Université de Nice-Sophia Antipolis — POLYTECH
PeiP2 — 2017–2018

Examen de Algorithmique et Structures de données

Durée : 2h
Aucun document autorisé
Mobiles interdits

Note : 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.

Vous choisirez de traiter uniquement 2 parties parmi les 3 suivantes

1 Listes

▶ 1. Quels sont les 3 résultats affichés par l'exécution de l'algorithme suivant :

```
Pile p = pilevide
  empiler(p,3)
  empiler(p,18)
  écrire(sommet(p))
  dépiler(p)
  empiler(p, sommet(p))
  écrire(sommet(p))
  dépiler(p)
  écrire(estVide(p))
```

```
18 3 faux
```

▶ 2. À l'aide de la classe `Stack`, programmez en Java dans une méthode `main` l'algorithme précédent.

```
public static void main(String [] args) {
    Stack<Integer> p = new Stack<Integer>();
    p.push(3);
    p.push(18);
    System.out.println(p.peek());
    p.pop();
    p.push(p.peek());
    System.out.println(p.peek());
    p.pop();
    System.out.println(p.empty());
}
```

▶ 3. Une file d'attente est une suite d'éléments accessible uniquement par ses deux extrémités. On ajoute un élément par un coté (la queue) et on retire un élément par l'autre coté (la tête). C'est le comportement FIFO (First-In – First-Out).

Les *files avec priorité* remettent en question le modèle FIFO des files d'attentes. Avec ces files, l'ordre d'arrivée des éléments n'est plus respecté. Les éléments sont munis d'une priorité et ceux qui possèdent les priorités les plus fortes sont traités en premier. On définit l'ensemble $\mathcal{F}ile$ des files d'attente formées d'éléments pris dans l'ensemble $\mathcal{E}$, sur lequel les opérations suivantes sont définies :

estVide	:	$\mathcal{F}ile{-}\mathcal{P}riorité$	$\rightarrow$	booléen
ajouter	:	$\mathcal{F}ile{-}\mathcal{P}riorité \times \mathcal{E} \times \mathcal{P}riorité$	$\rightarrow$	$\mathcal{F}ile{-}\mathcal{P}riorité$
premier	:	$\mathcal{F}ile{-}\mathcal{P}riorité$	$\rightarrow$	$\mathcal{E}$
supprimer	:	$\mathcal{F}ile{-}\mathcal{P}riorité$	$\rightarrow$	$\mathcal{F}ile{-}\mathcal{P}riorité$

L'opération estVide teste si la file avec priorité est vide ou pas, l'opération ajouter insère dans la file un nouvel élément avec sa priorité, l'opération premier renvoie l'élément le plus prioritaire de la file, et enfin l'opération supprimer retire l'élément le plus prioritaire de la file.

Écrivez en Java l'interface générique `FilePriorité<V,P>` qui déclare les opérations précédentes. Le type `V` représente la valeur des éléments de la file et `P` leurs priorités.

```
public interface FilePriorité<V,P  extends Comparable<P>> {
    public boolean estVide();
    public V premier() throws FileVideException;
    public void supprimer() throws FileVideException;
    public void ajouter(V e, P p);
}
```

▶ 4. Écrivez en Java la classe générique `FilePrioritéListe<V,P>` qui implémente les 4 opérations précédentes.

Pour cela, vous utiliserez une liste de type `LinkedList<E>` de l'API que vous maintiendrez ordonnée de façon décroissante sur les priorités pour avoir l'élément de plus forte priorité en tête de liste. Vous pourrez utiliser la classe `Élément` suivante pour regrouper la valeur et la priorité à mémoriser dans la `LinkedList`.

```
class Élément<V, P extends Comparable<P>> {
    private V valeur;
    private P priorité;
    Élément(V v, P p) {
        this.valeur = v;
        this.priorité = p;
    }
    V valeur() { return this.valeur; }
    P priorité() { return this.priorité; }
}
```

```
import java.util.*;

public class FilePrioritéListe<V, P extends Comparable<P>>
    implements FilePriorité<V,P> {

    protected LinkedList<Élément<V,P>> lp;

    public FilePrioritéListe() {
        lp = new LinkedList<Élément<V,P>>();
    }

    public boolean estVide() { return lp.size()==0; }

    public V premier() throws FileVideException {
```

```
        if (estVide()) throw new FileVideException();
        return lp.getFirst().valeur();
    }
    public void supprimer() throws FileVideException {
        if (estVide()) throw new FileVideException();
        lp.removeFirst();
    }

    public void ajouter(V v, P p) {
        int i=0;
        while (i<lp.size() && p.compareTo(lp.get(i).priorité())<0) i++;
        lp.add(i, new Élément<V,P>(v,p));
    }
}
```

.....

