IUT2 Université Grenoble Alpes — Département INFO

R1.01 – TP11
Listes chaînées (suite) / Pile

SI VOUS N'AVEZ PAS FINI LE TP10 : terminez-le avant de vous attaquer à celui-ci.

SINON : Nous vous proposons dans ce TP…

- *PARTIE A : de parfaire votre apprentissage des listes chaînées en codant de nouveaux algorithmes*
  NOTE : D'autres algorithmes pourront vous être proposés par vos enseignants.
- *PARTIE B : de réaliser une petite application de simulation d'une calculette RPN*

NOTE : Vous restez libres de commencer par l'une ou l'autre de ces parties

Partie A : Listes chaînées

1. Affichage détaillé et recherche d'une cellule dans une liste

1.1. Dans la classe `Utilitaire` du projet `TP10`, ajoutez puis codez les procédures suivantes :

a) Affichage détaillé d'une cellule

```
public static void afficheCellInt(Cellule<Integer> uneCellInt) {
    // { uneCellInt n'est pas null }
    //    => {l'adresse et l'info protée par uneCellInt ont été affichées
    //        Exemple : (Cellule@3f91beef / 80 }
```

b) Affichage détaillé, gauche droite d'une liste ce cellules, avec saut de ligne toutes les 5 cellules

le Modèle

```
public static void afficheGDdetaille(ListeChainee<Integer> listeInt) {
    // { } => { les cellules de listeInt ont été affichées
    //          de la 1ère à la dernière }
```

le Worker

```
private static void afficheGDdetailleWorker(Cellule<Integer> cellCour,
                                            int pos) {
    // { pos = position de cellCour dans la liste, paramètre du modèle ] =>
    // { affichage récursif avec saut de ligne toutes les 5 cellules }
```

NOTE : le worker devra utiliser `afficheCellInt`

c) Cellule ayant une position donnée dans une liste non vide – FORME SÉQUENTIELLE

```
public static Cellule<Integer> getCellPos(ListeChainee<Integer> listeInt,
                                          int pos) {
    //{ liste non vide, pos compris entre 1 et le nombre de cellules de liste}
    // => { {résultat = cellule en position pos dans liste }
```

1.2. Dans la classe `ListeInt_Main` du projet `TP10` :

- ✓ Mettez en commentaire les instructions existantes, à l'exception de celles concernant la déclaration et l'initialisation de la liste `listeInt`
- ✓ Affichez la liste `listeInt` en utilisant `afficheGDdetaille`
- ✓ Déclarez un entier `posCell` et initialisez-le avec l'instruction suivante :
  `posCell = (int) (Math.random()*listeInt.getLongueur()+1);`
- ✓ Affichez avec `afficheCellInt` la cellule qui est en position `posCell` dans `listeInt`

1.3. Testez…

2. Création d'une liste à partir d'une sous-liste d'une liste existante

2.1. Dans la classe `Utilitaire` du projet `TP10`, ajoutez la fonction suivante :

```
public static ListeChainee<Integer> sousListe(ListeChainee<Integer> listeInt,
                                              int posDeb, int posFin) {
    // { posDeb <= posFin
    //   posDeb et posFin compris entre 1 et le nombre de cellules de listeInt }
    //  => { résultat = nouvelle liste constituée à partir des cellules de listeInt
    //       dont la position est dans l'intervalle [posDeb, posFin]
```

2.2. Dans la classe `ListeInt_Main` du projet `TP10` :

- ✓ Déclarez deux entiers `pos1` et `pos2` et initialisez-les (*par saisie ou par affectation*) avec des entiers dont la valeur est comprise entre 1 et le nombre d'éléments de `listeInt`

  NOTE : vous veillerez à ce que `pos1` soit inférieur ou égal à `pos2`
- ✓ Déclarez une nouvelle liste chaînée d'entiers `newListe` et initialisez-la par appel de la fonction `sousListe`
- ✓ Affichez le nombre d'éléments et le contenu de `newListe`

2.3. Testez…

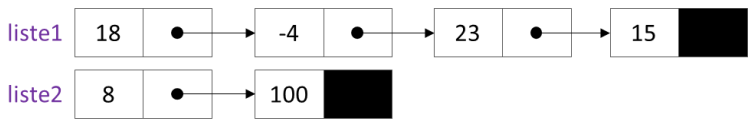
3. Insertion d'une liste dans une autre liste

