Nom ______________________________

Prénom ______________________________

Code permanent ______________________________

Solution de l'examen final

Date : 21 avril 2017

Titre du cours : Structures de données

Sigle et groupe : INF7341-40

Enseignant : Alexandre Blondin Massé

Instructions

1) Vous avez trois heures pour répondre à l'examen ;
2) Vous pouvez répondre aux *cinq* questions de l'examen ;
3) Les *quatre* meilleurs résultats seront comptabilisés ;
4) Vous avez droit à toute votre documentation ;
5) Il est interdit d'utiliser un ordinateur, peu importe sa taille et sa forme (téléphone portable, agenda électronique, etc.) ;
6) Il est interdit de parler et de prêter de la documentation à un autre étudiant ;
7) Au besoin, utilisez le verso comme brouillon ;
8) À moins d'avis contraire, justifiez toutes vos réponses et donnez le détail de vos calculs ;
9) Indiquez clairement vos réponses finales ;

Question	1	2	3	4	5	Total
Sur	25	25	25	25	25	100
Note						

Question 1. .... (25 points)

Il est possible d'implémenter la structure de données File à l'aide de deux piles. Dans cette question, pour simplifier l'écriture du code, on suppose qu'une pile p offre une opération p.DEPILER() qui retourne l'élément sur la tête de la pile *et* qui supprime cet élément de la pile (plutôt que de séparer les deux opérations comme nous l'avons fait en classe). De la même façon, on n'implémentera que l'opération DÉFILER() de la file plutôt que de diviser les opérations en deux opérations DÉFILER() et PREMIER().

Le constructeur d'une file vide f construit tout simplement deux piles vides qu'on appellera f.GAUCHE() et f.DROITE(). Pour enfiler, il suffit d'utiliser le pseudocode suivant :

```
1: procedure ENFILER(f : File, e : Element)
2:     f.GAUCHE().EMPILER(e)
3: fin procedure
```

L'opération de défilement est la plus complexe :

```
1: fonction DÉFILER(f : File) : Element
2:     si f.DROITE().ESTVIDE() alors
3:         tant que ¬f.GAUCHE().ESTVIDE() faire
4:             f.DROITE().EMPILER(f.GAUCHE().DÉPILER())
5:         fin tant que
6:     fin si
7:     retourner f.DROITE().DÉPILER()
8: fin fonction
```

(a) (10 points) Illustrez l'évolution de la structure de données si on effectue les opérations suivantes :

1. On crée une file vide ;
2. On enfile les éléments a, b et c ;
3. On défile un élément ;
4. On enfile les éléments d et e ;
5. On défile deux éléments.
6. On défile un élément.

Autrement dit, donnez la trace de cet algorithme en dessinant, après les étapes 1, 2, 3, 4, 5 et 6 le contenu des deux piles.

Solution:

Étape	Pile gauche (haut → bas)	Pile droite (haut → bas)
1		
2	c, b, a	
3		b, c
4	e, d	b, c
5	e, d	
6		e

(b) (15 points) Montrez que les opérations f.ENFILER et f.DEFILER ont une complexité amortie de $\Theta(1)$ à l'aide de la méthode du potentiel si la file est initialement vide. *Indice :* La fonction potentielle d'une file f est $\Phi(f) = |f.\text{GAUCHE}()|$, c'est-à-dire le nombre d'éléments dans la première pile.

Solution: Commençons par f.ENFILER. Soit f' la file obtenue après enfilement, $\hat{c}$ le coût amorti de l'enfilement et c son coût réel. Aussi, dénotons par g et g' le nombre d'éléments dans les piles f.GAUCHE() et f'.GAUCHE(). Alors

$$\begin{aligned} \hat{c} &= c + \phi(f') - \phi(f) \\ &= 1 + g' - g \\ &= 1 + (g + 1) - g \\ &= 2, \end{aligned}$$

qui est bien $\mathcal{O}(1)$.

Maintenant, étudions l'opération f.DEFILER. Utilisons la même notation qu'auparavant, c'est-à-dire définissons f', $\hat{c}$, c, g et g' de façon similaire pour l'opération de défilement. On doit considérer deux cas :

(1) Supposons d'abord que $g \neq 0$. Alors on dépile tout simplement l'élément de f.DROITE(), de sorte que

$$\begin{aligned} \hat{c} &= c + \phi(f') - \phi(f) \\ &= 1 + g' - g \\ &= 1 + g - g \\ &= 1, \end{aligned}$$

qui est $\mathcal{O}(1)$.