▶ 5. Donnez et expliquez les complexités des opérations *ajouter*, *premier* et *supprimer*.

Dans la mesure où l'élément de plus forte priorité est en tête de liste, la complexité des opérations *premier* et *supprimer* est égale à $\mathcal{O}(1)$. En revanche, la complexité de l'opération *ajouter* est $\mathcal{O}(n)$ puisqu'il faut maintenir la liste linéaire ordonnée.

.....

---

2 Tri

Dans cet exercice, on se propose d'implémenter et analyser le *tri fusion* (*merge sort*). Ce tri est un tri interne qui procède par interclassement de sous-listes triées. L'algorithme de ce tri s'exprime bien récursivement. On divise la liste à trier en deux sous-listes de même taille que l'on trie récursivement par fusion. Les deux sous-listes triées sont ensuite fusionnées par interclassement.

On fournit ci-dessous la version algorithmique de la méthode de fusion par interclassement, qui prend en entrée une liste et trois indices *début*, *milieu* et *fin* délimitant deux sous-listes conjointes, supposées chacune triée. La fonction *fusionnerSousListes* réordonne les éléments des sous-listes [*début,milieu*[et [*milieu,fin*[pour les rassembler en une unique sous-liste triée, [*début,fin*[.

```
procédure fusionnerSousListes(
          lst : liste d'éléments de T comparables,
          début : 1er élément de la 1ère sous-liste,
          milieu : 1er élément de la 2ème sous-liste,
          fin : élément après la fin de la 2ème sous-liste)

   tmp ←nouvelle liste vide
   i ← début
   j ← milieu
   tantque i < milieu et j < fin faire
        si clé(ième(lst, i)) ⩽ clé(ième(lst, j)) alors
             ajouter(tmp, ième(lst, i))
             i ← i + 1
        sinon
             ajouter(tmp, ième(lst, j))
             j ← j + 1
        finsi
    fintantque
    tantque i < milieu faire
         ajouter(tmp, ième(lst, i))
         i ← i + 1
    fintantque
```



```
    tantque j < fin faire
         ajouter(tmp, ième(lst, j))
         j ← j + 1
    fintantque
    pourtout k de 1 à taille(tmp) faire
            ième(lst, début+k-1) ← ième(tmp,k)
    finpour
finproc
```

▶ 6. Décrivez, de façon claire et synthétique, comment procède cette méthode : que fait-elle sur chaque sous-liste, comment les parcourt-elle ? On rappelle que dans la notation algorithmique, le rang du premier élément d'une liste est 1.

La méthode commence par itérer sur les deux sous-listes en alternance, en transférant à chaque fois dans un liste temporaire le plus petit des éléments courants de la premiere et la seconde sous-liste. Chaque sous-liste étant triée, l'ordre dans la liste résultante est préservé. La boucle cesse une fois la fin de la plus petite sous-liste atteinte. Dans le cas où les sous-listes ne sont pas de même taille, elle transfère ensuite tous les éléments restants. Finalement, la liste temporaire est transférée dans la liste d'entrée.

.....

▶ 7. Écrivez la méthode `fusionner` correspondant à l'algorithme précédent. Cet méthode possède l'en-tête suivant :

```
public static <C extends Comparable<C>, V>
  void fusionner(List<Element<C,V>> l, int début, int   milieu, int fin)
```

```
public static <C extends Comparable<C>, V>
void fusionner(List<Element<C,V>> l, int debut, int milieu, int fin)
{
  List<Element<C,V>> l1 = new ArrayList<Element<C,V>>();
  int i = debut;
  int j = milieu;
  while(i<milieu && j<fin)
    if (l.get(i).cle().compareTo(l.get(j).cle()) <= 0)
         l1.add(l.get(i++));
    else
         l1.add(l.get(j++));
  //
  while(i<milieu) l1.add(l.get(i++));
  while(j<end) l1.add(l.get(j++));
  for(int k=0; k<l1.size(); ++k) l.set(start+k, l1.get(k));
}
```

.....

