Université de Caen Normandie — Année 2016-2017
UFR des Sciences — 1ère session

Licence Sciences et Technologie – 2ème année
ELI31T, Algorithmique et structures de données
Mercredi 14 décembre 2016, durée 2h

Les notes et documents de cours, TD et TP sont autorisés.
Le sujet fait **4 pages**.
Les exercices sont *indépendants* et les questions sont largement indépendantes entre elles.
Attention à bien gérer votre temps.

Chaque candidat doit, en début d'épreuve, porter son nom dans le coin de la copie réservé à cet usage ; il le cachettera par collage après la signature de la feuille d'émargement.
Sur chacune des copies intercalaires, il portera son numéro de place et numérotera chaque copie.

Point à respecter : Comme on l'a fait en cours et en TD, vous écrirez les procédures demandées en *langage algorithmique* (pseudo-code).

Exercice 1. Listes simplement chaînées (durée indicative : 35 min)

Dans cet exercice, on considère des *listes d'entiers* supposés positifs ou nuls. Comme on le fait habituellement, on représente une telle liste par une structure *simplement chaînée* dont chaque nœud est une structure à deux champs : `val` un entier et `suiv` un pointeur sur nœud.

Question 1. Rappelez la définition de la structure `noeud` et du type `liste`. Dessinez la structure chaînée qui représente la liste $(3, 5, 3, 7, 1)$ qu'on appellera `L`. N'oubliez pas le pointeur qui donne accès au premier noeud de la liste. Sur cet exemple, précisez les deux valeurs `L->val` et `L->suiv->suiv->suiv->val`.

Question 2. Ecrivez une procédure `Nombre(x: entier, L: liste): entier` qui retourne le nombre de fois où l'entier `x` apparaît dans la liste `L`. Par exemple, `3` apparaît 2 fois dans la liste $(3, 5, 3, 7, 1)$ et `6` apparaît 0 fois dans cette liste.

Question 3. Ecrivez une procédure *récursive* `CopieListe(L: liste): liste` qui retourne une *copie* de la liste `L`, c'est-à-dire, une liste identique à `L` mais qui n'a aucun noeud commun avec `L`.

Question 4. a) Ecrivez une procédure *récursive* `Dernier(L: liste): pointeur sur noeud` qui retourne un pointeur sur le dernier noeud de la liste `L` si `L` n'est pas vide et retourne `None` sinon.

b) Ecrivez une procédure *itérative* `DernierIterative(L: liste): pointeur sur noeud` qui fait la même chose.

Question 5. Que fait la procédure `Mystere` suivante qui utilise les procédures `CopieListe` et `Dernier` construites aux questions précédentes ? Justifiez votre réponse en quelques mots.

```
Mystere(L1,L2: liste): liste
    L1bis: liste ; L1bis = CopieListe(L1)
    p: pointeur sur noeud ; p = Dernier(L1bis)
    si p = None alors retourner CopieListe(L2)
    sinon
        p->suiv = CopieListe(L2) ; retourner L1bis
```

Question 6. On dit qu'une liste `L1` est un *préfixe* d'une liste $\texttt{L2} = (\texttt{a}_1, \texttt{a}_2, \ldots, \texttt{a}_\texttt{n})$ si `L1` forme le début de la liste `L2`, c'est-à-dire, $\texttt{L1} = (\texttt{a}_1, \ldots, \texttt{a}_\texttt{k})$, pour un certain `k` tel que $\texttt{0} \leq \texttt{k} \leq \texttt{n}$. Par exemple, la liste `(1,7,2)` est un préfixe de la liste `(1,7,2,4,8)`. En particulier, la liste vide est préfixe de n'importe quelle liste et toute liste est préfixe d'elle-même.

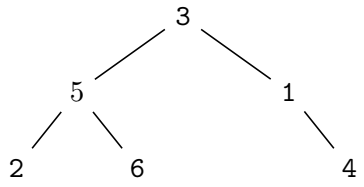
a) Ecrivez une procédure *récursive* `Prefixe(L1,L2: liste): booléen` qui retourne `Vrai` si la liste `L1` est un préfixe de la liste `L2` et `Faux` sinon.
Indication : Considérez d'abord le cas où `L1` est vide, puis le cas où `L2` est vide. Si `L1` et `L2` ne sont pas vides, à quelle condition `L1` est-elle préfixe de `L2` ?

b) Donnez la complexité de la procédure `Prefixe` en fonction de la longueur `n1` de `L1` et/ou de la longueur `n2` de `L2`. Justifiez votre réponse.
Indication : Considérez le nombre d'appels récursifs.

Exercice 2. Arbres binaires (durée indicative : 25 min)

Question 7. a) Rappelez la définition de la structure `noeud` et du type `arbre` utilisés pour représenter un arbre binaire sous la forme chaînée habituelle. On supposera que les valeurs des nœuds sont des entiers.

Dessinez la structure chaînée qui représente l'arbre binaire suivant.



N'oubliez pas le pointeur `A` qui donne accès à la racine de l'arbre.

b) Sur cet exemple, précisez les trois valeurs `A->val`, `A->gauche->droite->val` et `A->droite->gauche`.

c) Sur ce même exemple, donnez la liste des valeurs des nœuds de l'arbre binaire `A`
— en ordre *préfixe*,
— en ordre *postfixe*,
— en ordre *infixe*.

