UMONS

Structrure de données II
Windowing

Rapport du projet de Structures de données II

Élèves :
Gatta Nicolas
Vancauwenberghe Emilien

Enseignant :
Bruyère Véronique

7 avril 2023

Table des matières

1 Introduction

Dans le carde du cours de structure de données II, nous avons été amenés à réaliser un projet dans le langage Java accompagné d'une interface graphique. Ce projet a pour but final de créer et manipuler des structures de données pour ainsi les appliquer tout au long de la réalisation du projet. L'une structure de données les plus importantes dans ce projet sont les arbres de recherche à priorité (Priority Search Tree abrégé "PST").

L'objectif d'un PST est de pouvoir appliquer un "windowing" sur un ensemble de données. Le windowing est une technique utilisée en informatique graphique et en traitement d'images pour sélectionner et afficher une partie d'un ensemble de points ou d'une ligne brisée. Cette technique est souvent utilisée lorsque l'ensemble des données à afficher est trop grand pour tenir sur l'écran ou pour être traité efficacement. On peut dès lors retrouver l'utilisation de cette technologie dans les systèmes d'affichage de carte se trouvant dans les GPS ou encore sur Google Maps.

Il est intéressant de remarquer que le PST partage beaucoup de caractéristiques communes avec les tas ainsi que les arbres binaires de recherches(ABR), celles-ci ayant été étudiés durant les cours de structure de données II.

2 Présentation du problème

Pour ce projet, il est demandé d'écrire un programme accompagné d'une interface graphique permettant d'appliquer un windowing à un ensemble de segments verticaux et horizontaux dans $\mathbb{R}^2$. Il sera alors possible d'affiner les données pour y afficher les segments respectant les conditions. On peut alors retrouver les affinages suivants :

- $[X, X']$ x $[Y, Y']$
- $[-\infty, X']$ x $[Y, Y']$
- $[X, +\infty]$ x $[Y, Y']$
- $[X, X']$ x $[-\infty, Y']$
- $[X, X']$ x $[Y, +\infty]$

3 Le tas

3.1 Définition

Un tas est une structure de données arborescente où chaque nœud a une valeur supérieure ou égale à ses nœuds enfants. En d'autres termes, le nœud racine a la plus grande valeur de tous les nœuds de l'arbre.

Les tas sont utilisés pour implémenter des structures de données telles que des files d'attente prioritaires pour stocker et récupérer des éléments avec une priorité donnée. Les tas sont également utilisés pour trier les éléments par ordre croissant ou décroissant.

3.2 Utilisation dans le cadre du projet

Dans le cadre de ce projet, le tas a été utilisé pour trier les données avec la complexité la plus faible possible. Il est cependant intéressant de remarquer que la fonction de base de java Arrays.sort() permet de faire un tri en une complexité presque similaire à celle d'un tas. Le tas aura une tendance à être plus proche d'une complexité linéaire que le Arrays.sort() pour des grands ensemble de données.

3.3 Méthode de construction

Pour la construction du tas dans ce projet, nous avons utilisés heapify. L'algorithme Heapify prend en entrée un tableau d'éléments et un index en considérant le tableau d'élément comme un arbre binaire. Il commence par comparer la valeur de la racine avec celle de ses deux enfants, et s'il y a un enfant dont la valeur est plus grande, il échange la valeur de la racine avec celle de l'enfant plus grand. L'algorithme répète ce processus récursivement pour l'enfant échangé jusqu'à ce que la propriété de tas soit rétablie pour tout le sous-arbre. Par la suite, l'on pourra extraire les données une par une par ordre croissant sur les Y.

4 Priority Search Tree

4.1 Définition

Un arbre de recherche prioritaire (PST) est une structure de données arborescente utilisée pour stocker des éléments associés à une clé préférée. Les éléments sont stockés dans des nœuds d'arbre en fonction de leurs clés de priorité respectives, et PST permet une récupération rapide des éléments avec une priorité donnée.

Un PST se compose de nœuds, chacun ayant une clé de priorité et un sous-arbre gauche et droit. L'organisation des nœuds garantit que la clé de priorité du nœud de sous-arbre gauche est inférieure à la clé de priorité du nœud actuel, et la clé de priorité du nœud de sous-arbre droit est supérieure à la priorité du nœud actuel.

L'utilisation de PST permet la recherche, l'insertion et la suppression d'éléments en temps logarithmique sur le nombre total d'éléments stockés dans l'arborescence. PST est largement utilisé dans les applications de traitement de données telles que les systèmes de gestion de bases de données et les systèmes de planification de tâches.

4.2 Utilisation dans le cadre du projet

Dans ce projet, le PST sera utilisé pour stockés des segments composés de quatre coordonnées X, Y, X' et Y'. Chaque noeud du PST sera composé d'un segment lui même composé de deux points ainsi qu'une médiane.

