRecOptimized(KdTreeNode, int) double[]
regionIntersects(CompositePoint[], CompositePoint[], CompositePoint[], CompositePo
writeSample(BufferedWriter) void
regionFullyContained(CompositePoint[], CompositePoint[], CompositePoint[], Compos
rebuildTree() void
addPointAndRebuild(double[], String[]) void
findMaxRecOptimized(KdTreeNode, int) double[]
findMax(int) double[]
rangeSearch(double[], double[]) List<KdTreeNode>
createCompositeBound(double, boolean) CompositePoint
buildTreeString(KdTreeNode, String, boolean, List<String>, boolean) void
buildTree(List<Entry<double[], String[]>>, List<Entry<double[], String[]>>, int, int) e
reportSubtreeNodes(KdTreeNode, List<KdTreeNode>) void
populateLeafNodes(KdTreeNode) void
saveToFile(String) void
rangeSearchCompositeNodes(CompositePoint[], CompositePoint[]) ist<KdTreeNode>
searchKDTreeCompositeNodes(KdTreeNode, CompositePoint[], CompositePoint[], Cor
printTree() void
compositeKey(double[], int) CompositePoint
findMin(int) double[]
loadFromFile(String) void
executeQuery(String) List<Map<String, String>>
empty boolean
kdTree
1
kdTree
KdTreeTest
KdTreeTest()
testRangeSearchSameValues() void
testRangeSearchWithFile2() void
testSaveSampleWithEmptyTree() void
setUp() void
testBuildTreeWithLargeDataset() void
testFindMax() void
testPrintTreeWhenTreeNotBuilt() void
testInsertDuplicateValues() void
testExecuteQueryWithMultipleConditions() void
testLoadFromFileWithInvalidFormat() void
testExecuteQueryWithOperators() void
testRangeSearchExactMatch() void
getTestFilePath(String) String
testRangeSearchOverlappingRanges() void
testExecuteQueryWithValidCriteria() void
testExecuteQueryWithUnsupportedOperator() void
testSaveToFileWithoutWritePermission() void
testExecuteQueryWithThreeCriteria() void
testFindMin() void
testRangeSearchOutOfBounds() void
testRangeSearchNoResults() void
testPrintTree() void
testRangeSearch() void
testAddPointWithInvalidCriteria() void
testRangeSearchWithEqualBounds() void
testRangeSearchWithNullValues() void
testLoadFromFile() void
testExecuteQuery() void
testAddPointAndRebuild() void
KdTreeManager
KdTreeManager(int)
kdTree KdTree
rangeSearch(Scanner) void
findMaximum(Scanner) void
findMinimum(Scanner) void
loadSample(Scanner) void
saveSample(Scanner) void
printTree() void
addPoint(Scanner) void
kdTree KdTree
HeapSort
HeapSort()
swap(List<T>, int, int) void
heapSort(List<T>, Comparator<T>) void
heapify(List<T>, int, int, Comparator<T>) void
right(int) int
buildHeap(List<T>, int, Comparator<T>) void
left(int) int
1 kdTreeManager
SampleBuilder
SampleBuilder()
buildSample(Scanner, KdTreeManager) void
readDouble(Scanner, String) double
readPositiveInt(Scanner, String, int, String) int
1
Menu
Menu(KdTreeManager)
start() void
handleChoice(int) void
displayOptions() void
userChoice int
QueryExecutor
QueryExecutor()
executeQuery(Scanner, KdTreeManager) void

10 Mini-guide d'utilisation sous IntelliJ

1. Importer le projet

— Ouvrir IntelliJ et choisir `Open` (ou `Import Project`).

— Sélectionner le dossier racine du projet (contenant `pom.xml` ou `build.gradle`).

— Valider l'import (IntelliJ gère seul les dépendances).

2. Compiler

— Vérifier le `Project SDK` dans `File > Project Structure`.

— Sous Maven : exécuter `mvn clean install` (ou `package`) dans l'onglet `Maven Projects`.

— Sous Gradle : exécuter `clean` puis `build` dans l'onglet `Gradle`.

— En projet Java simple : `Build > Build Project`.

3. Exécuter l'application

— Localiser la classe contenant la méthode `main`.

— Dans `Run > Edit Configurations...`, créer une configuration `Application`.