(2) Supposons maintenant que $g = 0$. Alors on doit dépiler les éléments de f.GAUCHE() et les empiler sur f.DROITE() jusqu'à ce que la pile gauche soit vide. On obtient donc

$$\begin{aligned} \hat{c} &= c + \phi(f') - \phi(f) \\ &= g + g' - g \\ &= g + 0 - g \\ &= 0, \end{aligned}$$

qui est aussi $\mathcal{O}(1)$.

Question 2. .... (25 points)

Soit T un arbre binaire de n noeuds. La *hauteur minimale* de T, dénotée par $h_{\min}(T)$ est le nombre minimum de noeuds qu'on rencontre lorsqu'on commence à la racine et qu'on se dirige vers une feuille.

(a) (5 points) Est-ce vrai que $h_{\min}(T) \in \mathcal{O}(n)$? (Aucune justification demandée)

(a) **Vrai**

(b) (5 points) Est-ce vrai que $h_{\min}(T) \in \Omega(\log n)$? (Aucune justification demandée)

(b) **Faux**

(c) (15 points) Donnez le pseudocode d'un algorithme *récursif* qui calcule la hauteur minimale de T.

Solution: Il suffit de prendre la définition récursive de la hauteur d'un arbre binaire et de remplacer la fonction max par min. On trouve

1: **fonction** HauteurMinimale(T : arbre binaire) : naturel
2: **si** T.estVide() **alors**
3: **retourner** 0
4: **sinon**
5: h_G = HauteurMinimale(T.gauche())
6: h_D = HauteurMinimale(T.droit())
7: **retourner** $1 + \min(h_G, h_D)$
8: **fin si**
9: **fin fonction**

Question 3. .... (25 points)

Nous avons vu en classe la structure d'ensembles disjoints E, qui offrent minimalement les opérations suivantes :

- E.AjouterSingleton(e), qui ajoute un élément seul dans sa classe.
- E.Fusionner(e, f), qui fusionne les classes de e et de f.
- E.Representant(e), qui retourne un représentant de la classe de e. Ce représentant est le même qui est retourné pour tout autre élément f dans la classe de e.

Nous souhaitons maintenant ajouter une quatrième opération

- E.Afficher(e), qui affiche tous les éléments de la classe de e, dans n'importe quel ordre.

En donnant le pseudocode des quatre opérations, montrez qu'il est possible d'étendre la structure de données d'ensembles disjoints en préservant la même complexité *spatiale* et les mêmes *complexités amorties* pour les opérations AjouterSingleton, Fusionner et Representant, ainsi qu'une complexité de $\Theta(|e|)$ pour l'opération Afficher, où $|e|$ est le nombre d'éléments dans la même classe que e.

Indice 1 : Il suffit d'ajouter un attribut à chaque noeud qui retient la liste de ses enfants et qui doit être mise à jour au fur et à mesure. *Indice 2 :* Afin de conserver les mêmes complexités, il n'est pas nécessaire que l'attribut *enfants* correspondent à la réalité, mais il peut faire abstraction de la compression de chemins.

Solution: Il suffit de reprendre les trois fonctions et d'apporter les modifications suivantes :

- Dans AjouterSingleton, on initialise les enfants par une liste vide ;
- Dans Fusionner, on met à jour la liste d'enfants de la nouvelle racine.
- Dans Représentant, on ne met pas à jour la liste d'enfants de la nouvelle racine. Bref, on fait comme s'il n'y avait pas de compression de chemins.

```
1:  procedure AjouterSingleton(x : élément)
2:      x.parent ← x
3:      x.rang ← 0
4:      x.enfants ← [ ]
5:  fin procedure

6:  procedure Fusionner(x, y : éléments)
7:      x ← Représentant(x)
8:      y ← Représentant(y)
9:      si x.rang > y.rang alors
10:         y.parent ← x
11:         y.enfants.Ajouter(x)
12:     sinon
13:         x.parent ← y
14:         x.enfants.Ajouter(y)
15:         si x.rang = y.rang alors
16:             y.rang ← y.rang + 1
```

```
17:         fin si
18:     fin si
19: fin procedure

20: fonction Représentant(x : élément)
21:     si x ≠ x.parent alors
22:         x.parent ← Représentant(x.parent)
23:     fin si
24:     retourner x.parent
25: fin fonction

26: procedure Afficher(x : élément)
27:     Afficher x
28:     pour y ∈ x.enfants faire
29:         Afficher(y)
30:     fin pour
31: fin procedure
```

La complexité spatiale n'est pas modifiée puisque l'espace additionnel est $\mathcal{O}(n)$, où n est le nombre d'éléments. En effet, toutes les listes sont initialement vides et on n'ajoute un élément que s'il y a une fusion de deux classes disjointes, ce qui peut survenir au plus $n - 1$ fois.