On aura alors deux possibilités, si l'on considère uniquement les coordonnées X, le PST sera organisé tel un tas avec le minimum à la racine. Cependant, si l'on considère que les coordonnées Y, le PST sera organisé comme un ABR.

La formation de l'ABR dans ce cas est assez particulière vu que les fils sont triés par rapport à la médiane du noeud courant et les contraintes suivantes devront être respectées :

— le fils gauche aura sa coordonnée X plus grand que celle du noeud parent et sa coordonnée Y plus petite que la médiane du noeud parent.

— le fils droit aura sa coordonnée X plus grand que celle du noeud parent et sa coordonnée Y plus grande que la médiane du noeud parent.

4.3 Méthode de construction

On peut retrouver principalement deux méthodes de construction de PST : le bottom-up et le top-down.

La création d'un Prority Search Tree top-down débute par la division du plan en deux parties selon une ligne ou un axe. Chaque partie est ensuite subdivisée en deux autres parties, jusqu'à ce que chaque point soit associé à une feuille de l'arbre. Cette méthode est couramment utilisée pour construire des PST en deux dimensions.

La construction d'un Prority Search Tree bottom-up démarre par la création d'une structure de données initiale qui comprend tous les points. Les points sont ensuite répartis dans des nœuds de l'arbre selon leur position dans le plan. Les nœuds sont ensuite fusionnés en fonction de critères de division (tels que la division en deux parties égales) jusqu'à ce que chaque nœud ne contienne qu'un seul point ou un faible nombre de points. Cette méthode est souvent utilisée pour construire des PST en trois dimensions ou plus, là où la construction top-down est plus ardue à réaliser.

Pour ce projet, il a été décidé d'utiliser une construction top-down pour sa simplicité de mise en oeuvre comparer au bottom-up qui est considéré comme étant complexe à appliquer.

4.4 Initialisation de la construction

Avant d'entamer la construction complète de l'arbre nous effectuons un tri sur l'entièreté des segments.

L'ensemble des segments est initialement stocké sous forme d'une liste non triée sur laquelle nous effectuons une transformation en tas qui sera appliqué sur l'élément Y de la coordonnée du segment. Par la suite, on recherche le segment ayant la plus petite coordonnée X qui sera utilisé comme racine du PST. ensuite, l'élément qui est devenu le noeud racine de l'arbre est retiré du tas. Nous réutilisons ensuite ce tas lors de la génération complète de l'arbre.

4.5 Construction de l'arbre

Une fois la liste de segments transformée en tas et le noeud racine défini, nous pouvons créer les noeuds de l'arbre.

Nous calculons d'abord la médiane du noeud courant. Par exemple, si on considère que pour le moment nous n'avons que la racine de l'arbre déduis dans la phase d'initialisation, nous calculons donc la médiane de la racine en utilisant les segments contenu dans le tas qui n'ont pas encore été placé dans l'arbre. Une fois le segment médian défini, on récupère la valeur Y de la coordonnée de ce segment en l'introduisant comme valeur médiane du noeud courant.

Afin de définir les fils gauche et droit du noeud courant, nous coupons en 2 le tas à partir de l'élément médian trouvé précédemment. la sous liste gauche du tas constituera l'ensemble des descendants gauche du noeud courant. De même pour la sous liste droite qui contient les descendants droit du noeud courant.

Le segment qui deviendra le fils gauche du noeud actuel est le premier élément de la sous liste gauche du tas tandis que le fils droit sera le dernier élément de la sous liste droite du tas. Cette opération est répétée récursivement afin de créer les autres noeuds.

4.6 Windowing

Comme mentionné précédemment, le windowing consiste à sélectionner une partie plus ou moins grande des données d'un ensemble complet d'une collection. Ce qui signifie qu'au niveau de l'arbre à priorité il faudra élaguer les branches non désirées et ne garder ainsi que les noeuds contenant les segments que l'on souhaitent voir apparaître dans la fenêtre.

Il existe 3 cas d'interprétation des segments par la fenêtre :

1. Le segment est contenu entièrement dans la fenêtre.
2. Le segment a son point d'origine ou son point d'arrivée dans la fenêtre
3. Le segment traverse la fenêtre de part en part et a son point d'origine et d'arrivée en dehors de la fenêtre.

4.6.1 Types de fenêtres

Comme mentionné précédemment, il est possible d'effectuer un fenêtrage de plusieurs manière.

La fenêtre de base $[X, X']$ x $[Y, Y']$ affichera les segments qui se situent entièrement entre ses bornes ou ceux qui traverse les bords de la fenêtre. Le seul moment où les bords de la fenêtres ne seront pas transpercés par un segment sera quand la taille de cette dernière vaudra la même taille que le canvas.

Les fenêtres avancées semi-bornées $[-\infty, X']$ x $[Y, Y']$; $[X, +\infty]$ x $[Y, Y']$; $[X, X']$ x $[-\infty, Y']$ et $[X, X']$ x $[Y, +\infty]$ effectueront une sélection des segments en laissant un côté de la fenêtre non bornée et qui sélectionnera tous les segments jusqu'à toucher le bord du canvas.