▶ 8. Écrivez maintenant la méthode de tri fusion, qui divise la liste en deux listes de taille égale, trie les sous-listes récursivement puis les fusionnent avec la méthode précédente. La méthode `tri_fusion` qui tri les éléments de la liste l, compris entre les indices *debut* et *fin-1*, possède l'en-tête suivant :

```
public static <C extends Comparable<C>, V>
  void tri_fusion(List<Element<C,V>> l, int début, int fin).
```

```
public static <C extends Comparable<C>, V>
void tri(List<Element<C,V>> l, int debut, int fin) {
    if (fin-debut <= 1)
```

```
        return; //Un seul élément => déjà trié.
      //else
    int milieu = (debut+fin)/2;
    tri(l, debut, milieu);
    tri(l, milieu, fin);
    fusionner(l, debut, milieu, fin);
}
```

.....

▶ 9. Exprimez la complexité (en nombre de comparaisons effectuées) de l'algorithme de tri fusion pour une liste de taille n, en fonction de la complexité pour des listes de tailles inférieures.

Le tri effectue deux appels récursifs sur des listes de taille $\frac{n}{2}$, ayant chacun une complexité $\mathcal{O}(\frac{n}{2})$, plus une fusion sur des sous-listes de taille $\frac{n}{2}$. A chaque itération du premier `while` de la fusion, on effectue une comparaison et on place un élément dans la liste finale. Celle-ci étant de taille n, elle sera remplie après, au plus, n itérations et n comparaisons (on peut ignorer les `while` suivants, sans comparaisons). Donc, la complexité est égale à $\mathcal{O}(2 * \mathcal{O}(\frac{n}{2}) + n)$.

.....

---

3 Arbre binaire

▶ 10. Donnez la relation qui lie la profondeur p et le nombre de nœuds n d'un arbre binaire. En déduire la complexité minimale et maximale d'une opération sur une branche d'arbre binaire.

La profondeur p et le nombre de nœuds$n$ d'un arbre sont tels que $\lfloor \log_2 n \rfloor \leqslant p \leqslant n-1$, où $\lfloor \rfloor$ désigne la partie entière inférieure.

La compléxité maximale sera donc linéaire, et celle minimale sera logarithmique (en base 2).

.....

▶ 11. Quelles sont les 4 opérations de base sur un arbre binaire vues en TD ? Écrivez en Java l'interface générique `ArbreBinaire<T>`.

```
public interface ArbreBinaire<T> {
    public boolean estVide();
    public T valeur() throws ArbreVideException;
    public void setValeur(T e) throws ArbreVideException;
    public ArbreBinaire<T> sag() throws ArbreVideException;
    public ArbreBinaire<T> sad() throws ArbreVideException;
}
```

.....

▶ 12. Il est possible de dénoter de façon univoque les nœuds d'un arbre binaire par des mots formés d'une suite de 0 et de 1 obtenus en parcourant le chemin qui mène de la racine à ce nœud. Par définition, la racine est le mot vide $\varnothing$, et si un nœud est dénoté par le mot m son fils gauche est $m0$ et son fils droit $m1$.

Écrivez en Java la méthode `marquerArbre` qui prend en paramètre un arbre binaire de type `ArbreBinaire<String>` et qui marque chacun des nœuds selon la convention précédente. On considérera que l'interface `ArbreBinaire<T>` possède, en plus, la méthode `setValeur` qui affecte au nœud courant la valeur `e`. Cette méthode possède l'en-tête suivant :

```
public void setValeur(T e) throws ArbreVideException;
```

```
public static void marquerArbre(ArbreBinaire<String> a)
throws ArbreVideException
{
   if (!a.estVide()) marquerArbre(a,"");
}

private static void marquerArbre(ArbreBinaire<String> a, String m)
throws ArbreVideException
{
   if (!a.estVide()) {
       a.setValeur(m);
       marquerArbre(a.sag(), m+"0");
       marquerArbre(a.sad(), m+"1");
   }
}
```

.....