Clairement, les complexités temporelles amorties sont aussi les mêmes puisque les seules modifications apportées sont des mises à jour d'un attribut en temps constant (on insère un nouvel élément en queue de liste ou en tête de liste, par exemple).

Finalement, la complexité de Afficher(x) est clairement proportionnel à la taille de la classe de x, puisque chaque élément apparaît au plus une fois dans une liste. Pour voir que chaque élément est réellement accessible, il suffit d'observer que l'attribut *enfant* correspond à un arbre "traditionnel", c'est-à-dire non inversé, dans lequel on ne voit pas la compression de chemins, mais où on retient bien tous les éléments qui se trouvent dans cette classe.

Question 4. .... (25 points)

Nous avons vu en classe qu'il existait plusieurs représentations d'un ensemble partiellement ordonné (*poset*). En particulier, une de ces représentations est la représentation matricielle. Supposons donc qu'on ait un poset de n éléments $P = \{x_1, x_2, \ldots, x_n\}$ représenté par une matrice booléenne $n \times n$ et qu'on a $x_i \leq x_j$ si et seulement si $P[i,j] = vrai$, pour $i, j \in \{1, 2, \ldots, n\}$. Autrement dit, on peut utiliser la notation matricielle $P[i,j]$ pour accéder à l'information en temps $\mathcal{O}(1)$.

Proposez une implémentation des opérations suivantes à l'aide de la représentation matricielle. Dans chacun des cas, indiquez la complexité au pire cas à l'aide de la notation $\mathcal{O}$ (la complexité doit être polynomiale) :

(a) (5 points) GEQ(i, j) qui retourne vrai si et seulement si $x_i \geq x_j$.

(b) (10 points) COUVRE(i, j) qui retourne vrai si et seulement si x_i recouvre x_j, c'est-à-dire que $x_i \geq x_j$ et il n'existe aucun $z \in P$ tel que $x_i \geq z \geq x_j$.

(c) (10 points) JOIN(i, j) qui retourne le supremum de x_i et x_j s'il existe, ou qui retourne *rien* sinon.

Solution:

(a) Cette fonction est très facile :

```
1: fonction GEQ(P : poset, i, j : entiers) : booléen
2:     retourner P[j, i]
3: fin fonction
```

Sa complexité est $\mathcal{O}(1)$.

(b) Il suffit de suivre directement la définition :

```
1: fonction COUVRE(P : poset, i, j : entiers) : booléen
2:     si P[i, j] alors
3:         retourner faux
4:     sinon
5:         pour k ← 1, 2, ..., n faire
6:             si k ≠ i, j et P[j, k] et P[k, i] alors
7:                 retourner faux
8:             fin si
9:         fin pour
10:        retourner vrai
11:    fin si
12: fin fonction
```

La complexité dans ce cas est $\mathcal{O}(n)$, puisque toutes les opérations sont constantes et qu'il y a une seule boucle répétée n fois.

(c) On calcule d'abord tous les indices k tels que $x_k \geq x_i, x_j$. Ensuite, on vérifie si un de ces éléments est plus petits que tous les autres (rappelons que le supremum doit être unique).

```
1: fonction JOIN(P : poset, i, j : entiers) : booléen
2:     candidats ← ∅
3:     pour k ← 1, 2, ..., n faire
4:         si P.GEQ(k, i) et P.GEQ(k, j) alors
5:             candidats.AJOUTER(k)
6:         fin si
7:     fin pour
8:     pour c ∈ candidats faire
9:         si P.GEQ(c', c) pour tout c' ∈ candidats alors
10:            retourner c
11:        fin si
12:    fin pour
13:    retourner rien
14: fin fonction
```

