Examen de 1ère session - 2 juin 2010
Documents de cours, TD, TP autorisés
Ouvrages interdits

Exercice 1 : Cours (3 points)

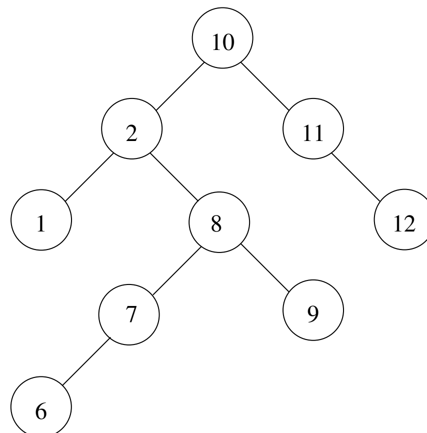
Pour cet exercice, notation QCM : réponse fausse = des points en moins.

Q 1.1 Citer un exemple de tri par comparaisons.

Q 1.2 Citer un exemple de tri qui ne soit pas par comparaisons.

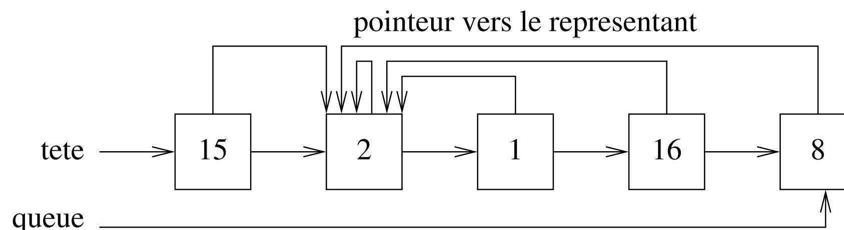
Q 1.3 Qu'est-ce qu'un tri sur place (ou en place) ?

Q 1.4 Donner la(les) opération(s) permettant de rééquilibrer l'arbre suivant obtenu après l'ajout de 6 dans un AVL. Dessiner l'arbre obtenu, préciser sur l'arbre obtenu les valeurs de déséquilibre des nœuds :



Exercice 2 : Structures linéaires (8 points)

On s'intéresse dans cet exercice à la représentation d'ensembles **disjoints**[1] de nombres compris entre 1 et `MAX`. Un ensemble est la donnée de 1 ou plusieurs nombres dont l'un d'eux, n'importe lequel, est le **représentant** de l'ensemble. Pour stocker un ensemble nous allons utiliser une liste chaînée. Par exemple, l'ensemble $\{15, 2, 1, 16, 8\}$, ayant pour représentant 2, sera stocké dans la liste chaînée suivante :



La structure en PASCAL d'un ensemble est la suivante :

[1] Cela veut dire que l'intersection de deux ensembles est vide.

```
type
   ELEMENT = 1..MAX;
   PTRCELLULE = ^CELLULE;

   CELLULE = record
      valeur       : ELEMENT;
      suivant      : PTRCELLULE;
      representant : PTRCELLULE;
   end {record};

   ENSEMBLE = record
      tete  : PTRCELLULE; // vers la premiere cellule
      queue : PTRCELLULE; // vers la derniere cellule
   end {record};
```

De plus les ensembles seront eux-mêmes stockés dans une table de hachage `t`. On précise que la table contiendra des objets de type `ENSEMBLE`.

```
   // une table de hachage qui stocke
   // les ensembles
   TABLE_HACHAGE = ...;
var
   t : TABLE_HACHAGE;
```

On suppose disposer des primitives d'insertion, de suppression et de recherche usuelles sur les tables de hachage.

Q 2.1 Indiquer ce qui vous semble le plus opportun dans le contexte de cet exercice : utiliser une table de hachage à résolution des collisions par chaînage ou bien une table de hachage en adressage ouvert ? Justifier.

Q 2.2 Donner le code de la fonction de hachage, associant un entier à un ensemble :

`function h (e : ENSEMBLE) : CARDINAL;`

Q 2.3 Donner toutes les déclarations de type permettant de définir la structure de la table de hachage (on supposera qu'elle contient au maximum `MAX` clés).

