

Contrôle de Algorithmique et Structures de données

Durée : 2h

Aucun document autorisé

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)
ecrire(sommet(p))
depiler(p)
empiler(p, sommet(p))
ecrire(sommet(p))
depiler(p)
ecrire(estVide(p))
```

- 2. À l'aide de la classe Stack, programmez en JAVA dans une méthode main l'algorithme précédent.
- 3. Une file d'attente est une suite d'éléments accessible uniquement par ses deux extrémités. On ajoute un élément par un côté (la queue) et on retire un élément par l'autre côté (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 *File* des files d'attente formées d'éléments pris dans l'ensemble $\mathcal{E}$, sur lequel les opérations suivantes sont définies :

estVide	: <i>File</i> – <i>Priorité</i>	→ booléen
ajouter	: <i>File</i> – <i>Priorité</i> $\times \mathcal{E} \times$ <i>Priorité</i>	→ <i>File</i> – <i>Priorité</i>
premier	: <i>File</i> – <i>Priorité</i>	→ $\mathcal{E}$
supprimer	: <i>File</i> – <i>Priorité</i>	→ <i>File</i> – <i>Priorité</i>

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.

- 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; }
}
```

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

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.

- 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)
```

- 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 *début* 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).
```

- 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.
-

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.
- 11. Quelles sont les 4 opérations de base sur un arbre binaire vues en TD? Écrivez en JAVA l'interface générique `ArbreBinaire<T>`.
- 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 $\emptyset$, 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;
```