LST-A 2005–2006

Documents autorisés

Examen API2

Durée : 2h00

6 janvier 2006

Le sujet comporte 3 exercices qui peuvent être traités indépendamment les uns des autres. L'exercice 2 utilise un type défini dans l'exercice 1 mais ne dépend pas de la résolution des questions de cet exercice.

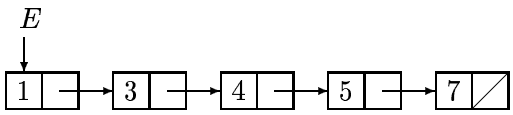
Exercice 1. Ensembles d'entiers

On se propose d'étudier des opérations sur les ensembles d'entiers. On représente les ensembles par des listes *strictement croissantes*. On suppose dans la suite que l'on dispose de l'unité `U_liste` et donc on pourra utiliser toutes les primitives sur les listes. On admet donc les déclarations suivantes :

```
type  ENSEMBLE = LISTE ;
const ENSEMBLEVIDE = LISTEVIDE ;

// estEnsembleVide(E) = true si E est vide
function estEnsembleVide(const E: ENSEMBLE): Boolean ;
begin
  estEnsembleVide := estListeVide(E)
end; // estEnsembleVide
```

Ainsi, l'ensemble $E = \{1,3,4,5,7\}$ sera représenté par :



Question 1.1. *Soit la fonction* PASCAL *suivante :*

```
function appartient(const x: ELEMENT; const E: ENSEMBLE) : Boolean ;
begin
  if estEnsembleVide(E) then appartient := false
  else
    appartient := (x=tete(E)) or ((x>tete(E)) and appartient(x, reste(E))) ;
end; // appartient
```

(1) *Démontrer que si* `x` *est de type* `CARDINAL` *et si* `E` *est de type* `ENSEMBLE` *et a pour valeur une suite strictement croissante d'entiers, alors cette fonction a pour valeur* `true` *ou* `false` *suivant que l'entier* x *appartient ou non à l'ensemble* `E`.

(2) *Est-on certain que cette fonction exploite bien le fait que les entiers sont rangés par ordre strictement croissant, c'est-à-dire qu'il est inutile de poursuivre le parcours de la liste une fois dépassé l'élément* `x` *?*

(3) *Donner le nombre d'appels à la fonction* `tete` *dans le meilleur et dans le pire des cas.*

Question 1.2. *Écrire une fonction qui calcule la réunion d'un ensemble quelconque et d'un singleton.*

Question 1.3. *Écrire une fonction récursive nommée* `reunion1` *qui calcule la réunion de deux ensembles sans utiliser la fonction* `appartient`.

Question 1.4. *Donner une autre version de cette fonction,* `reunion2`, *qui utilise la fonction* `appartient`.

Question 1.5. *Évaluer, pour les fonctions* `reunion1` *et* `reunion2` *le nombre d'appels à la fonction* `tete` *dans le pire des cas.*

Exercice 2. Relations binaires

On considère des relations binaires sur des ensembles d'entiers de la forme $\{1, \ldots, n\}$, où $n > 0$ est un entier donné. Une relation binaire $i \mathcal{R} j$ sur l'ensemble des entiers $\{1, \ldots, n\}$ est représentée par un tableau `R` de longueur n (indicé de 1 à n) dont chaque élément `R[i]` est égal à l'ensemble des entiers j tels que $i \mathcal{R} j$.

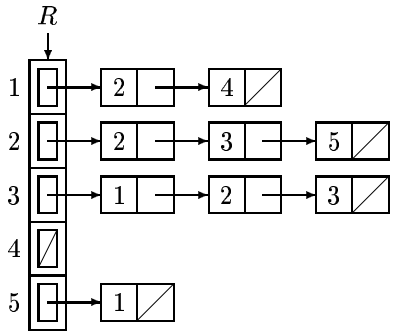
Pour la suite de cet exercice, on se donne les déclarations suivantes :

```
type RELATION = array[1..n] of ENSEMBLE ;
```

où `ENSEMBLE` est le type de l'exercice précédent.

Question 2.1. *Écrire une procédure nommée* `afficherRelation` *qui affiche le diagramme d'une relation binaire.*

Exemple :



Relation

```
    1  2  3  4  5
1|     x     x
2|     x  x     x
3|  x  x  x
4|
5|  x
```

Affichage

Question 2.2. *Écrire une fonction* `inverse` *telle que si* `R` *est un tableau (de type* `RELATION`*) représentant une relation $\mathcal{R}$,* `inverse(R)` *renvoie un tableau représentant la relation $\mathcal{S}$ définie par :*

$$i \mathcal{S} j \iff j \mathcal{R} i$$

Ainsi, l'instruction `afficherRelation(inverse(R))` *donnerait :*

```
    1  2  3  4  5
1|        x     x
2|  x  x  x
3|     x  x
4|  x
5|     x
```

Exercice 3. Maximier et Arbre binaire de recherche

Soit le tableau de nombres $T =$ | 10 | 20 | 4 | 16 | 2 | 40 | 3 | 22 | 24 | 25 | indexé de 1 à 10.

On veut construire le *Maximier quasi-complet* correspondant à ce tableau en utilisant la procédure `construireMaximier` définie en cours.