La manière dont nous parcourons l'arbre à priorité nous permet d'effectuer efficacement les fenêtres $[X, X']$ x $[Y, Y']$ et $[-\infty, X']$ x $[Y, Y']$. En ne se concentrant que sur les points de départ des segments, on peut se permettre d'élaguer toute une branche de l'arbre qui ne sera pas à explorer. En effet, le point de départ du segment peut se trouver dans la fenêtre et qu'importe l'endroit ce dernier s'arrête ou le segment à son origine en dehors de la fenêtre du côté gauche ou inférieur de celle-ci. Les origines des segments se trouvant hors de la fenêtre du côté droit ou supérieur ne seront donc pas explorés dans l'arbre.

Afin de pouvoir effectuer les fenêtres non bornées $[X, +\infty]$ x $[Y, Y']$; $[X, X']$ x $[-\infty, Y']$ et $[X, X']$ x $[Y, +\infty]$ nous devons appliquer des transformations sur la fenêtre ainsi que sur l'ensemble des segments présent dans la fenêtre qui vont être affichés.

Il y a 2 opérations que nous effectuons pour obtenir les autres fenêtres :

1. Swap
2. Opposition

4.6.1.1 Swap

Le Swap permet d'effectuer la transformation de la fenêtre $[-\infty, X']$ x $[Y, Y']$ vers la fenêtre $[X, X']$ x $[-\infty, Y']$ en échangeant la coordonnée X du point de départ avec la coordonnée du point d'arrivée de la fenêtre.

