Grands Moulins

20/06/2021 - 9h00-11h00

Initiation à la Programmation (IP2)

Rappel : vous disposez de brouillon pour réfléchir. Les réponses attendues tiennent en quelques lignes seulement : comprenez bien que cela ne vous prendra pas beaucoup plus de temps de n'écrire au propre que lorsque vous aurez les idées claires. Si en revanche vous rendez quelque chose de confus, de peu lisible, d'ambigu, le correcteur ne peut que le constater et ne pas vous donner de bénéfice pour des présentations approximatives.
Conseil méthodologique : appliquez vous à terminer un nombre éventuellement limité d'exercice mais faits soigneusement. Il vaut mieux en faire un peu moins complètement, plutôt que d'essayer de tout faire maladroitement. Le barème est indicatif. **Aucun document autorisé**.

Exercice 1 (4 points)
On vous rappelle les classes qui permettent de définir une liste simplement chaînée.

```
public class List {
  private Cell head;
}
public class Cell {
  public int data;
  public Cell next;
}
```

Le problème est le suivant : étant donnés une liste l et un paramètre k, on veut modifier l'ordre des cellules de l en effectuant une "rotation" de k cellules dans l'ordre inverse des aiguille d'une montre.

Par exemple si la tête de l pointe sur une cellule contenant 10 suivie de $20, 30, 40, 50, 60$, terminé par *null*, et que $k = 4$. Après cette "rotation" l pointera vers des cellules contenant, dans cet ordre : $50, 60, 10, 20, 30, 40$ (et toujours terminé par *null*)

Quelques précisions :

- on part ici de vrai listes simplement chaînées, et pas de listes circulaires.
- vous n'utiliserez que les cellules qui existent déjà, sans en construire de nouvelles.
- les champs sont public pour simplifier votre rédaction : **n'écrivez pas de getter ou de setter**

1. (2 points) Ecrivez une méthode `void rotate_it(int k)` de `List` qui est itérative et procède ainsi :
   - elle ne fait rien dans le cas d'une liste vide ou qui n'a qu'un élément
   - elle recherche le dernier élément et le relie au premier (la liste devient temporairement circulaire). On continue en progressant encore de k cellules supplémentaires ; puis on peut la couper et redéfinir la tête convenablement pour que la liste redevienne "normale" et porte le résultat attendu.
2. (2 points) On veut ici une solution purement récursive. Il vous faudra écrire une méthode `void rotate_rec(int k)` dans `List` et une méthode `void append_rec(Cell x)` dans `Cell` qui ajoute une cellule en fin. Vous pourrez procéder ainsi :
   - écarter les cas triviaux (ou terminaux)
   - isoler la première cellule de la liste, et l'ajouter en fin
   - recommencer tant que nécessaire

Exercice 2 (4.5 points) On utilise ici un système à 2 tableaux de même longueur *etiq* et *pere* pour modéliser un arbre dont les noeuds sont étiquetés par des caractères. L'idée est que chaque indice i permette de caractériser un noeud différent : la valeur de son étiquette est donnée par $etiq[i]$, l'indice de son père est caractérisé par $pere[i]$ (vaut -1 par convention lorsqu'il n'y a pas de père). Si deux noeuds ont le même père, le fils gauche est celui de plus petit indice, l'autre sera fils droit.

1. **(1 point)** Dessinez l'arbre que modélise ce système :

```
char [] etiq= {'p', 'a', 'l',  'm',  'i', 'e', 'r', 's'};
int [] pere =  {-1, 0, 0, 1, 2, 2, 4, 5};
```

2. On veut s'assurer qu'un tableau pere non null décrit une situation qui est bien représentable par un arbre binaire. Ecrivez :

   (a) **(0.5 point)** une méthode checkOneRoot qui s'assure qu'il y aurait bien une seule racine,

   (b) **(0.5 point)** une méthode checkLegalValues qui s'assure que les valeurs d'indices des pères seraient légales,

   (c) **(1 point)** une méthode checkBin qui s'assure que dans l'arbre personne n'aurait plus de 2 fils.

3. **(1.5 point)**

   On vous donne la trame de code suivante qui définit les noeuds et les arbres. Vous y trouverez une méthode traduit qu'il faudra compléter : elle prend en argument les 2 tableaux dont nous venons de décrire l'interprétation, et elle devra retourner l'arbre correspondant à ce système si c'est possible.

```
public class Node {
  private char etiq;
  private Node fg, fd;
  Node(char e, Node x, Node y) { etiq=e; fg=x; fd=y; }
  public void setG(Node x) {fg =x;}
  public void setD(Node x) {fd =x;}
  public Node getG() {return fg;}
}
// ------------------------------------------------------------------------
public class Tree {
  private Node root;
  private Tree(Node r){ root=r;}

  public static Tree traduit(char [] etiq, int [] pere) {
    // NE PAS COMPLETER LA PARTIE SUIVANTE (elle n'apporte aucun point)
    if (...) {
      // traite les cas ou l'un des tableaux est nul,
      // ou ils sont de tailles differentes,
      // ou ils ne remplisent pas les conditions du codage
      return new Tree(null);
    }
    // ON SUPPOSE MAINTENANT QU'ON EST DANS DE BONNES CONDITIONS
    int nb=pere.length;
    Node [] all=new Node [nb];
    for (int i=0;i<nb;i++) all[i] = new Node (etiq[i],null, null);
    Node racine=null; // qui changera au cours de vos calculs
    .... // a completer
    return new Tree(racine);
  }
}
```

à partir de la ligne 22, on peut se concentrer sur le décodage du système. On a déjà commencé pour vous à construire dans all autant de noeuds que nécessaire, en ne leur donnant que leurs valeurs d'étiquette et en fixant leurs fils gauches et droits à null

A partir de la ligne 27 il vous reste à effectuer les rattachements des noeuds de all entre eux, et trouver la racine.

Ecrivez les quelques lignes qui se trouveront là.

Exercice 3 (6 points)

Cet exercice a pour point de départ l'observation d'un arbre naturel où sont éventuellement posés des oiseaux. Voici les classes que nous utiliserons :

```
public class Oiseau {
  public int poids;
  public char espece;
}
public class Noeud {
  private Oiseau bird;
  private Noeud filsG;
  private Noeud filsD;
}
public class Arbre {
  private Noeud sommet;
}
```

1. **(1 point)** Nous allons attribuer un nom d'espèce aux oiseaux dont on constate qu'ils sont effectivement présents dans l'arbre. On décide que l'oiseau du sommet sera de l'espèce 'A', ceux qui sont à l'étage juste inférieur seront de l'espère 'B' et ainsi de suite. On établi ainsi une correspondance entre le nom de l'espèce et la distance au sommet. Ecrivez cette méthode void nommeEspece() dans Arbre ainsi que celle qui lui est naturellement associée dans Noeud. On rappelle qu'en java, on peut se permettre des opérations entre les caractères et entiers, ainsi :

   ```
   char x='a';
   for (int i=0;i<26;i++) System.out.print(x+i);
   System.out.println();
   ```

   affiche l'alphabet sur une seule ligne.

2. **(2 points)** On souhaite établir la liste des oiseaux présents en les prenant dans l'ordre qui correspond au parcours en largeur de l'arbre (d'abord l'oiseau du sommet, puis ceux à distance 1 en les prenant de gauche à droite, puis ceux à distance 2, etc ...).
   Ecrivez deux méthodes jumelles (dans Arbre et dans Noeud) LinkedList<Oiseau> repertorie() qui permettent d'établir cette liste. Vous utiliserez des LinkedList<Oiseau> et des LinkedList<Noeud> disponibles grâce à la librairie standard java. Elles sont munies des opérations habituelles[1], inutile donc de les redéfinir.

3. On a l'intuition que plus les espèces sont lourdes en moyennes, plus elles occupent une position élevée dans l'arbre. Nous allons nous donner les moyens d'en avoir le coeur net en 2 étapes.

   (a) **(2 points)** en partant du résultat de repertorie, vous allez calculer la moyenne des poids de chaque espèce, puis construire un seul "oiseau moyen" par espèce et le stocker dans une nouvelle liste de prototypes. (Il représentera symboliquement chaque espèce en moyenne). Ecrivez une/des méthodes pour implémenter totalement static LinkedList<Oiseau> construitPrototypes(LinkedList<Oiseau> l)

   (b) **(1 point)** Ecrivez une méthode qui permette d'analyser le résultat précédent et confirme/infirme notre intuition.

[1] add qui ajoute en fin ou à un endroit précis, clear, contains, get, getFirst isEmpty, remove, removeFirst, size ...

Exercice 4 (5 points) Dans cet exercice on vous fourni un code complet, qui s'exécute sans erreurs. Il ne possède aucun commentaires explicatifs, et les noms des méthodes/variables sont complètement neutres (a, b, c, f, g, h, k, x), ce qui rend difficile de savoir ce qui est fait. C'est l'objet de cet exercice.

1. **(2 points)** Arrivé en ligne 20, qui a t'il d'affiché à l'écran ?
2. **(3 points)** Attention, cette seconde question est plus difficile, vous pourriez y passer un gros quart d'heure (au brouillon !). On attend de vous :
   (a) que vous nous donniez le nouvel affichage fait lorsqu'on arrive en ligne 23
   (b) une description synthétique du rôle que joue la fonction h et de celui de son paramètre.

```
public class List {
  private Cell head;
  public void f(int d){
    Cell x = new Cell(d);
    x.next = head;
    head = x;
  }
  public void g() {
    Cell temp = head;
    while (temp != null) {
      System.out.print(temp.data + " ");
      temp = temp.next;
    }
    System.out.println();
  }
  public static void main(String [] args){
    List t = new List();
    for (int i=9;i>0;i--) t.f(i);
    t.g();
    // Question 1
    t.head = t.head.h(3);
    t.g();
    // Question 2
  }
}
public class Cell {
  public int data;
  public Cell next;
  public Cell(int d){ data = d; next = null; }
  public Cell h(int k){
    Cell a = null;
    Cell b = this;
    Cell c = null;
    int x = 0;
    while (x < k && b != null) {
      c = b.next;
      b.next = a;
      a = b;
      b = c;
      x++;
    }
    if (c != null) this.next = c.h(k);
    return a;
  }
}
```