Université de Nice-Sophia Antipolis — POLYTECH
PeiP2 — 2016–2017

Examen de Algorithmique et Programmation JAVA

Durée : 1h30
Aucun document autorisé
Mobiles interdits

Nom : Prénom : Groupe :

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.

---

1 Recherche

▶ 1. Écrivez de façon algorithmique et récursive la fonction `rechercheDichoR` qui recherche un élément `E` de clé `c` de façon dichotomique dans une liste linéaire ordonnée `lst`. Si la clé n'est pas trouvée, il faudra évidement le signaler. Vous donnerez la complexité de cette recherche dans le pire des cas. Rappel :

longueur : $\mathcal{L}_o \rightarrow naturel$
ième : $\mathcal{L}_o \times naturel \rightarrow \mathcal{E}$

L'en-tête de cette fonction est le suivant :

```
{ Antécédent : à compléter
  Rôle : ....
}
fonction rechercheDichoR(lst, c, gauche, droite) : E
```

```
{ Antécédent : longueur(lst)≥1
  Rôle : recherche dichotomique, écrite de façon récursive
}
fonction rechercheDichoR(lst, c, gauche, droite) : E
  si gauche>droite alors exception CléNonTrouvée
  sinon
      { gauche ⩽ droite et ∀ k, 1 ⩽k <gauche, clé(ième(lst,k))<c et
        ∀k, droite < k ⩽longueur(lst), clé(ième(lst,k)) > c
      }
      milieu←(gauche+droite)/2
      x←ième(lst, milieu)
      si c=clé(x) alors rendre x
      sinon
          si c<clé(x) alors
              rendre rechercheDichoR(lst, c, gauche, milieu-1)
          sinon { c>clé(x) }
               rendre rechercheDichoR(lst, c, milieu+1, droite)
          finsi
      finsi
  finsi
finfonc
```


2 Liste ordonnée

▶ 2. On définit récursivement l'ensemble des listes ordonnées $\mathcal{L}_o$ comme : $\mathcal{L}_o = \emptyset \mid \mathcal{E} \times \mathcal{L}_o$, avec $\mathcal{E}$ l'ensemble des éléments (valeur + clé) de la liste muni d'une relation d'ordre sur la clé. Donnez les axiomes de l'opération supprimer qui supprime un élément de clé c dans une liste ordonnée l. On considérera la relation d'ordre « inférieur ou égal ». La signature de l'opération supprimer est la suivante :

supprimer : $\mathcal{L}_o \times \mathcal{C}lé \rightarrow \mathcal{L}_o$

---

1. $\nexists l,\ l = \text{supprimer}(\text{listevide}, c)$
2. $\text{clé}(e) = c \Rightarrow \text{supprimer}(< e, l >, c) = l$
3. $\text{clé}(e) < c \Rightarrow \text{supprimer}(< e, l >, c) =< e, \text{supprimer}(l, c) >$
4. $\text{clé}(e) > c \Rightarrow \nexists l',\ l' = \text{supprimer}(< e, l >, c)$

3 Tri

Dans cet exercice, on se propose d'étudier un algorithme de tri appelé tri à bulles (*bubble sort*). Le tri à bulles permute les éléments mal ordonnés, en faisant « *remonter* » les éléments les plus petits vers le début de la liste. On présente ci-dessous une version algorithmique et itérative de cette méthode de tri.

```
fonction triBulles(lst : liste d'éléments comparables)
   faire
        échangés ← faux
        pourtout i de longueur(lst)-1 à 1 faire
            si clé(ième(lst, i+1)) < clé(ième(lst, i)) alors
               échanger(ième(lst, i+1), ième(lst, i))
               échangés ← vrai
            finsi
        finpour
   tantque échangés
finfonc
```

▶ 3. Proposez une implémentation en Java.

```
public static  <C extends Comparable<C>, V> void tri(List<Élément<C,V>> lst){
  boolean échangés;
  do {
     échangés = false;
     for (int i = lst.size()-2; i >= 0 ;--i)
          if (lst.get(i+1).cle().compareTo(lst.get(i).cle()) < 0) {
              Élément<C,V> temp = lst.get(i+1);
              lst.set(i+1, lst.get(i));
              lst.set(i, temp);
              échangés = true;
          }
  } while(échangés);
}
```

▶ 4. Quelle est la complexité (en nombre de comparaisons effectuées) de cet algorithme sur une liste de longueur n si :
— la liste est déjà triée dans l'ordre croissant ;
— la liste est triée dans l'ordre décroissant.