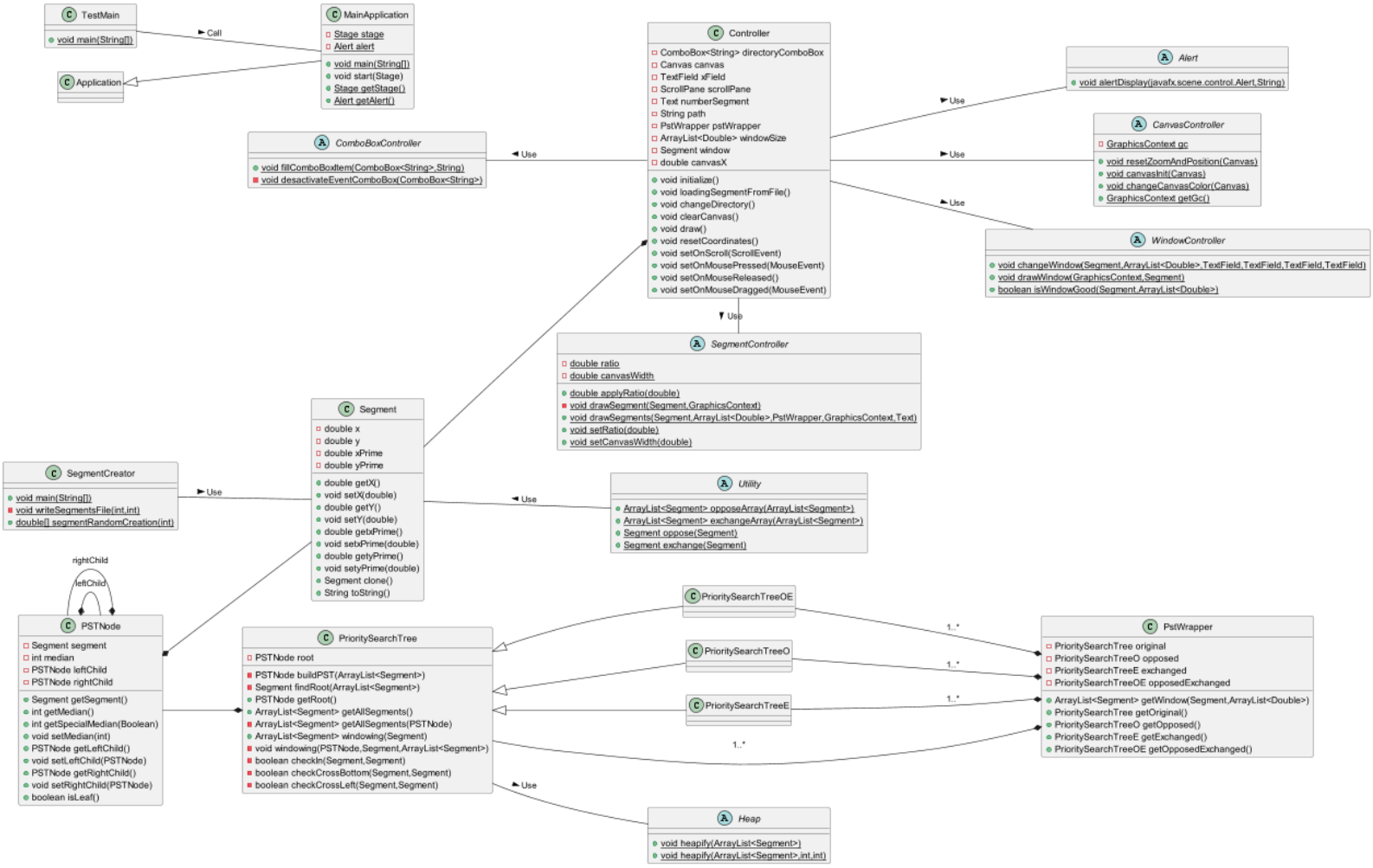
4.6.1.2 Opposition

L'opposition permet d'effectuer la transformation de la fenêtre $[-\infty, X']$ x $[Y, Y']$ vers la fenêtre $[X, +\infty]$ x $[Y, Y']$ en inter-changeant les coordonnées du point d'origine X et X' de la fenêtre.

4.6.1.3 Exemple

Pour effectuer la transformation $[-\infty, X']$ x $[Y, Y']$ -> $[X, X']$ x $[Y, +\infty]$ il suffit d'appliquer les 2 transformations.

5 Diagramme de classe



5.1 Segment

La classe Segment représente un segment avec un point de départ (x,y) et un point d'arrivée (xPrime, yPrime), et elle contient des méthodes pour accéder et modifier les coordonnées des points et pour cloner et afficher la représentation de l'objet.

5.2 Heap

La classe Heap fournit des méthodes statiques pour créer un tas à partir d'un ArrayList de Segments, ce qui peut être utilisé pour trier les segments en fonction de leurs coordonnées y en ordre croissant. Les méthodes heapify effectuent cette tâche en utilisant l'algorithme de tri par tas.

5.3 PstNode

La classe PSTNode représente un nœud dans un arbre de recherche de priorité. Elle contient un objet Segment, une valeur médiane et des pointeurs vers ses enfants gauche

et droit.

5.4 PstWrapper

La classe PstWrapper est une classe d'encapsulation pour les arbres de recherche de priorité, qui fournit une fenêtrage efficace pour des types spécifiques de fenêtres en utilisant plusieurs arbres de recherche de priorité. Elle contient des méthodes pour obtenir des segments d'arbres spécifiques et pour obtenir les segments qui intersectent une fenêtre donnée.

5.5 PrioritySearchTree

La classe PrioritySearchTree est une implémentation d'un arbre de recherche de priorité utilisant une liste de segments pour la construction de l'arbre. Elle contient les méthodes pour construire l'arbre de manière récursive, obtenir la racine de l'arbre, obtenir tous les segments de l'arbre, et rechercher les segments qui intersectent avec une fenêtre donnée. Elle contient également des méthodes utilitaires pour vérifier si un segment intersecte une fenêtre et pour trouver le segment ayant la plus petite coordonnée x dans une liste de segments.

5.6 PrioritySearchTreeO

La classe PrioritySearchTreeO est une extension d'un arbre de recherche de priorité pour laquelle les coordonnées des segments ont toutes été opposées, c'est-à-dire que les X et les Y sont devenu -X et -Y.

5.7 PrioritySearchTreeE

La classe PrioritySearchTreeE est une extension d'un arbre de recherche de priorité pour laquelle les coordonnées des segments ont été échangées, c'est-à-dire que les X et les Y ont été intervertis.

5.8 PrioritySearchTreeOE

La classe PrioritySearchTreeOE est une extension d'un arbre de recherche de priorité pour laquelle les coordonnées des segments ont été échangées et opposées, c'est-à-dire que les X et les Y ont été intervertis puis ont été opposées.

6 Explication des algorithmes et complexité

6.1 Heap

6.1.1 Algorithmes heapify

Algorithm 1: Sort

```
Result: Tas ordonné
Entry : segmentList;
taille ← taille[segmentList];
for i ← (taille/2)-1 à 0 pas de -1 do
    heapify(segmentList, i, taille);
end
for i ← taille - 1 à 0 pas de -1 do
    tempSegment ← segmentList[0];
    segmentList[0] ← segmentList[i];
    segmentList[i] ← tempSegment;
    heapify(segmentList, 0, i)
end
```

Algorithm 2: Heapify

```
Result: Tas ordonné
Entry : segmentList, index, taille;
largestNode ← index;
leftChild ← 2*index+1;
rightChild ← 2*index+2;
if leftChild < taille and segmentList[leftChild].getY() >
 segmentList[largestNode].getY() then
    largest ← leftChild;
end
if right < taille and segmentList[rightChild].getY() >
 segmentList.[largestNode].getY() then
    largestNode ← leftChild;
end
if largest != index then
    tempSegment ← segmentList[index];
    segmentList[index] ← segmentList[largest];
    segmentList[largest] ← tempSegment;
    return heapify(segmentList, largestNode, taille);
end
```

6.1.2 Explication

Cet algorithme met en place une structure de tas (heap) pour trier une liste de segments en fonction de leurs coordonnées Y. Il utilise l'opération de base de la structure de tas pour organiser la liste dans l'ordre croissant des coordonnées Y.