— Sélectionner la classe principale (avec `main`) et cliquer `Run`.

4. Utilisation

— Au lancement, un menu peut s'afficher (chargement/sauvegarde de données, ajout de points, requêtes, etc.).

— Saisir les chemins de fichier et les commandes selon l'interface proposée.

— Les requêtes SQL simples (`SELECT ... WHERE ...`) sont exécutées via le Kd-tree.

5. Conseils rapides

— Si un fichier n'est pas trouvé, vérifier le chemin (absolu/relatif).

— En cas de doublon ou d'erreur lors de l'ajout d'un point, se référer au format attendu (x, y, critères supplémentaires).

— Pour sauvegarder, suivre le format d'écriture (même structure que pour le chargement).

11 Conclusion et Remerciements

En définitive, ce projet autour de l'implémentation d'un **Kd-tree** illustre toute la richesse qu'une structure de données arborescente peut apporter en termes d'efficacité pour la *recherche de plages multidimensionnelles*. J'y ai successivement abordé la **construction naïve**, puis la **construction optimisée** (basée sur un présortage global), montré comment gérer les **cas particuliers** de coordonnées identiques et, enfin, étudié la **généralisation** aux trois dimensions. Ces travaux se sont concrétisés par la mise en place d'un programme Java capable :

— de **charger et sauvegarder** un échantillon de données, en suivant un format texte adapté ;

— d'**ajouter dynamiquement** de nouveaux points (avec reconstruction automatique de la structure) ;

— d'**effectuer des requêtes SQL** simples sur les deux premiers critères numériques, notamment grâce à l'algorithme de *range search* dans le Kd-tree ;

— d'**afficher visuellement** la structure arborescente en ASCII-art et de fournir des réponses sur la valeur minimale ou maximale d'un critère donné.

Sur le plan algorithmique, le projet a mis en évidence la **complexité maîtrisée** du Kd-tree (construction en $O(n \log n)$, recherches en $O(\sqrt{n}+k)$ en 2D) et son **adaptabilité** à la présence de points à coordonnées identiques via l'espace des *nombres composites*. De plus, la **généralisation en 3D** montre comment alternent les trois axes (`year`, `height`, `courses`) et ouvre la voie à des dimensions supplémentaires, au prix d'une analyse plus délicate des performances.

Remerciements. Je tiens à exprimer toute ma gratitude à **Madame la Professeure Véronique BRUYERE** pour ses conseils pédagogiques et son accompagnement tout au long de ce projet, ainsi qu'à **Monsieur l'Assistant Christophe GRANDMONT** pour son aide technique, ses retours constructifs et sa disponibilité. Leurs orientations m'ont permis de structurer efficacement ma démarche et d'approfondir ma compréhension des Kd-trees, tant du point de vue théorique que pratique.

Dans l'ensemble, ce travail confirme à quel point les **structures de données géométriques** – et le Kd-tree en particulier – sont des briques essentielles pour la conception de systèmes de gestion de données multidimensionnelles. La mise en œuvre décrite dans ce rapport constitue donc une base solide pour de futurs développements, qu'il s'agisse d'extensions fonctionnelles (gestion de requêtes plus complexes) ou d'optimisations supplémentaires (balancement dynamique, prise en compte de dimensions plus nombreuses, etc.).

12 Références

Références

[1] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry : Algorithms and Applications*. 2nd Edition, Springer, 2000. **(La référence la plus utilisée dans ce rapport)**

[2] Jon L. Bentley. *Multidimensional binary search trees used for associative searching*. Communications of the ACM, 18(9) :509–517, 1975.

[3] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. *An algorithm for finding best matches in logarithmic expected time*. ACM Transactions on Mathematical Software (TOMS) 3.3 (1977) : 209–226.

[4] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[5] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. *An optimal algorithm for approximate nearest neighbor searching fixed dimensions*. Journal of the ACM (JACM), 45(6) :891–923, 1998.

[6] Andrew W. Moore. *An introductory tutorial on kd-trees*. Technical report, Robotics Institute, Carnegie Mellon University, 1991.

[7] *Kd-tree*. Wikipedia, The Free Encyclopedia. Available at : https://en.wikipedia.org/wiki/K-d_tree.

[8] Google Search. Available at : https://www.google.com. Used as a tool to find various academic papers and articles on Kd-Trees.