II.1102 : Algorithmique et Programmation
TP 4: Structures de données

Patrick Wang

21 – 25 octobre 2019

1 Objectifs du TP

— Continuer de travailler sur les tableaux ;
— Continuer de voir les effets de la portée des variables et du passage par valeur ou référence ;
— Se familiariser avec les structures de contrôle présentées en cours ;
— Introduire quelques algorithmes classiques de manipulation de graphes et d'arbres.

2 Rappels

2.1 Contenant et contenu

Les structures de données présentées en cours ont la particularité d'être des types de variable à part entière. Il faut ensuite définir le type des éléments contenus dans ces structures de données. C'est l'une des principales différences avec les tableaux, dont le type est fortement lié à leurs contenu.

En Java, il existe un grand nombre de structures de données mises à notre disposition. Il est important de les connaître et de savoir les manipuler lorsque nous avons à développer une application en Java (ou dans tout autre langage de programmation). Une compétence importante pour un développeur est donc savoir quand se tourner vers la documentation officielle pour y trouver les informations concernant les méthodes implémentées et les différences entre toutes ces structures de données.

2.2 Piles et files

Les piles et files sont toutes les deux implémentées en Java avec la classe `ArrayDeque`. Ces deux structures de données sont très proches, la seule différence portant sur la façon dont les éléments sont retirés.

Dans le cas d'une file, on parle de *First-In-First-Out*. Cela signifie que les éléments retirés de la file sont ceux qui y ont été ajoutés en premier. Dans le cas d'une pile, on parle alors de *Last-In-First-Out*. Cela signifie que les éléments retirés de la pile sont ceux qui y ont été ajoutés en dernier.

Ces deux structures de données seront probablement intéressantes à utiliser dans le cadre de votre projet, puisque vous aurez à implémenter une file d'instructions pouvant potentiellement être inversée avec la carte "Bug".

2.3 Listes

Les listes sont probablement les structures de données les plus proches de tableaux. Plusieurs implémentations existent et il faut donc en connaître les principales différences : la sauvegarde en mémoire, et la complexité en temps pour récupérer un élément à un indice donné.

Encore une fois, il est important de savoir se retourner vers la documentation officielle pour comprendre les subtilités entre toutes les différentes implémentations.

2.4 Ensembles

Les ensembles sont des structures de données parfois sous-estimées. Ces ensembles se différencient des listes de deux façons : il n'y a pas de notion d'ordre dans un ensemble, et les ensembles garantissent l'unicité des éléments.

Les ensembles sont donc particulièrement intéressants lorsque l'on cherche à déterminer si un élément est présent dans une collection puisque la complexité en temps est de $\mathcal{O}(1)$ contre $\mathcal{O}(n)$ pour les `ArrayList`.

2.5 Dictionnaires

Les dictionnaires utilisent le principe de *(clé, valeur)* pour organiser des données. Il n'y a donc pas de notion d'ordre dans un dictionnaire (sauf si on utilise une `TreeMap`).

Il est donc important de savoir quand utiliser une liste ou un dictionnaire :

— La notion d'ordre est-elle importante pour moi ?

— La notion de clé est-elle importante pour moi ?

Puisque les dictionnaires n'implémentent pas d'indice, le parcours des éléments d'un dictionnaire se fait différemment de celui d'une liste.

2.6 Graphes

Les graphes sont des structures de données n'ayant pas vraiment d'implémentation directe en Java. Pour autant, ces représentations sont si importantes de nos jours qu'il est important de savoir les manipuler.

Les graphes présentant beaucoup de caractéristiques différentes (orienté, pondéré, complet, cyclique, etc.), il est important de choisir la bonne structure de données pour représenter un graphe. L'objectif de ce TP est donc aussi de vous faire réfléchir sur l'utilisation de(s) structure(s) de données adéquate(s) pour résoudre vos problèmes.