6.1.3 Complexité

La complexité de l'algorithme va dépendre de la taille du tableau en entrée. La taille du tableau sera ici appelé *n*. De manière général, le tri par tas à une complexité se trouvant dans $\mathbf{O(n * log_2 n)}$.

6.1.3.1 Algorithme 1

L'algorithme **Sort** possède une complexité dans le pire des cas en $\mathbf{O(log_2(n))}$. Ce dernier est constitué de 2 boucles *for* toutes les 2 en $\mathbf{O(n)}$

La première boucle construit un tas à partir du tableau d'entrée. Il commence au dernier parent du tas (à l'index N/2-1, où N est la taille du tas) et descend jusqu'à la racine du tas (à l'index 0). Pour chaque nœud parent, appeler la méthode heapify pour conserver la propriété du tas. Puisque la hauteur du tas est $\mathbf{O(log_2(n))}$, la méthode heapify prend $\mathbf{O(log_2(n))}$. Ainsi, la première boucle prend $\mathbf{O(n * log_2(n))}$ pour construire le tas.

Une fois le tas construit, la deuxième boucle de la méthode de tri récupère les éléments du tas afin de trier le tableau. la boucle commence au dernier élément du tas (à l'index N-1) et itère jusqu'à la racine du tas (à l'index 0). À chaque itération, la racine du tas (c'est-à-dire le plus grand élément du tas) est remplacée par l'élément courant tiré du tas. La méthode heapify est ensuite appelée sur le tas réduit (à l'exclusion des éléments extraits) pour conserver la propriété du tas. La méthode heapify prend le temps $\mathbf{O(log_2(n))}$. Ainsi, la deuxième boucle prend $\mathbf{O(n * log_2(n))}$ pour récupérer les éléments du tas et trier le tableau.

6.1.3.2 Algorithme 2

La complexité d'heapify dans le pire cas est en$\mathbf{O(n * log_2 n)}$

La méthode heapify est appelée n fois dans la première boucle de l'algorithme précédent (algorithme 1), où n est la taille du tas. Chaque appel à la méthode heapify prend $\mathbf{O(log_2(n))}$ en temps, de sorte que la complexité temporelle totale de la méthode heapify est $\mathbf{O(n * log_2(n))}$. La méthode heapify elle-même maintient la propriété de tas d'un sous-arbre enraciné à un index i donné dans le tas. Elle initialise d'abord le plus grand élément à la racine (i). Ensuite, elle vérifie si le fils gauche en position (2*i+1) ou le fils droit en position (2*i+2) de la racine est plus grand que la racine. Si c'est le cas, il met à jour le plus grand élément vers l'enfant de gauche ou de droite, respectivement. Si l'élément le plus grand n'est pas la racine, il échange la racine avec l'élément le plus grand et appelle récursivement heapify sur le sous-arbre concerné. Ce processus garantit que le sous-arbre enraciné en i conserve la propriété de tas.

Par conséquent, la complexité de cet algorithme est en $\mathbf{O(n * log_2(n))}$.

6.2 Recherche du X minimum

6.2.1 Algorithme

Algorithm 3: findRoot

```
Result: Le segment avec le plus petit X
Entry : Une liste de segment;
segmentWithLowestX ← null;
for i ← 0 à taille[segments] pas de 1 do
    if segmentWithLowestX est non vide or segments[i].getX() <
     segmentWithLowestX.getX() then
        segmentWithLowestX ← segments[i];
    end
end
return segmentWithLowestX
```

6.2.2 Explication

Le but de cet algorithme est de trouver le segment avec la plus petite coordonnée X dans une liste de segments donnée.

6.2.3 Complexité

La complexité de l'algorithme va dépendre de la taille du tableau en entrée. La taille du tableau sera ici appelé n

- L'initialisation de la variable *segmentWithLowestX* à null qui est en **O(1)**.
- La boucle *for* qui va être exécutée n fois. A l'intérieur de cette boucle *for*, se trouve une instruction *if* qui vérifie si le segment actuel a une coordonnée X inférieure à la coordonnée X du segment avec la plus petite coordonnée X trouvée jusqu'à présent. Si cette condition est vraie, le segment actuel est affecté à la variable *segmentWithLowestX*. Sinon, l'algorithme passe au segment suivant dans la boucle. Le *if* a donc une complexité constante **O(1)** alors que le **for** a une complexité **O(n)**
- En conclusion, la complexité général de cette algorithme est donc dépendant de la taille de la liste de départ soit **O(n)** où n est la taille de la liste

6.3 Priority Search Tree

6.3.1 Algorithme buildPST

Algorithm 4: PrioritySearchTree

```
Result: Arbre PST
Entry : segmentList;
heapify(segmentList);
rootNode ← buildPST(segmentList)
```