Question 3.1. *Quel est l'état du tableau à la fin de cette construction ? Dessiner l'arbre qui lui correspond.*

Question 3.2. *Échanger les éléments* `T[1]` *et* `T[10]` *et réorganiser le pré-maximier* `T[1..9]` *en maximier.*

A l'aide du tableau initial (voir début de l'exercice), on veut construire un arbre binaire ordonné par des insertions aux feuilles, en le parcourant de gauche à droite.

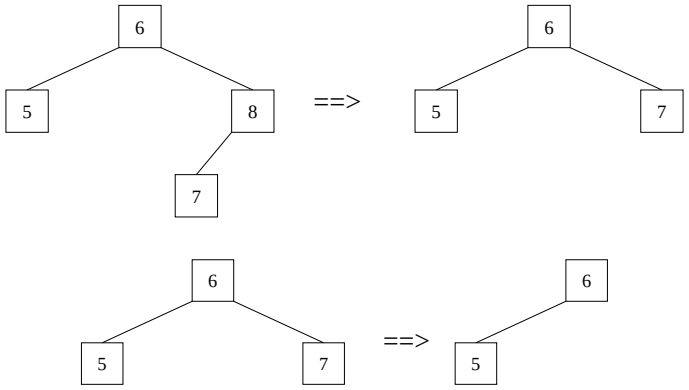
Question 3.3. *Dessiner l'arbre binaire ordonné obtenu.*

Question 3.4. *Retirer l'élément* 20 *et montrer l'état de l'arbre binaire ordonné après le retrait.*

Question 3.5. *On veut réaliser une opération de modification qui consiste à supprimer l'élément maximal d'un ABO non vide.*

Que dire du nombre de fils du nœud maximal d'un ABO ?

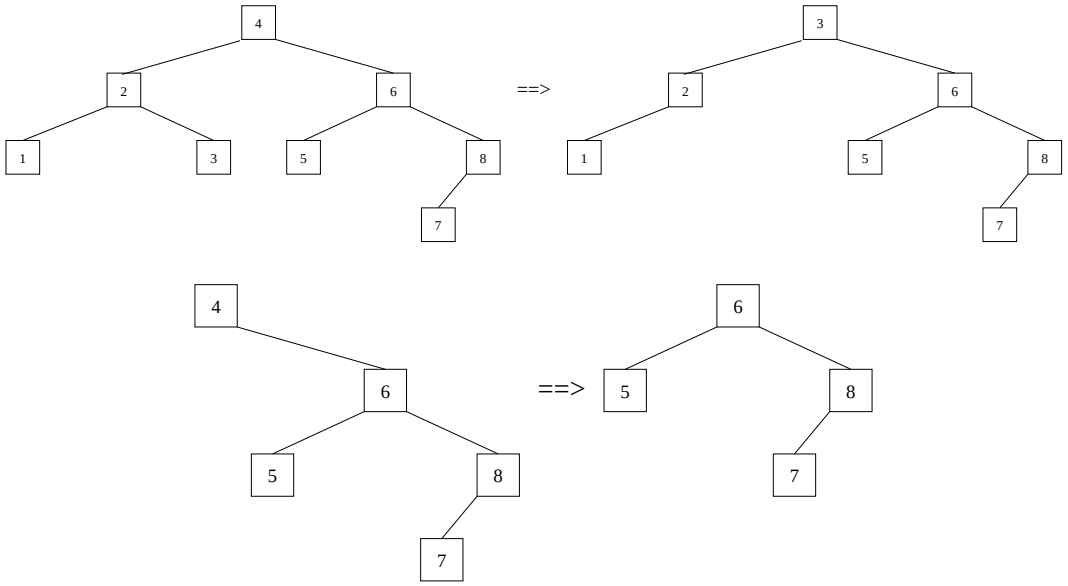
Voici deux exemples d'ABO modifiés par l'action `supprimerMax(a)`



Réalisez la procédure `supprimerMax`.

Question 3.6. *On veut réaliser une opération de suppression de la racine d'un ABO non vide.*

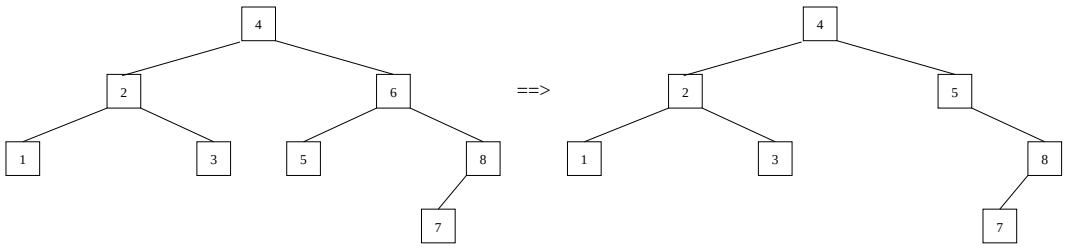
Voici deux exemples d'ABO modifiés par l'action `supprimerRacine(a)`.



Réalisez la procédure `supprimerRacine` *en utilisant la procédure* `supprimerMax`.

Question 3.7. *On veut enfin réaliser une opération de suppression d'un nœud dans un ABO non vide.*

Voici un exemple d'ABO modifié par l'action `supprimerNoeud(a,6)`



Réalisez la procédure `supprimerNoeud`. *Vous pourrez faire l'hypothèse que l'élément* e *à supprimer est présent dans l'ABO et est unique.*