3 Exercices

3.1 Exercice 1 : Déplacement sur un plateau

Dans cet exercice, nous allons déplacer un pion sur un plateau de taille 8×8. Pour cela, nous aurons besoin de trois variables :

— Une variable `char[][] plateau` permettant de représenter un plateau de taille 8×8 ;

— Une variable `int[] position` permettant de représenter la position du pion sur le plateau. Ce tableau sera de taille 2 : le premier élément donne l'indice de ligne, le second élément donne l'indice de colonne ;

— Une variable `char direction` permettant de représenter la direction du pion. Les valeurs possibles seront les quatre points cardinaux : `N`, `S`, `E`, `O`.

Questions :

1. Déclarez les variables `plateau`, `position`, et `direction` en tant que variables statiques *globales*.
2. Créez et implémentez une fonction statique `void initialisation()` qui va initialiser le plateau avec des caractères `<espace>` et va positionner le pion dans le coin inférieur gauche.
3. Modifiez la déclaration de la variable `direction` pour l'initialiser avec la valeur `'E'`.
4. Créez et implémentez une fonction statique `ArrayDeque<String> creationFile()` qui va :
   — Déclarer une nouvelle file ;
   — Demander à l'utilisateur de saisir une instruction parmi les trois suivantes :
      — `''A''` pour avancer ;
      — `''G''` pour faire un quart de tour vers la gauche ;
      — `''D''` pour faire un quart de tour vers la droite.
   — Si la saisie est valide, on l'ajoute en fin de file ;
   — Répéter cette saisie tant qu'elle est incorrecte ou que la taille de la file est différente de 5.
   — Retourner la file.
5. Créez et implémentez la fonction statique `void deplacement(ArrayDeque<String> instructions)` qui va :
   — Prendre en paramètre la file d'instructions retournée par la fonction `creationFile()` ;
   — En suivant le principe FIFO d'une file, mettre à jour les variables globales `position` et `direction`.
   — Attention, si une instruction fait sortir le pion du plateau, alors celle-ci est retirée de la file sans être prise en compte.
6. Dans le `main()`, appelez la fonction `initialisation()`.
7. Puis, dans une boucle infinie, appelez les fonctions `creationFile()` et `deplacement()`, puis affichez la valeur des variables `position` et `direction`.

3.2 Exercice 2 : Algorithmes sur les graphes et arbres

3.2.1 Exercice 2.1. : Algorithmes BFS et DFS

Soit le graphe illustré en Figure 1.

Nous allons lui appliquer deux algorithmes de recherche d'éléments : *Breadth First Search* (BFS) et *Depth First Search* (DFS). Ces deux algorithmes permettent de parcourir l'ensemble des nœuds d'un arbre de deux façons différentes.

Le pseudocode pour l'algorithme BFS est le suivant :

```
procedure BFS(Graphe, depart):
    Initialiser une file: queue
    Initialiser un ensemble: discovered
    Initialiser une liste: path
    Ajouter depart a queue
    TANT QUE queue n'est pas vide:
        node <- queue.premierElement
        SI node absent de discovered
            Ajouter node a discovered
            Ajouter node a path
            Pour chaque voisin de node:
```

```
            SI voisin est absent de discovered:
                Ajouter voisin a discovered
                Ajouter voisin a queue
Afficher path
```

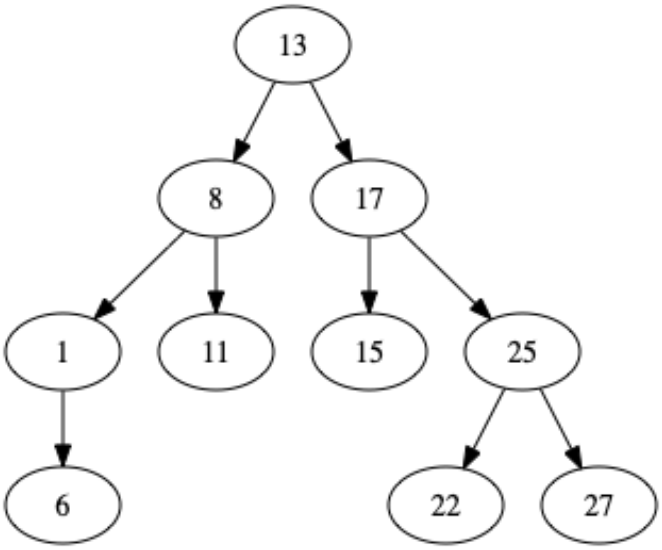

FIGURE 1 – Exemple d'arbre binaire de recherche.

Pour implémenter l'algorithme DFS, il se trouve qu'il faut reprendre l'implémentation de l'algorithme BFS et simplement remplacer la file par une pile.

Questions :

1. Représentez ce graphe à l'aide d'une `TreeMap` ;
2. Créez une fonction statique `void breadthFirstSearch(TreeMap graphe, int depart)` et implémentez-la en vous aidant du pseudocode de la fonction BFS ;
3. Créez une fonction statique `void depthFirstSearch(TreeMap graphe, int depart)` et implémentez-la en vous aidant du pseudocode de la fonction BFS, puis en remplaçant la file par une pile ;
4. En analysant les *chemins* affichés par ces deux fonctions, pouvez-vous expliquer pourquoi ces deux algorithmes s'appellent ainsi ?

3.2.2 Algorithme de Dijkstra

Soit le graphe pondéré illustré en Figure 2. L'objectif de cet exercice est de déterminer le plus court chemin pour aller de `A` à `J`.

L'algorithme de Dijkstra est un algorithme permettant de calculer le plus court chemin entre deux sommets d'un graphe, cet algorithme suit le principe suivant :

Initialisation Tous les sommets sont à une distance ∞ du point de départ (en Java, on pourra utiliser la valeur max des entiers `Integer.MAX_VALUE`).

Principe En partant du point de départ, on va mettre à jour les distances des sommets atteignables (cf. les sommets B, C, et E en Figure 2). Puis, on va décider de se rendre au

sommet **minimisant** la distance totale parcourue. En se déplaçant sur un sommet, on va ensuite mettre à jour la distance des sommets nouvellement atteignables.

Fin On répète ce principe tant que chaque sommet n'a pas encore été atteint.

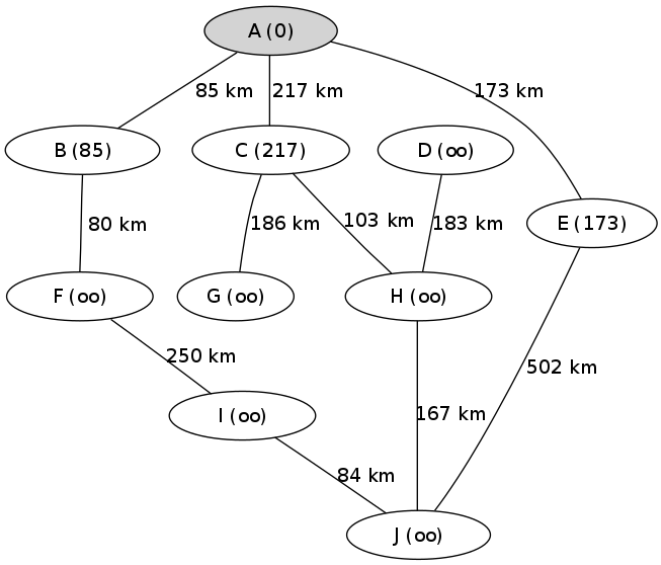

FIGURE 2 – Graphe pondéré

Une description animée est disponible sur la page Wikipédia de l'algorithme de Dijkstra : `https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra`. La Figure 2 est d'ailleurs prise de cette page.

Questions :

1. Prenez le temps de vous familiariser avec l'algorithme de Dijkstra en consultant les pages Wikipédia française et anglaise ;
2. Réfléchissez aux structures de données que vous utiliserez pour implémenter cet algorithme de Dijkstra ;
3. En vous appuyant sur le pseudocode fourni sur les pages Wikipédia, implémentez l'algorithme de Dijkstra et vérifier que le chemin le plus court trouvé correspond à `A-C-H-J`.