Algorithm 5: buildPST

```
Result: noeud PST
Entry : Tas trié segmentList;
if segmentList isEmpty then
    return null;
end
if taille[segmentList] == 1 then
    return PSTNode ← segmentList[0];
end
segmentRoot ← findRoot(segmentList);
segmentList ← segmentList.remove(segmentRoot);
node ← PSTNode(segmentRoot);
if taille[segmentList] > 0 then
    median ← taille[segmentList] / 2;
    node.median ← segmentList[median].getY();
    node.leftChild ← buildPST(segmentsList.subList(0, median)));
    node.rightChild ← buildPST(segmentsList.subList(median,
      taille[segmentList])) ;
end
return node;
```

6.3.2 Explication

L'objectif de ce code est de construire une structure de données d'arborescence de recherche prioritaire (PST) à l'aide d'une liste de segments. PST est une structure de données pour les recherches de plage unidimensionnelle ou bidimensionnelle qui permet une interrogation efficace des segments qui croisent une plage ou une fenêtre donnée.

6.3.3 Complexité

La complexité global des deux algorithmes combinés est alors en $\mathbf{O(n * log_2(n))}$. Et voici le détail pour arriver à ce résultat.

6.3.3.1 Algorithme 4

- La première ligne fait appel à heapify pour trier la liste des segments reçue. L'appel de cette fonction se fait en $\mathbf{O(1)}$ mais la fonction en elle-même s'exécute en $\mathbf{O(n * log_2(n))}$. (Pour plus de détail, voir la partie sur le tas)
- La deuxième ligne fait appel à la fonction buildPst. L'appel en lui même est en $\mathbf{O(1)}$ auquel il faut rajouter la complexité de la fonction buildPST détailler ci-dessous.

6.3.3.2 Algorithme 5

- Les premières instructions que nous pouvons rencontrer sont deux instructions **if**. L'une se charge de vérifier que l'on n'a pas une liste de segment vide et l'autre se charge de vérifier si il n'y a qu'un élément et si c'est le cas de juste renvoyer un noeud seul. Cela se fait en temps constant $\mathbf{O(1)}$
- La méthode findRoot a une complexité ayant déjà été calculé en $\mathbf{O(n)}$. Cela est du au fait qu'elle doit parcourir la liste entière des segments pour s'assurer que le X le plus petit ait bien été trouvé.
- Par la suite, le segment avec le X le plus petit est retiré de la liste et un noeud est créé. Cela se fait en temps constant $\mathbf{O(1)}$
- Le dernier **if** rencontrer va vérifier qu'il reste des éléments dans la liste de segment ce qui se fait alors en $\mathbf{O(1)}$
- Dans le if, on va retrouver une instruction pour initialiser la médiane du noeud qui se fait en $\mathbf{O(1)}$ et deux appels récursifs à la fonction buildPST sur la moitié gauche et droite de la liste ce qui va alors demander un temps proportionnel à la taille de la liste. Chaque sous liste est au plus égal à n/2 et donc la complexité sera de $\mathbf{O(n * log_2(n))}$

6.4 Windowing

6.4.1 Algorithmes Windowing

Algorithm 6: Windowing

Result: liste des segments présent dans la fenêtre
Entry : segment utilisé pour la fenêtre windowSegment, noeud actuel currentNode, segments dans la fenêtre segmentsInsideWindow;
if *currentNode isEmpty* **then**
  return;
end
segment ← currentNode.getSegment();
if *check segment inside window or check segment cross window left side or check segment cross window bottom* **then**
  segmentsInsideWindow[taille[segment]];
end
if *currentNode.getMedian() >= window.getY()* **then**
  windowing(leftChild[currentNode], window, segmentsInsideWindow)
end
if *currentNode.getMedian() <= window.getyPrime()* **then**
  windowing(rightChild[currentNode], window, segmentsInsideWindow);
end

6.4.2 Complexité

L'algorithme Windowing a une complexité en temps dans le pire des cas en $\mathbf{O(n)}$ où n est le nombre de noeuds à parcourir dans l'arbre.

- La condition où on vérifie que le noeud parcourue est vide est en $\mathbf{O(1)}$
- L'assignation à une variable locale du segment contenu dans le noeud parcouru est en $\mathbf{O(1)}$
- La condition où on vérifie si le segment traverse la fenêtre ou si il est présent dans celle-ci est en $\mathbf{O(1)}$
- Les deux appels récursif sont quand a eu en $\mathbf{O(\log_2(n))}$ car il traverse une hauteur du PST

7 Mode d'emploi

7.1 Comment compiler le programme

Pour compiler le programme, nous utilisons gradle. Pour se faire, il faut utiliser la commande gradle suivante : **gradle clean build**.
Il est alors possible de retrouver le résultat du build dans le dossier **build/libs**