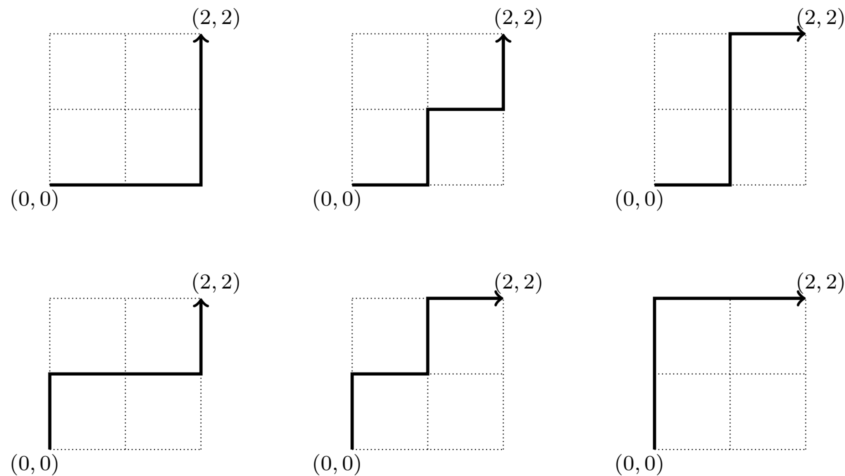
Supposons que la variable *attributs* est implémentée comme un ensemble à l'aide d'une structure d'arbre équilibrée. Alors la complexité de l'algorithme se détaille comme suit :

- La boucle 3–7 s'effectue en $\mathcal{O}(n \log n)$, puisqu'on insère n éléments en $\mathcal{O}(\log n)$.
- La boucle 8–12 s'effectue en $\mathcal{O}(n^2)$, puisqu'on a $\mathcal{O}(n)$ candidats et qu'on vérifie s'il est plus petit que tous les autres en $\mathcal{O}(n)$.

Ainsi, la complexité globale est $\mathcal{O}(n^2)$.

Question 5. .... (25 points)

Soient $m, n \geq 0$ deux entiers. Dans cette question, on s'intéresse à la génération de tous les chemins qu'on peut dessiner sur une grille de dimensions $m \times n$ du point en bas à gauche $(0,0)$ jusqu'au point en haut à droite (m,n) en utilisant seulement des déplacements vers la droite et vers le haut. Par exemple, si $m = n = 2$, alors il existe 6 chemins :



Pour simplifier l'écriture, on représente un chemin par un mot sur l'alphabet $\{d, h\}$, qui dénote les déplacements *droite* et *haut* respectivement.

(a) (10 points) Donnez le pseudocode d'un générateur *récursif*

générateur CHEMINS$(m, n$: naturels) : mots

qui génère tous les mots (sans répétition, et sans en oublier) sur l'alphabet $\{d, h\}$ correspondant à tous les chemins possibles de $(0,0)$ à (m,n) n'utilisant que des déplacements *droite* et *haut*. *Indice :* Tout chemin de $(0,0)$ à (m,n) passe soit par $(m-1, n)$ ou par $(m, n-1)$.

Solution: Dénotons par $\cdot$ la concaténation de deux chemins. Alors on trouve

```
1:  générateur CHEMINS(m, n : naturels) : mots
2:      si n = 0 alors
3:          générer d^m
4:      sinon si m = 0 alors
5:          générer h^n
6:      sinon
7:          pour c ∈ CHEMINS(m − 1, n) faire
8:              générer c · d
9:          fin pour
10:         pour c ∈ CHEMINS(m, n − 1) faire
11:             générer c · h
```

```
12:         fin pour
13:     fin si
14: fin générateur
```

(b) (15 points) Analysez la complexité d'initialisation, de délai et spatiale du générateur proposé en (a).

Solution: Soient $I(m,n)$, $D(m,n)$ et $S(m,n)$ les complexités d'initialisation, de délai et spatiale respectivement.

- Tout d'abord, pour l'initialisation, nous avons

$$I(m,n) = I(m-1,n) + \mathcal{O}(1)$$

puisqu'il faut initialiser récursivement un générateur avec les paramètres $(m-1,n)$ et tout le reste se fait en temps constant. On trouve donc $I(m,n) = \Theta(m)$.

- Pour le délai, on a plutôt

$$D(m,n) = \max(D(m-1,n), D(m,n-1)) + \mathcal{O}(1),$$

puisque la concaténation se fait en temps constant, mais le délai dépend du temps pris pour récupérer récursivement le prochain chemin, selon qu'on est dans la boucle 7–9 ou 10–12. On constate donc que $D(m,n) = m+n$, puisqu'il s'agit essentiellement de reculer du point (m,n) au point $(0,0)$ en faisant $m+n$ pas (peu importe le chemin).

- Finalement, la complexité spatiale satisfait

$$S(m,n) = \max(S(m-1,n), S(m,n-1)) + \mathcal{O}(1),$$

qui a pour solution $\Theta(m+n)$, puisqu'on ne conserve qu'un itérateur récursivement, ainsi que des variables de taille constante. En fait, la seule variable qui n'est pas de taille constante est c, mais elle n'a pas besoin d'être conservée en mémoire dans l'état du générateur, puisqu'aussitôt qu'elle est récupérée récursivement, on peut la jeter.