SESSION 2000

MPI008

CONCOURS COMMUNS POLYTECHNIQUES

ÉPREUVE SPÉCIFIQUE-FILIÈRE MP

INFORMATIQUE

DURÉE : 2 heures

Tournez la page S.V.P.

L'usage de toute machine est interdit.

PRÉAMBULE: Les trois parties qui composent ce sujet sont indépendantes et peuvent être traitées par les candidats dans un ordre quelconque.

Partie I : Logique et calcul des propositions

La civilisation Atlante est célèbre pour son niveau scientifique et son goût pour les énigmes logiques. Chaque laboratoire Atlante protégeait ses découvertes par une énigme. Pour accéder aux découvertes d'un laboratoire, il fallait répondre correctement à l'énigme. Si la solution proposée était incorrecte le contenu du laboratoire était détruit. L'interprétation des énigmes (c'est-à-dire les valeurs de vérité associées à chaque proposition composant une énigme) était gérée par des règles propres à chaque laboratoire. Ces règles étaient inscrites sur le seuil de chaque laboratoire.

Un groupe d'archéologues vient de mettre à jour la porte d'un laboratoire Atlante dont le contenu est intact. Sur son seuil figure l'inscription suivante :

Les propositions composant une énigme sont alternativement vraies et fausses, c'est-à-dire que :

- *soit les propositions de numéro pair sont vraies et les propositions de numéro impair fausses,*
- *soit les propositions de numéro pair sont fausses et les propositions de numéro impair vraies.*

Les archéologues arrivent au sas conduisant aux découvertes du laboratoire. Sur la porte se trouvent deux leviers en position haute étiquetés A et B ainsi que les trois propositions suivantes :

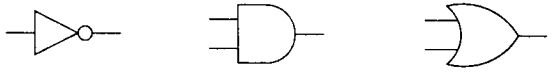
P1 Il faut baisser le levier A,
P2 Il faut baisser simultanément les leviers A et B,
P3 Il ne faut pas baisser le levier B.

Question I.1 *Exprimer P_1, P_2 et P_3 sous la forme de formules du calcul des propositions dépendant de A et de B.*

Question I.2 *Exprimer la règle figurant sur le seuil de la porte de laboratoire dans le contexte des propositions P_1, P_2 et P_3.*

Question I.3 *En utilisant le calcul des propositions (résolution avec les formules de De Morgan ou table de vérité), déterminer l'action à effectuer pour ouvrir la porte du sas sans détruire le contenu du laboratoire.*

Les portes logiques sont représentées graphiquement par les symboles :



Question I.4 *Les leviers A et B sont des interrupteurs qui laissent passer un courant (valeur logique vraie) quand ils sont abaissés. Proposer un circuit électronique composé de portes logiques comportant deux sorties O et D. La sortie O laisse passer le courant pour ouvrir le sas sans détruire le contenu. La sortie D laisse passer le courant pour détruire le contenu du laboratoire.*

Partie II : Algorithmique et programmation en CaML

Cette partie doit être traitée par les étudiants qui ont utilisé le langage CaML dans le cadre des enseignements d'informatique. Les fonctions écrites devront être récursives ou faire appel à des fonctions auxiliaires récursives. Elles ne devront pas utiliser d'instructions itératives (`for`, `while`, ...) ni de références.

Le but de cette partie est l'étude de la structure de « tas » (arbre binaire parfait partiellement trié) et de son application au tri de listes d'entiers. L'intérêt de cette structure est double : elle permet non seulement de réduire le nombre d'opérations de comparaison nécessaires au tri d'une liste mais elle peut également être implantée de manière simple et efficace en utilisant une structure linéaire de type liste ou tableau.

L'algorithme du tri par tas exploite deux opérations élémentaires sur les tas : l'insertion d'un nouveau nœud et l'extraction de la racine. Nous n'étudierons que l'opération d'insertion.

Cette étude sera réalisée en trois étapes : l'étude de l'opération d'insertion dans les structures d'arbre parfait puis de tas, et l'utilisation de la structure de tas dans un algorithme de tri.

1 Représentation des arbres binaires

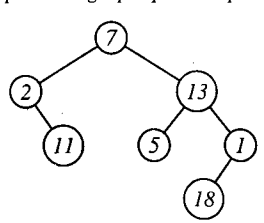
Un arbre binaire composé de nœuds étiquetés par des entiers est représenté par le type CaML :

```
type arbre = Vide | Noeud of int * arbre * arbre;;
```

Exemple II.1 *Le terme suivant*

```
Noeud( 7,
   Noeud( 2,
      Vide,
      Noeud( 11, Vide, Vide)),
   Noeud( 13,
      Noeud( 5, Vide, Vide),
      Noeud( 1,
         Noeud( 18, Vide, Vide),
         Vide)))
```

est alors associé à l'arbre binaire représenté graphiquement par :



Question II.1 *Écrire en CaML une fonction* `taille` *de type* `arbre -> int` *telle que l'appel* `(taille a)` *renvoie le nombre de nœuds contenus dans l'arbre* `a`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

La profondeur d'un nœud est définie comme le nombre de liaisons entre la racine de l'arbre et ce nœud. La profondeur d'un arbre est égale au maximum des profondeurs de ses nœuds. Un niveau dans l'arbre est composé de tous les nœuds ayant la même profondeur.

Dans l'exemple précédent, l'arbre est de profondeur 3 et les différents niveaux contiennent les nœuds suivants :

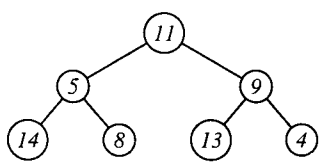
Profondeur	Contenu
0	7
1	2,13
2	1,5,11
3	18

2 Étude des arbres binaires parfaits

2.1 Arbres binaires complets

Un arbre binaire complet est un arbre binaire dont tous les niveaux sont complets, c'est-à-dire que tous les nœuds d'un même niveau ont deux fils sauf les nœuds du niveau le plus profond qui n'ont aucun fils. Autrement dit, le niveau de profondeur p contient 2^p nœuds.

Exemple II.2

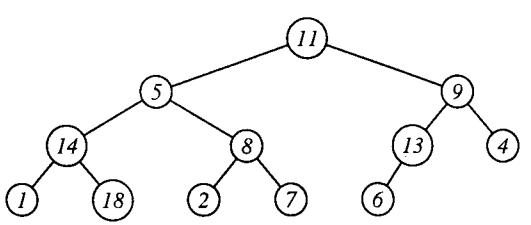


Question II.2 *Calculer le nombre de nœuds d'un arbre binaire complet de profondeur p.*

2.2 Arbres binaires parfaits

Un arbre binaire parfait est un arbre binaire dont tous les niveaux sont complets sauf le niveau le plus profond qui peut être incomplet auquel cas ses nœuds sont alignés à gauche de l'arbre.

Exemple II.3



On remarque qu'il s'agit d'un sous-arbre complet étendu par un dernier niveau partiel contenant les nœuds les plus profonds.

On remarque également que : si l'on complète la partie droite du niveau partiel d'un arbre parfait, on obtient un arbre complet.

Question II.3 *Déterminer un encadrement du nombre n de nœuds dans un arbre parfait en fonction de la profondeur p de cet arbre.*

Question II.4 *En déduire la profondeur d'un arbre parfait contenant n éléments.*

2.3 Numérotation et occurrence des nœuds

L'algorithme naïf d'insertion d'un nœud dans un arbre parfait repose sur un parcours en largeur de l'arbre pour trouver la position du dernier nœud.

Ce parcours coûteux (complexité de l'ordre de $O(\texttt{taille}(arbre))$) peut être évité en utilisant une représentation du chemin menant de la racine à la position où doit être inséré le nouveau nœud.

L'algorithme d'insertion consiste ensuite à parcourir l'arbre en suivant le chemin jusqu'à atteindre la position puis à insérer le nouveau nœud.

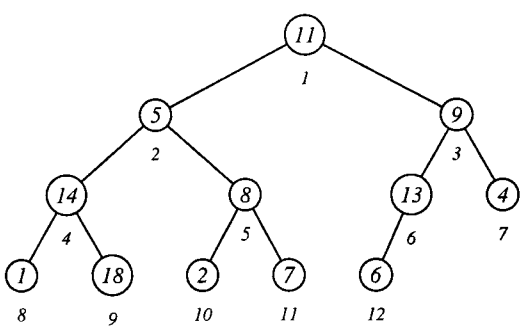
Les questions suivantes permettent de construire cette représentation du chemin.

2.3.1 Numérotation hiérarchique des nœuds

La numérotation hiérarchique des nœuds d'un arbre parfait consiste à les numéroter par un parcours en largeur en partant de la racine (numéro 1) et en parcourant chaque niveau de gauche à droite.

Dans les exemples suivants, le numéro de chaque nœud sera noté en-dessous de son étiquette.

Exemple II.4



Question II.5 *Soit un nœud de numéro n dans le niveau de profondeur p, calculer le nombre de nœuds qui se trouvent à sa gauche dans le niveau de profondeur p.*

Question II.6 *En déduire le nombre de nœuds, dans le niveau de profondeur $p+1$, qui se trouvent à la gauche des fils du nœud de numéro n (n faisant parti du niveau de profondeur p).*

Question II.7 *Soit un nœud de numéro n, calculer les numéros de ses fils gauche et droit.*

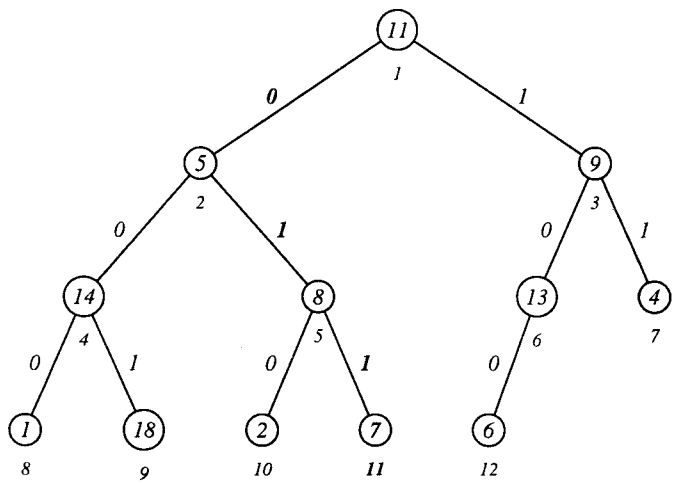
Question II.8 *Déduire de la question précédente, le numéro du père du nœud de numéro n.*

Pour définir plus simplement les opérations de manipulation des nœuds d'un arbre parfait, nous utiliserons systématiquement les numéros des nœuds.

2.3.2 Occurrence d'un arbre

Pour accéder à un nœud, nous devons représenter le chemin dans l'arbre allant de la racine au nœud. Pour cela, nous étiquetons la liaison entre un nœud et son fils gauche par l'entier 0 et la liaison entre un nœud et son fils droit par l'entier 1. Nous appelons alors occurrence, ou chemin, du nœud de numéro n la liste des étiquettes suivies pour aller de la racine à ce nœud. Dans l'exemple suivant, l'occurrence du nœud de numéro 11 étiqueté par la valeur 7 est 0,1,1.

Exemple II.5



Question II.9 *Écrire en CaML une fonction* `occurrence` *de type* `int -> int list` *telle que l'appel* `(occurrence n)` *renvoie une liste d'entiers (parmi 0 et 1) représentant le chemin menant de la racine au nœud de numéro* `n` *avec* $n > 0$. *Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question II.10 *Expliquer la fonction* `occurrence` *définie dans la question précédente.*

Nous utiliserons systématiquement les occurrences des nœuds dans toutes les opérations de manipulation des arbres parfaits. