Cours Developpement Efficace

Introduction aux Listes simples en Java

En Java, on a une classe Cellule `Cell.java` qui représente une Cellule pour notre Liste

`[1, null] -> [2, null] -> [3, null] -> |///`

```java
// class Cell.java
public class Cell {
    public int value;
    public Cell next;
    public Cell(int value) {
        this.value = value;
        next = null;
    }
}
```

```java
// class List.java
public class List {
    public Cell head;
    public int size;
    public List() {
        head = null;
        size = 0;
    }
    // Reste du code ci-dessous
}
```

Maintenant on souhaite renvoyer une cellule par la méthode `public Cell find (int value) {}`.

```java
public Cell find (int value) {
    Cell c = head;
    while ((c != null) && (c.value != value)) {
        c = c.next;
    }
    return c;
}
```

Maintenant on souhaite renvoyer la valeur d'une cellule par la méthode `public Cell get (int index) {}`.

```java
public Cell get (int index) {
    Cell c = head;
    int i = 0;
    while ((c != null) && (i < index)) {
        c = c.next;
        i += 1;
    }
}
```

On souhaite ajouter une valeur a la liste par la méthode `public Cell append (int value) {}`.

```java
public Cell append (int value) {
    Cell c = null;
    Cell newCell = new Cell (value);
    if (size == 0) {
        head = newCell;
    } else {
        c = get(size - 1);
        c.next = newCell;
    }
    size += 1;
    return newCell;
}
```

On souhaite ajouter une valeur à un index précis par la méthode `public Cell insert (int value, int index) {}`.

```
public Cell insert (int value, int index) {
    Cell c = null;
    Cell newCell = new Cell(value);
    if (index > size) { index = size; }
    if (size == 0) { head = newCell; }
    else if (index == 0) {
        newCell.next = head;
        head = newCell;
    } else {
        c = get(index-1);
        newCell.next = c.next;
        c.next = newCell;
    }
    size += 1;
    return newCell;
}
```

On souhaite maintenant remplacer une valeur par la méthode `public Cell replace (int replace, int index) {}`.

```
public Cell replace (int value, int index) {
    if ((index<0) && (index >= size)) return null;
    Cell c = get(index);
    c.value = value;
    return c;
}
```

Les Collections en Java

- **Set** : ensemble d'objets non indicé, sans doublons
- **List** : ensemble d'objets non indicé, éventuellement avec doublons
- **Map** : ensemble associatif d'objets, non indicé, chaque objet étant associé à une clé
  - Une clé est unique dans **Map** mais plusieurs clés peuvent être associées à un même objet.
- **Queue** : ensemble d'objets, non indicé avec un schéma d'accès de type FIFO/LIFO
  - **FIFO** : First In First out
  - **LIFO** : Last In First Out

Les piles sont souvent utilisés en programmation.

En **réalité**, Set, List, Map et Queue sont des interfaces.

En **pratique**, on utilise les implémentations HashSet, ArrayList, HashMap, ArrayDeque. Ces collections sont génériques :

```
// On ne peut pas écrire cela
Set <int> set = new HashMap<int>();

// On écrit plutôt cela
Set <Integer> set = new HashMap<Integer>();
Set <Integer> set = new HashMap<>();

// ------------------
// Methodes communes
// ------------------
// int size();
// void clear();
// boolean istemply();
```

HashSet: - **boolean add(E e)**: ajoute l'élement e de la classe E. Si E n'est pas de la classe (même classe) indiquée par constructeur, erreur compil. Renvoie vrai si l'insertion a lieu.

- **boolean remove(Object o)**: enlève o s'il existe et renvoie true, sinon false, pas d'erreur de compil si o n'est pas de la même classe.

- **boolean contains(Object o)**: vérifie si l'élément spécifié est présent dans le HashSet et retourne true si l'élément existe, sinon elle retourne false.

Exemple :

```
public class A {}
public class B extends A {}

Set<A> set = new HashSet<>();

set.add(new A());
set.add(new A());
set.add(new B());

set.add(new Date()); // Cette ligne provoque erreur compil car Date pas enfant de A
```

ArrayList: - **boolean add(E e)**: ajoute en fin de liste.

- **void add(int index, E e)**: insertion en index. Si index < 0 ou index $>$ taille liste, ne fait rien et lève une exception.
- **E get(int index)** : retourne l'élément à l'index spécifié.
- **int indexOf(Object o)** : retourne l'index de la première occurrence de l'élément o, ou -1 si l'élément n'est pas trouvé.
- **E remove(int index)** : supprime et retourne l'élément à l'index spécifié.

Exemple :

```
import java.util.ArrayList;

public class A {}
public class B extends A {}

List<A> list = new ArrayList<>();
A a1 = new A();
A a2 = new A();
Date d = new Date();

list.add(a1);
list.add(a2);
list.add(d);
list.add(15, a1); // Erreur d'execution

A aa = null;
aa = list.get(0); // aa référence le même objet que a1
aa = list.remove(1); // aa référence le même objet que a2
int pos = list.indexOf(a1); // Ok, renvoie 0
pos = list.indexOf(d); // Ok, renvoie -1
```

HashMap : - **V put (K key, V value)**: rajoute/modifie couple {clé; valeur}. Si clé existe déjà l'ancienne valeur est écrasée par la nouvelle. Renvoie l'ancienne valeur ou null su elle n'existe pas.

- **V get(Object key)**: renvoie l'objet associé à la clé key si elle existe ou null.
- **V remove(Object key)**: supprime la valeur associée à la clé.
- **boolean containsKey(Object key) / boolean containsValue(Object value)**
- **Set keySet()**: renvoie un set des clés.

Exemple :

```
Map<String, A> map1 = new HashMap<>();
Map<A, Integer> map2 = new HashMap<>();

A a1 = new A();
A a2 = new A();
Date d = new Date();
```

```
map1.put("toto", a1);
map1.put("toto", a2); // écrasement de a1 par a2
map1.put(a1, "tutu"); // erreur compil
map2.put(a1, new Integer(10));
map2.put(d, new Integer(5));
map2.containsKey(d); // renvoie False
map2.containsKey(a1); // rnevoie True
map2.remove(d); // Ne fait rien
map1.remove("toto"); // Renvoie a1
map2.remove(a1); // renvoie 10
```

ArrayDeque: - **File**: - **boolean offer(E e)**: ajoute e en fin de queue - **E poll()**: supprime et renvoie l'élément en tête de queue. Si la queue est vide, renvoie null. - **Pile**: - **void push(E e)**: ajoute en tête de queue. - **E pop()**: supprime et renvoie l'élément de tête de queue. Contrairement à pull provoque erreur si queue vide.

- **boolean offerFirst(E e)/boolean offerLast(E e)**
- **void addFIrst(E e)/void addLast(E e)**
- **E getFirst()/E getLast()**
- **E peekFirst()/E peekLast()**

Exemple:

```
import java.util.ArrayDeque;

Queue<Double> q = new ArrayDeque<>();
// FIFO
q.offer(1);
q.offer(2);
int val = q.pull(); // renvoie 1

// LIFO access
q.push(3);
val = q.pop(); // renvoie 3
val = q.pop(); // renvoie 2
```

Parcourir une collection en Java

Par le biais de : for

Contrainte: on ne peut pas modifier la collection que l'on parcoure.

Exemple :

```
import java.sql.Array;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;

Set<String> set = new HashSet<String>();
List<Date> list = new ArrayList<>();
Map<Integer, String> map = new HashMap<>();

for(String s : get) {
    // s ...
}
for (Date d : list){
    // d ...
}
for (MapEntry<Integer, String> e : map.entrySet()){
    Integer k = e.getKey();
    String v = e.getValue();
}
```

Par le biais de : iterator

Exemple :

```
import java.sql.Array;
import java.util.*;

Set<String> set = new HashSet<String>();
List<Date> list = new ArrayList<>();
Map<Integer, String> map = new HashMap<>();

// SET
Iterator<String> is = set.iterator();
while(is.

hasNext()){
String s = is.next();
// ...
if(...) is.remove();

remove();
}

// LIST
ListIterator il = list.listIterator();
while (il.hasNext()) {
    Date d = il.next();
    // ...
    il.previous();
    il.add(E e);
    il.remove();
}

// MAP
Iterator<Integer> in = map.keySet().iterator();
while (in.hasNext()) {
    Integer k = in.next();
    String v = map.get(k);
    // ...
}
```

Astuce :

```
Map<Integer, Double> map = new HashMap<>();
// Mauvaise pratique : attention aux types ! Fonctionne quand même en Java
map.put(5, 1.234); // le compilateur va faire le nécessaire pour transform int en Integer et double en Double
// Bonne pratique
double val = map.get(5); // val = 1.234
Double d = 6.6;
if (d == 6.6) {
    // ...
}
```

Les arbres en Java

Parcours en profondeur d'abord

```
type_retour parcours(Node n)
    // traitement 1/n (optionnel)
    pour caque fils de n
        type_retour val = parcours (fils)
        // tests sur val (optionnel)
        si val == ...
```

```
        sinon ...
        fin si
    fin pour
    // traitements (optionnel)
    retourne valeur_defaut
```

Exemple d'arbre :

```
        1(5)
  2(1)  3(3)  6(4)
        4(8)
        5(2)
```

Exemple de Noeud en Java :

```
class Node {
    public int val;
    public List<Node> children;

    Node (int val) {
        this.val = val;
    }

    // ...
}
```

Méthode pour trouver une valeur dans un arbre en Java:

```
boolean contains(Node n, int val) {
    // Si noeud à la valeur souhaitée, on renvoie la valeur.
    if (n.noeud == val) {
        return true;
    }

  // Test pour toutes les branches.
    for (Node fils : n.children) {
        boolean rep = cotains(fils, val);
        if (rep == true) return true;
    }

    return false;
}
```

Méthode récursive pour trouver la taille de l'arbre :

```
int treeDepth(node n, int level) {
    // Si on est sur une feuille, alors size du fils 0 donc on return level.
    if (n.children.size() == 0) {
        return level;
    }
    int maxDepth = 0;

    // Test pour toutes les branches.
    for (Node fils : n.children) {
        int depth = treeDepth(fils, level + 1);
        // Si la hauteur est supérieur pour cette branche.
        if (depth > maxDepth) {
            maxDepth = depth;
        }
    }

    return maxDepth;
}
```

Méthode non récursive pour trouver la taille de l'arbre :

```
int treeDepth(Node n) {
  // Si on est sur une feuille, alors size du fils 0 donc on renvoie level.
  if (n.children.size() = 0) {
    return 1;
  }
  int maxDepth = 0;

  // Test pour toutes les branches.
  for (Node fils : n.children) {
    int depth = treeDepth(fils);
    // Si la hauteur est supérieur pour cette branche.
    if (depth > maxDepth) {
      maxDepth = depth;
    }
    return maxDepth + 1;
  }
}
```

Méthode non récursive pour trouver le nombre de feuille dans un arbre :

```
int nbNeaves(Node n) {
  // Si on est sur une feuille, alors on renvoie 1.
  if (n.children.size() == 0) {
        return 1;
    }
    int nbLeaves = 0;
    // Test pour toutes les branches.
    for (Node fils : n.children) {
        // On fait le processus pour chaque nœud fils.
        nbLeaves += nbLeaves(fils);
    }
    return nbLeaves;
}
```

Parcours en largeur d'abord

Exemple d'arbre :

```
        1(5)
  2(1)  3(3)  4(4)
  5(2)  6(8)
        7(2)
```

```
public boolean contains(Node n, int val) {
    // Si la valeur du nœud actuel est celle que l'on cherche.
    if (n.val == val) {
        return true;
    }

    // Initialisation d'une file FIFO pour effectuer une recherche en largeur (BFS).
    Queue<Node> queue = new ArrayDequeue<>();

    // Ajoute tous les enfants du nœud actuel à la file.
    for (Node fils : n.children) {
        queue.offer(fils); // Ajoute chaque enfant à la queue.
    }

    // Tant que la file n'est pas vide, continue la recherche.
    while (!queue.isEmpty()) {
        // Récupère et supprime le nœud en tête de la file.
        // poll() est utilisé pour retirer le premier élément de la queue.
```

```
        Node parent = queue.poll();

        // Vérifie si le nœud courant contient la valeur recherchée.
        if (parent.val == val) {
            return true;
        } else {
            // Ajoute tous les enfants du nœud courant à la file pour les explorer ensuite.
            for (Node fils : parent.children) {
                queue.offer(fils);
            }
        }
    }
    return false;
}
```

Méthode pour utiliser un neoud sératatif:

```
// Nouvel arbre.

        1(5)
  2(1)  3(3)  4(4)
  5(2)  6(8)  7(10)
        8(12)

// La méthode de séparation.

[1][3][4][///][2][8][10][///][12]
```

Méthodes pour obtenir nombre de noeuds par level avec la séparation :

```
import java.util.ArrayDeque;
import java.util.ArrayList;

public List<Node> getNodesForLevel(Node n, int Level) {
  // Ajoter les noeuds séparatifs
  List<Node> list = new ArrayList<>();
  if (level == 0) {
    list.add(n);
  } else if (level > 0) {
    int currentLevel = 1;
    Node separ = new Node(-9999);
    Queue<Node> queue = new ArrayDeque<>();
  }
    for (Node fils : n.children) {
        queue.offer(fils);
    queue.offer(separ);

    // Donner les profondeurs pour les différentes branches
    while (!queue.isEmpty()) {
        Node parent = queue.poll();
        if (parent == separ) {
            if (currentLevel == level) {
                return list;
            } else {
                currentLevel += 1;
                queue.offer(separ);
            }
        }
        else {
            if (currentLevel == level) {
                list.add(parent);
            } else {
```

```
                for(Node fils : parent.children) {
                    queue.offer(fils);
                }
            }
        }
    }
  }
}
```

Méthode pour trouver la taille maximum par hauteur:

```
        (4)           1
  (1)   (2)   (5)     3
(3)(10)(18)(2)  (4)   5
                (5)   1
```

Développement inefficace (parcours plusieurs fois l'arbre):

```
public int maxWidth(Node n) {
    int max = 0, width = 0, level = 0;
    while ((width = nbNdesByLevel(n, 0, level)) > 0) {
        if (width > max) {
            max = width;
        }
      level++;
    }
}

public int nbNodesByLevel(Node n, int currentLevel, int searchLevel) {
    if (currentLevel == searchLevel) return 1;
    int nb = 0;
    for (Node fils : n.children) {
        nb += nbNodesByLevel(fils, currentLevel+1, searchLevel);
    }
    return nb;
}
```

Développement efficace (parcours une seule fois l'abre) :

```
import java.util.ArrayList;
import java.util.List;

public int maxWidth(Node n) {
  int max = 0;
  List<Integer> nbNodesByLevel = new ArrayList<>();
  countByLevel(n, 1, nbNodesByLevel);
  // trouver le maximim dans nbNodesByLevel.
  return max;
}

public void countByLevel(Node n, int level, List<Integer>nbNodes) {
    if (nbNodes.size() < level) {
        nbNodes.add(1); // rajoute un 1 pour tous les levels.
    } else {
        nbNodes.set(level, nbNodes.get(level) + 1);  // on rajoute en + de ceux déjà comptés.
    }
    for (Node fils : n.children) {
        countByLevel(fils, level+1, nbNodes);
    }
}

// [1, 3, 5, 1]
```

Arbre binaire ordonné

```
class Node {
  Node left;
  Node rigth;
  int value;

  public Node(int value) {
    this.value = value;
    left = null;
    rigth = null;
  }


}
```

```
        (4)
  (12)      (28)
(1)  (15)
    (13)
```

Méthode d'insertion :

```
public void insert(int value) {
  insertRecur(root, value);
}

private void insertRecur(Node n, int value) {
    // Si arbre vide on met au root.
  if (n == null) {
    root = new Node(value);
  } else if (value <= n.value) {
    // Si il y a un noeud gauche.
    if (n.left != null) {
      insertRecur(n.left, value);
    } else {
      n.left = new Node(value);
    }
    // else if n'est pas forcément nécessaire.
  } else if (value > n.value) {
    // Si il y a un noeud droit.
    if (n.right != null) {
        insertRecur(n.rigth, value);
    } else {
        n.rigth = new Node(value);
    }
  }
}

public void insertIter(int value) {
    Node newNode = new Node(value);
    // Si arbre vide on met au root.
    if (root == null) {
        root = newNode;
        return;
    }
    // On se met à la racine.
    Node n = root;
    Node father = null; // pour pouvoir faire marche arrière.
    int lastDirection = -1; // 1: gauche 2: droite
    while (n != null) {
        father = n;
```

```
        if (value <= n.value) {
            n = n.left;
            lastDirection = 1;
        } else {
            n = n.rigth;
            lastDirection = 2;
        }
    }
    if (lastDirection == 1) father.left = newNode;
    if (lastDirection == 2) father.rigth = newNode;
}
```

Méthode de recherche dans un arbre binaire ordonné :

```
public Node search(int value) {
    Node n = root;
    while ((n != null) && (n.value != value)) {
        if (value <= n.value) {
            n = n.left;
        } else {
            n = n.rigth;
        }
    }
    return n; // return nœud si trouvé ou null si valeur non trouvée.
}
```

Intêret de ce type d'arbre : recherche beaucoup plus rapide.