7.2 Comment exécuter les tests

Pour exécuter les tests, nous utilisons gradle. Pour se faire, il faut utiliser la commande gradle suivante : **gradle test**.
Il est alors possible de retrouver le résultat dans le dossier **build/reports/tests/test**

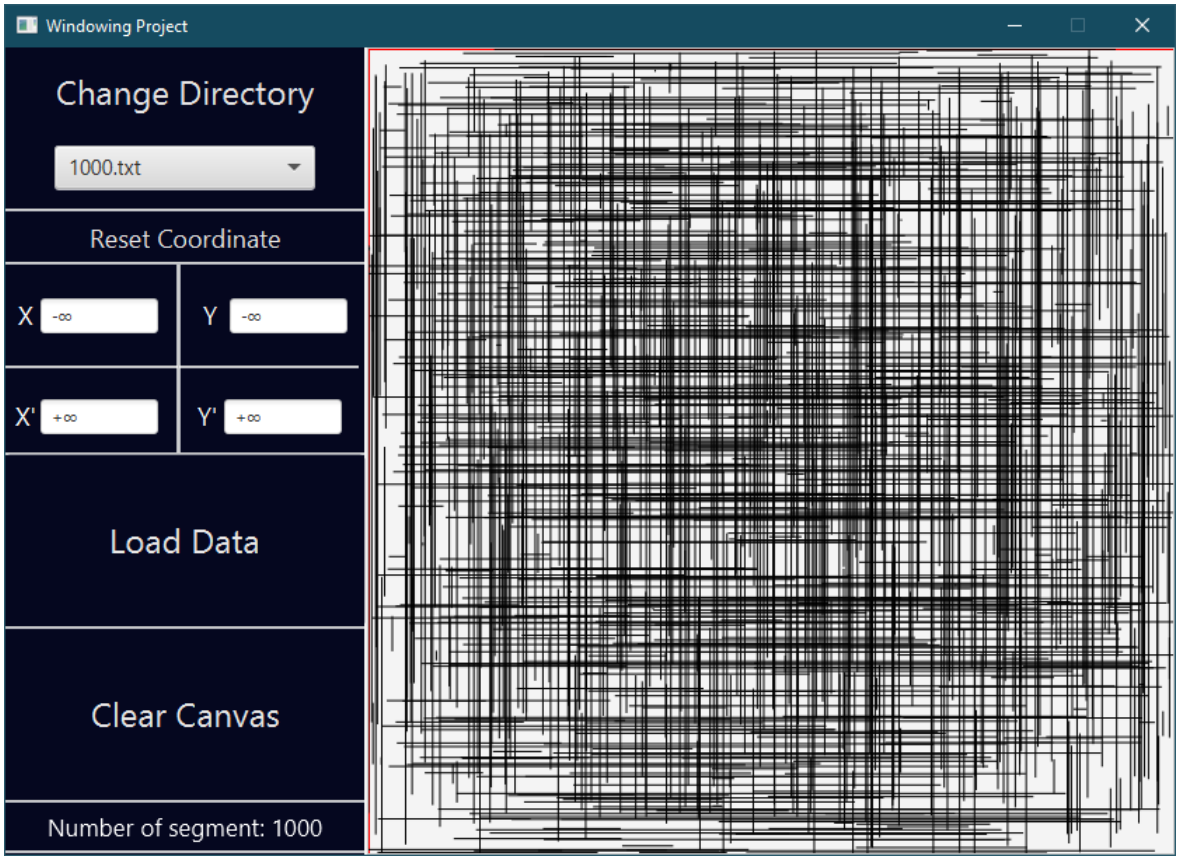
7.3 Comment générer la javadoc

Pour générer la javadoc, nous utilisons gradle. Pour se faire, il faut utiliser la commande gradle suivante : **gradle javadoc**.
Il est alors possible de retrouver le résultat dans le dossier **build/doc**

7.4 Comment exécuter le programme

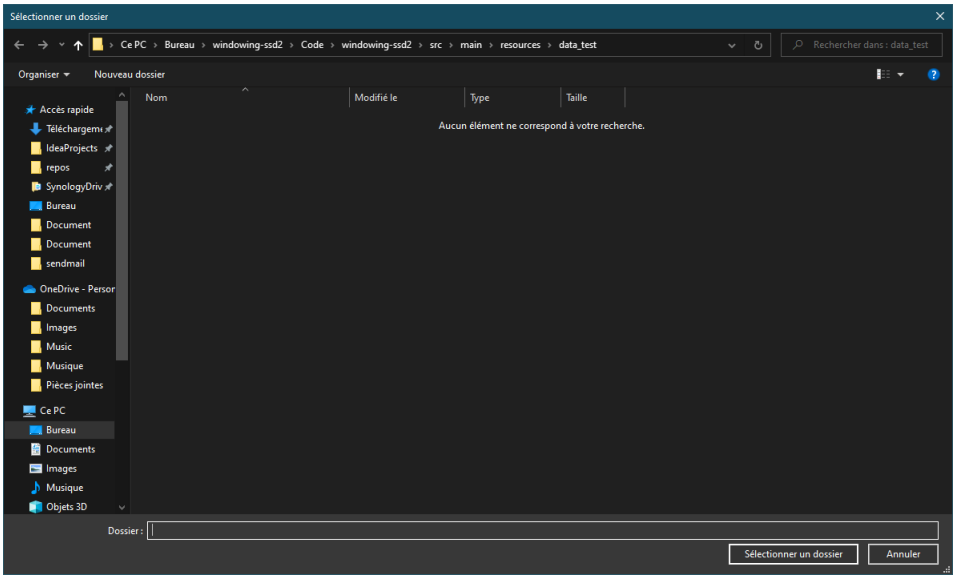
Pour exécuter le programme, deux manières s'offre à vous :
— Utiliser le .jar
— Utiliser Intellij ou un autre IDE et exécuter le TestMain