Q 2.4 Quel est alors le coût asymptotique en nombre de comparaisons de l'insertion, de la recherche et de la suppression d'un ensemble dans le pire des cas ?

Nous nous intéresserons aux opérations suivantes sur les ensembles :

- `procedure creer_ensemble (x : CARDINAL);` crée un nouvel ensemble dont le seul élément (et donc le représentant) est `x` et l'insère dans la table de hachage (on supposera que `x` n'est présent dans aucun des ensembles de la table),
- `procedure union(x , y : CARDINAL);` réunit les ensembles dont les représentants sont `x` et `y` en un seul nouvel ensemble, dont le représentant est `y`, et met à jour la table de hachage (on supposera que les deux ensembles représentés par `x` et `y` sont effectivement présents dans la table et comme les ensembles sont disjoints, on n'aura pas à se préoccuper de la détection d'éventuels doublons). Par exemple si la table contient $\{3, 7, 9\}$ représenté par 3 et $\{1, 10, 2\}$ représenté par 2 alors après l'exécution de `union(3,2)` la table ne contiendra plus que l'ensemble $\{3, 7, 9, 1, 10, 2\}$ représenté par 2.

Dans la suite, on supposera que l'ensemble dont le représentant est `x` contient n éléments et que l'ensemble dont le représentant est noté `y` contient m éléments.

Q 2.5 Ecrire le code de la procédure `creer_ensemble`.

Q 2.6 Quelle est la complexité asymtotique en nombre de comparaisons de la fonction `creer_ensemble` dans le pire des cas ? Justifier.

Q 2.7 Quel est l'intérêt dans la représentation des ensembles d'avoir un pointeur sur le dernier élément (champ `queue` de la structure `ENSEMBLE`) ?

Q 2.8 Ecrire le code de la procédure `union`.

Q 2.9 Quelle est la complexité asymptotique en nombre de comparaisons de la fonction `union` dans le pire des cas ?

Un utilisateur de l'unité procède à la suite d'opérations suivante :

```
1 creer_ensemble(1)
2 creer_ensemble(2)
3     ...
4 creer_ensemble(n)
5 union(1,2)
6 union(2,3)
7 union(3,4)
8     ...
9 union(n-1,n)
```

Q 2.10 Donner le nombre exact d'opérations de mise à jour/création de `CELLULE` pour chacune des $2n-1$ opérations données ci-dessus.

Q 2.11 En déduire le nombre moyens d'opérations de mise à jour/création de `CELLULE` de la suite d'opérations donnée ci-dessus.

Q 2.12 Ecrire une suite d'opérations similaire qui aboutisse au même ensemble (pas forcément le même représentant) mais qui se déroule en temps linéaire.

Exercice 3 : Structures arborescentes (6 points)

Deux arbres binaires de recherche (ABR) de formes différentes peuvent contenir les mêmes valeurs.

On souhaite mettre au point un algorithme efficace qui détermine si deux ABR, passés en paramètres, sont équivalents (si ils contiennent les mêmes valeurs avec le même nombre d'occurrences de chacune).

Dans les questions 3.1 et 3.2, l'idée consiste à transformer les ABR en listes puis à manipuler les listes ainsi obtenues.

Q 3.1 Combien de comparaisons sont nécessaires au minimum, dans le pire des cas ? Quelle est la complexité en espace ? Justifier clairement.

Q 3.2 Implanter l'algorithme en PASCAL.

Voici un algorithme :

```
1  procedure mystere (t : ABR);
2  var
3    x : ABR;
4    p : PILE; //pile d ABR initialement vide
5  begin
6    x := t;
7    repeat
8      while x <> ARBRE_VIDE do begin
9        empiler(p,x);
10       x := fils_gauche(x);
11     end {while};
12     if not pilevide(p) then begin
13       x := depiler(p);
14       write(valeur(x));
```

```
15        x := droite(x);
16      end {if};
17    until pilevide(p);
18    writeln;
19  end {mystere};
```

Q 3.3 Expliquer ce que fait cette procédure.

