Université de Nice-Sophia Antipolis — vendredi 31 mai 2013
POLYTECH PeiP2

DS d'Algorithmique et Programmation Java

Durée: 1h30
Documents non autorisés

Note : Les 4 parties de ce DS sont indépendantes. La qualité des commentaires, avec notamment la présence d'affirmations significatives, ainsi que les noms donnés aux variables, l'emploi à bon escient des majuscules et la bonne indentation rentreront pour une part importante dans l'appréciation du travail.

---

1 Pile

```
/**
 *  Rôle : empile dans la Pile courante l'objet x
 *
 *  @param x l'objet de type T générique à empiler
 */
public void empiler(T x) {
    int INCR = 20; // incrément d'extension de la pile
    if (estPleine()) {
        // agrandir le tableau
        T [] nouveau = (T []) new Object[lesÉléments.length+INCR];
        // recopier les valeurs de la pile dans le nouveau tableau
        for (int i=0; i<lesÉléments.length; i++)
            nouveau[i] = lesÉléments[i];
        lesÉléments = nouveau;
    }
    // il y a de la place pour x
    lesÉléments[sp++] = x;
}
```

---

2 Le minimum excludant d'un ensemble (mex)

```
public static int mex(Liste<Integer> l) {
    int size = l.longueur();
    // pour toute valeur i
    // on part de 0 jusqu'au mex qui au plus vaudra size
    for (int i=0; i < size; i++) {
        int j;
        // on regarde tous les éléments aux indices j de la liste
        for(j=1; j <= size; j++)
            if (l.ième(j)==i)
                // on a trouvé la valeur i, on passe à la suivante
                j=size+2;
        // si on atteint la fin du tableau,
        // c'est que c'est la 1ère valeur manquante et on la retourne
```

```
        if(j==size+1) return i;
    }
    return size;
}

public static int mexOrdonne(Liste<Integer> l) {
    int size = l.longueur();

    if (size==0) return 0;

    int prev = l.ième(1);
    int next = prev;
    if(prev != 0) return 0;
    // quand la liste est ordonnée, un seul parcours est nécessaire
    for(int i=2; i <= size; i++) {
        prev = next;
        next = l.ième(i);
        if((next!=prev) && (next-prev!=1))
            return prev+1;
    }
    return next+1;
}
```

3 Tri

```
/**
 *  Rôle : la méthode estTriée retroune vrai si
 *         la liste l est ordonnée de façon croissante,
 *         et faux sinon.
 *
 *  @param : l la liste à tester
 */
public static boolean estTriée(Liste<Integer> l) {
    if (l.longueur()==0) return true;
    for (int r=2; r<= l.longueur(); r++)
        if (l.ième(r)<l.ième(r-1))
            // la liste n'est donc pas ordonnée
            return false;
    // la liste est ordonnée
    return true;
}

/**
 *  Rôle : trie la liste l en un temps non borné.
 *
 *   @param : l la liste à trier
 */
public static void trier(Liste<Integer> l) {
    // générateur de nombres aléaoires
    Random rand = new  Random();
    while (!estTriée(l))
        // la liste n'est pas triée =>
```

```
        // tirer 2 indices au hasard et
        // échanger les 2 éléments associés
        l.échanger(rand.nextInt(l.longueur())+1,
                   rand.nextInt(l.longueur())+1);
    // la liste est triée
}
```

Cet algorithme ne peut trier la liste ne un temps borné, mais la probabilité d'obtenir *in fine* une liste triée n'est pas nulle.

Les tests montrent que le nombre d'éléments maximun pour trier dans un temps raisonnable la liste est 10 ou 11.

---

4 Arbre binaire

```
/**
 *  Rôle : renvoie true si l'arbre courant est une feuille
 *         c-à-d, qu'il n'a ni sous arbre gauche, ni sous arbre droit,
 *         et faux sinon
 */
public boolean estFeuille() {
    return this.sag().estVide() && this.sad().estVide();
}

/**
 *  Rôle : renvoie le nombre de feuilles de l'arbre courant
 *         grâce à un parcours récursif en profondeur
 */
public int nbFeuilles() {
    if (this.estVide()) return 0; // l'arbre courant est vide, donc sans feuille
        if (this.estFeuille()) return 1; // l'arbre courant est une feuille
        else // déclencher les appels récursifs sur les sous-arbres
             return this.sag().nbFeuilles() + this.sad().nbFeuilles();
}

/**
 *  Rôle : numérote les feuilles de l'arbre courant de gauche à droite
 *  à partir du numéro indiqué
 *
 *  @param num l'entier à partir duquel numéroter les feuilles
 *   @return l'entier à partir duquel l'arbre parent peut continuer à numéroter
 *      (sert pour numéroter les feuilles du sous arbre droit de l'arbre parent)
 */
public int numéroterFeuilles(int num) {
    if (this.estVide()) return num; // arrêt des appels récursifs
    if (this.estFeuille()) {// l'arbre courant est une feuille donc numéroter
        this.setNuméro(num);
        num++; // préparer la prochaine numérotation
        return num;
   }
   // else déclencher récursivement numérotation
   // des feuilles des 2 sous-sarbres
   int nouvNum = this.sag().numéroterFeuilles(num);
```

```
        nouvNum = this.sad().numéroterFeuilles(nouvNum);
        return nouvNum;
}
```