7.5 Interface utilisateur



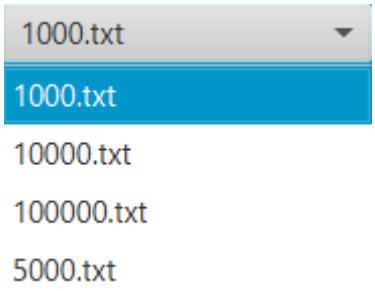
7.5.1 Changement de dossier

Le bouton **Change Directory** permet à l'utilisateur de changer de dossier. Il aura la possibilité grâce à l'interface de windows/apple/linux de sélectionner un dossier.



7.5.2 Sélection de fichier

Le menu déroulant permet de sélectionner quel fichier doit être utilisé lors de la création du Priority Search Tree.



7.5.3 Réinitialisation des coordonnées

Le bouton **Reset Coordinate** permet à l'utilisateur de réinitialiser les coordonnées.

7.5.4 Sélection des coordonnées

Cette partie permet à l'utilisateur de rentrer des coordonnées de la window pour par la suite effectuer le windowing grâce à celle-ci.

7.5.5 Chargement des segments

Le bouton **Load Data** permet à l'utilisateur d'afficher les données sur le canvas. A chaque appui sur le bouton, le zoom ainsi que le déplacement sont réinitialisés aux positions de base.

7.5.6 Effacement du canvas

Le bouton **Clear Canvas** permet à l'utilisateur d'effacer tous les segments sur le canvas.

7.5.7 Nombre de segments

Affiche le nombre de segments actuellement affichés sur le canvas.

7.5.8 Le canvas

Le canvas permet l'affichage des segments ainsi que de la window.

8 Idées d'améliorations

Voici les quelques idées d'améliorations qui pourront être appliquées sur le projet de manière ultérieure :

- Permettre à l'utilisateur d'avoir une window avec plusieurs coordonnées non-bornées au lieu d'une seule.
- Effectuer lors du zoom une réduction de l'épaisseur des segments.

9 Difficultés rencontrées lors du projet

La première difficulté que vous avez mentionnée est la lecture d'un document scientifique en anglais. La lecture de documents scientifiques peut s'avérer difficile même pour les l'anglophone, et elle peut l'être encore plus pour les locuteurs non natifs. Les documents scientifiques utilisent souvent une terminologie complexe, un jargon et un langage technique, ce qui peut les rendre difficiles à comprendre. En outre, les documents scientifiques supposent souvent un certain niveau de connaissances préalables, ce qui peut constituer un défi pour les lecteurs qui ne sont pas familiers avec le domaine ou le sujet. Pour surmonter cette difficulté, il peut être utile de diviser le document en sections plus petites et de se concentrer sur la compréhension de chaque section avant de passer à la suivante.

La deuxième difficulté que vous avez mentionnée est le fait d'avoir un autre projet en même temps pour un autre cours. Il peut être difficile de concilier plusieurs projets et travaux, en particulier lorsque les échéances se chevauchent. Il est facile de se sentir dépassé et stressé lorsqu'on essaie de gérer plusieurs projets à la fois.

10 Conclusion

En résumé, la réalisation de ce projet Data Structures 2 nous a permis de mettre nos connaissances en pratique en utilisant différentes structures de données telles que les tas, les arbres de recherche et les arbres de recherche prioritaires. Chacune de ces structures est utilisée pour résoudre des problèmes spécifiques, tels que le tri et l'organisation efficaces des données, la recherche d'éléments avec une complexité temporelle minimale et l'optimisation de l'accès aux données en fonction de la priorité.

Ce travail, basé sur un livre scientifique de langue anglaise, nous a permis de découvrir des techniques innovantes comme le fenêtrage. De plus, les travaux nous permettront de comprendre l'utilisation de cette technologie dans divers domaines, notamment Google Maps utilisant cette technologie pour la performance de l'application et accélération du chargement des cartes.