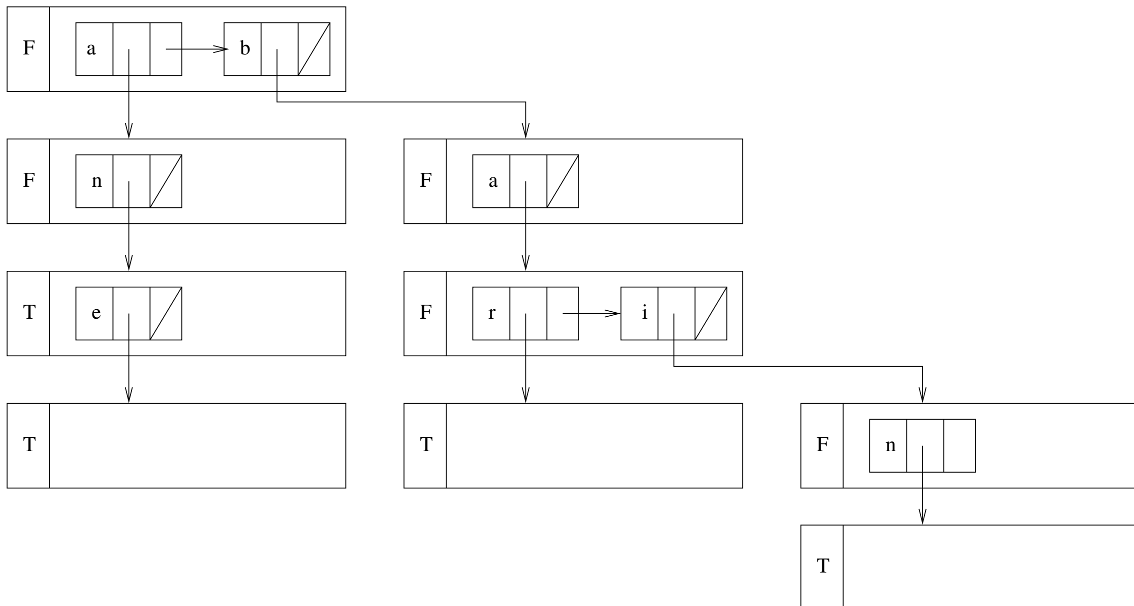
Q 3.4 Donnez sa complexité asymtotique en nombre d'opérations `empiler` et `depiler`. Quelle est la taille maximale atteinte par la pile ?

Q 3.5 En déduire une implémentation de `equivalents` qui utilise un espace mémoire moins important que celui de la question 3.2 (au moins dans le meilleur des cas) mais qui conserve une complexité en temps optimale.

<u>Exercice 4</u> : Dictionnaire (3 points)

En TP vous avez développé une structure de dictionnaire servant à mettre en place un correcteur orthographique. Le comportement de ce dernier consistait à afficher le mot le plus proche du mot mal orthographié.

Nous souhaitons modifier la structure du dictionnaire. Au lieu que chaque nœud du dictionnaire contienne un tableau de pointeurs vers les fils, chaque nœud contiendra une liste de couples pointeur,lettre. Une illustration est donnée ci-dessus où les mots du dictionnaire sont `an, ane, bar, bain`. Notons que les fils ne sont pas nécessairement dans l'ordre de l'alphabet, cela dépend de l'ordre d'insertion des mots dans le dictionnaire.



Le type `NOEUD` du dictionnaire sera donc déclaré ainsi :

```
1  NOEUD = record
2    estUnMot : BOOLEAN;
3    lesFils  : LISTE_DE_COUPLE;
4  end {record};
5  DICO = ^NOEUD;
```

Q 4.1 Déclarer le type `LISTE_DE_COUPLE`.

Q 4.2 On suppose comme dans le projet qu'il existe un unique dictionnaire appelé `dico` qui aura été correctement initialisé avant l'ajout du premier mot. Ecrire la procédure d'ajout d'un mot au dictionnaire :

```
1  procedure ajouter (const mot : MOT);
```