Si la liste est déjà triée, on fait une unique passe sur toute la liste, comparant chaque élément avec le précédent, la complexité est en $\mathcal{O}(n)$.

Si la liste est triée dans l'ordre décroissant, à chaque itération de la boucle extérieur un seul élément est placé correctement en « *remontant* la liste, en effectuant à chaque fois $n-1$ comparaisons. Soit un total de $(n-1)^2$ comparaisons et une complexité égale à $\mathcal{O}((n^2))$.

▶ 5. En admettant que sa complexité moyenne soit similaire à sa complexité dans le pire des cas, comment le tri à bulles se compare-t-il au tri par insertion et au tri rapide, dont vous rappellerez les complexités moyennes (toujours en nombre de comparaisons effectuées) ?

Tri par insertion : $\mathcal{O}(n^2)$.
Tri rapide : $\mathcal{O}(n \log(n))$.
Le tri à bulles est d'une complexité comparable à celle du tri par insertion, mais est bien moins performant que le tri rapide.

▶ 6. À la fin de la k-ème itération de la boucle **tantque**, quels sont les éléments bien triés à coup sûr ? Utilisez cette observation pour proposer une amélioration de l'algorithme. À votre avis, dans quelle mesure la complexité (en nombre de comparaisons) de la méthode est-elle modifiée ?

Les k plus petits éléments ont été correctement placés en début de liste. On peut donc modifier la valeur limite de la boucle **pourtout**, en fonction du dernier élément déplacé à à chaque itération de la boucle **tantque**.

À la fin d'une itération, tous les éléments à la gauche de celui-ci sont bien placés, on n'aura plus à les considérer dans la suite.

```
fonction triBulles(lst : liste d'éléments comparables)
    m ← 1
    faire
        dernier ← longueur(lst)
        pourtout i de longueur(lst)-1 à m faire
            si clé(ième(lst, i+1)) < clé(ième(lst, i)) alors
                échanger(ième(lst, i+1), ième(lst, i))
                dernier ← i
            finsi
        finpour
        m ← dernier + 1
    tantque m < longueur(lst)
```

Dans ce dernier cas le nombre de comparaisons effectuées peut être diminué de moitié ; on restera cependant avec une complexité quadratique. On peut de plus remarquer que le nombre d'échanges ne sera pas modifié.

4 Arbre Binaire

On rappelle qu'un arbre binaire implémente l'interface suivante :


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

▶ 7. Écrivez en Java une méthode `échangerSousArbre` qui échange le sous-arbre gauche et le sous-arbre droit de l'arbre courant.

```
public void échangerSousArbre() {
        ArbreBinaire<T> sa = sag;
        sag = sad;
        sad = sa;
}
```

▶ 8. En TD, nous avons développé une méthode qui calcule la hauteur d'un arbre binaire. En utilisant cette méthode sans la récrire, écrivez en Java la méthode `trierSaParHauteur` qui transforme l'arbre courant de telle sorte que pour tout noeud de cet arbre, la hauteur du sous-arbre gauche soit plus petite que celle du sous-arbre droit (c'est-à-dire que s'il existe un noeud de l'arbre qui ne vérifie pas cette propriété, la fonction doit inverser les deux sous-arbres de ce noeud). Remarque : ne pas gérer l'exception `ArbreVideException` dans un **try-catch**.

```
public void trierSaParHauteur() throws ArbreVideException {
        trierSaParHauteur(this);
}

private void trierSaParHauteur(ArbreBinaire<T> a)
throws ArbreVideException
{
      //condition d'arrêt
      if (a.estVide()) return;
      //algorithme sur le noeud courant
        if (hauteur(a.sag()) > hauteur(a.sad()))
              échangerSousArbre();
        //appels récursifs
        trierSaParHauteur(a.sag());
        trierSaParHauteur(a.sad());
}
```