Question 8. Ecrivez une procédure (très simple)
`ConstruitArbre(x: entier, A1, A2: arbre): arbre`
qui construit et retourne l'arbre dont la valeur de la racine est `x`, le *sous-arbre gauche* est `A1` et le *sous-arbre droit* est `A2`.

Question 9. Pour l'arbre binaire `A` représenté ci-dessus, de quels noeuds est constitué son niveau `0` ? son niveau `1` ? son niveau `2` ? son niveau `3` ?

De façon générale, quel est le niveau d'un fils (gauche ou droit) d'un noeud de niveau `n` ?

Question 10. Que fait la procédure `afficheQuoi` suivante ? Justifiez votre réponse.

```
afficheQuoi(A: arbre)
    si A != None alors
        n: entier; F: file de (pointeur sur noeud, entier)
        F = initFile() ; F = enfiler(F,(A, 0))
        Tant que non fileVide(F) faire
            (A, n) = tête(F) ; F = défiler(F) ; afficher(A->val, n)
            si A->gauche != None alors F = enfiler(F,(A->gauche, n+1))
            si A->droit != None alors F = enfiler(F,(A->droit, n+1))
```

Exercice 3. Forêts (durée indicative : 25 min)

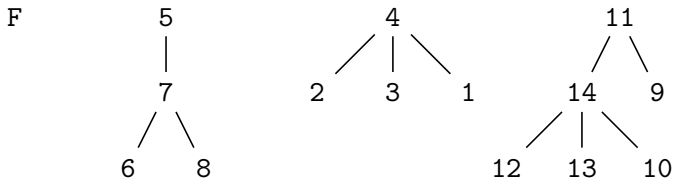
Question 11. Pour la forêt ci-dessous `F` formée de 3 arbres (arbres généraux), donnez la liste des valeurs des noeuds :

a) en ordre *postfixe*,

b) dans l'ordre du *parcours en largeur*.

Rappels : Le parcours postfixe d'une forêt `F` consiste à parcourir en ordre postfixe le 1er arbre de `F`, puis son 2ème arbre, etc.

Le parcours en largeur de `F` consiste à parcourir d'abord les noeuds de niveau 0 de `F` (donc les racines de la forêt), puis ceux de niveau 1, etc.



Question 12. Rappelez la définition de la représentation chaînée *fils-frère* d'un arbre général ou d'une forêt. On supposera que les valeurs des noeuds sont des entiers.

Dessinez la représentation fils-frère de la forêt `F` représentée ci-dessus.

Dans toute la suite de l'exercice, les forêts sont supposées être représentées sous la forme fils-frère.

Question 13. Ecrivez une procédure `AffichePostfixe(F: forêt)` qui affiche les valeurs des noeuds de la forêt `F` en ordre postfixe.

Question 14. On rappelle que dans un arbre ou une forêt une *feuille* est un noeud qui n'a pas de fils. Ecrivez une procédure `NbFeuilles(F: forêt): entier` qui retourne le nombre de feuilles de la forêt `F`.

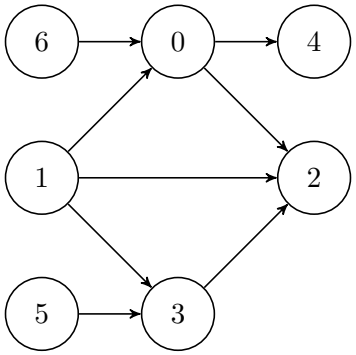
Question 15. Que fait la procédure `Automne` suivante ? Justifiez votre réponse.

```
Automne(F: forêt): forêt
    si F = None alors retourner None
    si F->fils = None alors retourner Automne(F->frère)
    sinon
        F->fils = Automne(F-> fils); F->frère = Automne(F->frère); retourner F
```

Tournez la page SVP pour le dernier exercice

Exercice 4. Graphes orientés (durée indicative : 20 min)

Question 16. Représentez le graphe orienté suivant `G = (S,A)` (où `S = {0,1,2,3,4,5,6}`) par son tableau des listes de successeurs `G.T[ ]` avec donc ici `G.n = 7` :



Question 17. Dans un graphe orienté, on appelle *degré sortant d'un sommet* `x` le nombre de successeurs de `x` (donc le nombre d'arcs sortant de `x`). Par exemple, dans le graphe ci-dessus le degré sortant du sommet `0` est 2 à cause des 2 arcs `(0,4)` et `(0,2)`, seuls arcs sortant de `0`.

Le *degré sortant d'un graphe orienté* est le maximum des degrés sortants de ses sommets. Quel est le degré sortant du graphe ci-dessus ? Justifiez votre réponse.

Question 18. *Cas général :* Ecrivez une procédure `DegreSortant(G: graphe): entier` qui calcule et retourne le degré sortant d'un graphe orienté `G` représenté par son tableau des listes de successeurs.

Donnez la complexité de votre procédure en fonction du nombre `n` de sommets et du nombre `p` d'arcs du graphe.

Le graphe orienté `G = (S,A)` représenté ci-dessus *n'a pas de circuit*. On rappelle qu'un *ordre topologique* d'un graphe `G = (S,A)` orienté sans circuit est un ordre (liste ordonnée) des sommets (encore appelés tâches) de `G` qui respecte les contraintes de précédence : si `(x,y)` est un arc de `G`, alors `x` doit être *avant* `y` dans la liste obtenue.

Question 19. En exécutant l'algorithme de tri topologique (vu en cours), calculez un ordre topologique des sommets du graphe `G` représenté ci-dessus. Ne donnez pas les détails de l'exécution.