3.1. Dans la classe Utilitaire du projet `TP10`, ajoutez la fonction suivante :

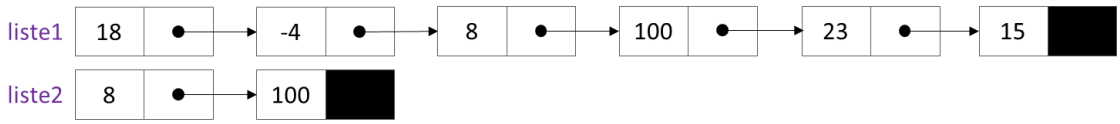
```
public static void insereL2DansL1(ListeChainee<Integer> l1,
                                   ListeChainee<Integer> l2, int pos) {
    // { } => { Autant de cellules qu'en comporte l2 ont été insérées dans l1
    //          * leur attribut info est identique à celui des cellules de l2
    //            et elles se succèdent selon l'ordre qu'elles avaient dans l2
    //          * la cellule en position pos dans l1 a pour cellule suivante
    //            la première de ces cellules
    //          * la dernière des cellules insérées a pour cellule suivante celle
    //            qui suivait initialement la cellule de position pos dans l1 }
```

EXEMPLE ILLUSTRÉ :

- Listes initiales



- Listes suite à l'insertion de liste2 dans liste1 après la cellule en position 2 dans liste1



3.2. Dans la classe `ListeInt_Main` du projet `TP10` :

- ✓ Déclarez et initialisez une nouvelle liste de 5 entiers, nommée `listeInt2`

  NOTE : vous pouvez initialiser cette liste avec des valeurs aléatoires (cf. TP10 1.1)
- ✓ Réaffichez le contenu de `listeInt`, puis affichez le contenu de `listeInt2`
- ✓ Insérez `listeInt2` dans `listeInt` après la cellule en position `pos1` dans `listeInt`
- ✓ Affichez le nombre d'éléments de `listeInt` et son contenu, suite à l'insertion

3.3. Testez…

4. Création d'une liste triée et sans doublons, par fusion de deux listes

4.1. Dans la classe Utilitaire du projet TP10, ajoutez la fonction suivante :

```
public static ListeChainee<Integer> fusionL1L2(ListeChainee<Integer> l1,
                                               ListeChainee<Integer> l2) {
    // { l1 et l2 non vides } =>
    //  {résultat = une liste triée dont les cellules portent les infos des cellules
    //               de l1 et celles des cellules de l2, sans doublons }
```

INDICATIONS : Pensez à utiliser les fonctions développées dans le TP10, partie B

4.2. Dans la classe `ListeInt_Main` du projet `TP10` :

- ✓ Déclarez et initialisez une nouvelle liste chaînée d'entiers, nommée `newListeT`
- ✓ Initialisez `newListeT` avec le résultat de la fonction `fusionL1L2` appliquée à `listeInt` et `listeInt2`
- ✓ Affichez le nombre d'éléments de `newListeT` et son contenu

4.3. Testez…

Partie B : Concept de pile & Application

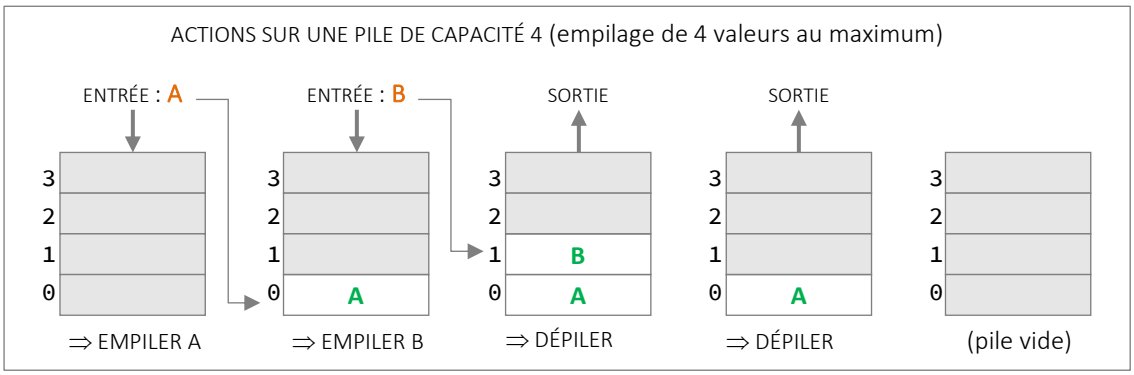
Dans cette partie, vous devrez :

- déclarer et manipuler des structures de type tableau
- déclarer et gérer des exceptions que vous aurez vous-mêmes déclarées

Le document Infos&Rappels_pourTP11_B, accessible depuis ce parcours, vous présente toutes les informations utiles.

LISEZ ATTENTIVEMENT CE DOCUMENT…

En informatique, une pile est une structure pour mémoriser des données, fondée sur le principe "dernier arrivé, premier sorti" (en anglais, on parle du principe LIFO : Last In, First Out).



La plupart des microprocesseurs et processeurs utilisent une pile.

Les premières calculatrices, parmi lesquelles les calculatrices Hewlet-Packard destinées aux calculs bancaires, utilisent une pile pour l'implémentation de la notation polonaise inversée (RPN : Reverse Polish NOTATION)

Préliminaires…

- Dans votre répertoire `R1.01`, créez un nouveau répertoire `TP11_Files` et exécutez l'instruction : `cp -r /users/info/pub/1a/R1.01/TP11_Files/*.* .`
- Sous IJ, créez un projet `TP11`
- Dans le répertoire `src` de ce projet ajoutez les 5 classes java de votre répertoire `TP11_Files`
- Étudiez le contenu de ces classes

B1. Classe PileFloat – Tests de fonctionnement d'une pile

1.1. Dans le projet `TP11`, ajoutez une classe `PileFloat` dont le squelette-vous est donné ci-dessous :

```
public class PileFloat {
    private int sommet; // indice de la dernière valeur empilée
    private final float[] pile; // tableau pour représenter une pile de float

    // constructeur
    public PileFloat (int capacite) {
        // {capacite = nombre de niveaux de la pile à construire}
        // => {cette pile est dimensionnée, sommet = -1 (aucune valeur empilée)}
        this.pile = new float[capacite];
        this.sommet = -1;
    }

    // getters
    int getSommet() {
        return sommet;
    }

    float getValPile(int pos) {
        // {pos compris entre 0 et nombre de niveaux de la pile} =>
        //    {résultat = valeur de pile à l'indice pos}
        return pile[pos];
    }

    //----- méthodes - À COMPLÉTER ------
    boolean estVide() {
        // {} => {résultat = vrai si sommet = -1}
        return true; // À REMPLACER
    }

    boolean estPleine() {
        //{} => {résultat = vrai si sommet = capacité}
        return true; // À REMPLACER
    }

    void empile(float val) throws EPilePleine {
        //{} => {si la pile n'est pas pleine, val a été insérée dans la pile et
        //       sommet est mis à jour, sinon, l'exception EPilePleine est levée
        //       avec un message approprié}

    }

    float depile() throws EPileVide {
        //{} => {si la pile n'est pas vide, la dernière valeur empilée a été retirée
        //       de la pile et sommet est mis à jour, sinon, l'exception EPileVide est levée
        //       avec un message approprié}
        return 0.0f; // À REMPLACER
    }
}
```

1.2. Dans la classe `PileFloat`, écrivez le code des méthodes `estVide`, `estPleine`, `empile` et `depile`

1.3. Dans le projet `TP11`, créez une classe `TestPile` et ajoutez-y une procédure `main` dans laquelle :

- ✓ vous déclarerez une variable de type `PileFloat` pouvant mémoriser 4 valeurs de type `float`
- ✓ vous ajouterez les instructions permettant de tester les différentes méthodes de la classe `PileFloat.`

Exemples de séquence de test :

- empiler 3 float, dépiler, dépiler, réempiler 2 float, essayer d'en empiler un de plus
- empiler 2 float, dépiler, dépiler, essayer de dépiler un float de plus

NOTE : après chaque dépilement, vous afficherez la valeur dépilée

B2. Simulation d'une calculatrice en notation polonaise inversée

Dans un temps pas si lointain, les calculatrices fonctionnaient selon le mode *Reverse Polish Notation* (RPN), appelée aussi *Notation Polonaise Inversée* (NPI) en français, ou encore *notation post-fixée*.

- Selon cette notation, les opérandes sont présentés avant les opérateurs.
  Exemple : pour calculer (3 + 4) x 5, il fallait taper un 3, appuyer sur ENTER, taper un 4, puis un +, puis un 5, puis un x.
- L'intérêt de cette méthode est de permettre des calculs complexes, sans pour autant utiliser beaucoup d'adresses mémoire et sans utiliser de parenthèses.

Les calculatrices basées sur cette méthode utilisent une **pile de 4 niveaux** et disposent d'une touche **ENTER** dont la fonction est d'empiler la valeur tapée par l'utilisateur sur le clavier de la calculette.

ILLUSTRATIONS

Calcul de : (3 + 4) x 5
en RPN : 3 ENTER 4 + 5 x

séquence	***3***	**ENTER**	***4***	**+**	***5***	**x**
écran	3	3	4	7	5	35

PILE							
?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?
?	3	3	3	7	7	7	35

Calcul de : 3 + (4 x 5)
en RPN : 3 ENTER 4 ENTER 5 x +

séquence	***3***	**ENTER**	***4***	**ENTER**	***5***	**x**	**+**
écran	3	3	4	4	5	20	23

PILE									
?	?	?	?	?	?	?	?	?	?
?	?	?	?	?	?	?	?	?	?
?	?	?	4	4	4	20	20	?	
?	3	3	3	3	3	3	3	23	

- *la ligne séquence décrit la succession des touches de la calculette pressées par l'utilisateur*
- *la ligne écran (en bleu) représente le contenu de l'écran de la calculette au fil des touches pressées*
- *dans le tableau PILE, les valeurs empilées sont sur fond vert et les valeurs dépilées sont sur fond orange*

EXPLICATIONS

- **ENTER** a pour effet d'empiler la dernière valeur **saisie** ou **empilée** (s'il est suivi d'un autre **ENTER**, cette valeur est à nouveau empilée, etc.)
- Les calculs se font toujours dans le sens « *plus ancien* OPERATION *plus récent* »
- Le choix d'un opérateur binaire OPB déclenche le calcul et l'affichage du résultat de l'opération :
  - ✓ **dernier empilé** OPB **nombre entré** si l'utilisateur a entré un nombre avant de sélectionner l'opérateur
    ⇒ *le dernier empilé est dépilé et le résultat du calcul est empilé*
  - ✓ **avant dernier empilé** OPB **dernier empilé** si l'utilisateur n'a pas entré de nombre avant de sélectionner l'opérateur
    ⇒ *l'avant dernier empilé et le dernier empilé sont dépilés et le résultat du calcul est empilé*

2.1. Dans votre nouveau projet, créez une classe `Calculette` où vous coderez les fonctions ou procédures nécessaires à la simulation du fonctionnement d'une calculatrice RPN EN RESPECTANT LES CONTRAINTES SUIVANTES :

C1. La calculatrice utilise une pile de **4 niveaux** dédiée à la mémorisation de **réels** (type `float`)

C2. Une séquence de calcul est représentée par une chaîne composée de *nombres* et de *mots*, ces *mots* représentant soit la touche ENTER, soit un opérateur arithmétique :

> Exemple de séquence de calcul : "3 ENTER 4 plus 5 multiple"
> *Formule mathématique représentée : (3 + 4) x 5 / Résultat attendu : 35*

C3. Les opérateurs devant être traités sont des opérateurs binaires :

OPÉRATEURS BINAIRES	MOTS QUI LES REPRÉSENTENT
+	plus
×	multiplie
/	divise
↑	exposant

NOTE : la notation ↑ est une notation proposée par *Knuth* pour représenter l'exponentiation[1]

C4. Chaque erreur susceptible d'être produite par une séquence de calcul donnera lieu au traitement des exceptions appropriées

2.2. Testez en exécutant chacune des séquences données dans le tableau ci-dessous :

Séquence (chaîne de caractères)	*Formule*	*Résultat attendu*
30 ENTER 2 plus 5 multiplie	(30 + 2) x 5	160.00
30 ENTER 2 ENTER 5 multiplie plus	30 + 2 x 5	40.00
30 ENTER 2 plus 5 ENTER 7 moins multiplie	(30 +2) x (5 – 7)	-64.00
30.5 ENTER 2 ENTER 0.5 divise plus	30.5 + 2/0.5	34.50
3.5 ENTER 4 ENTER 1 ENTER 7 divise plus multiplie	3.5 x (4 + 1/7)	14.50
1 ENTER 2 ENTER 3 ENTER 4 ENTER 5 ENTER 6 plus multiplie divise multiplie divise	*Tentative de calculer « dans l'ordre » :* 1 / (2 x (3 / (4 x (5 + 6))))	ERREUR (pile pleine)
4 ENTER 3 ENTER 1 ENTER 3 divise 9 multiplie moins divise	*Tentative de calculer* 4 / (3 – (1/3) x 9)	ERREUR (division par zéro)
2 ENTER 4 exposant	2^4	16.00
-3 ENTER 2 exposant	$(-3)^2$	9.00
4 ENTER -2 exposant	$(4)^{-2}$ *ou* $1/4^2$	0.0625
4 ENTER 1 ENTER 2 divise exposant	$\sqrt{4}$ *ou* $4^{1/2}$	2.00
-9 ENTER 1 ENTER 2 divise exposant	*Tentative de calculer la racine carrée de* -9	ERREUR (exposant non entier appliqué à un négatif)
2.5 ENTER 2 exposant 1.4 ENTER 2 exposant moins	$2.5^2 – 1.4^2$	4.29
2.5 ENTER 1.4 plus 2.5 ENTER 1.4 moins multiplie	(2.5 +1.4) x (2.5 – 1.4)	4.29
80 ENTER 2 divise 24 ENTER 6 divise moins 2 exposant	$(80/2 – 24/6)^2$	1296.0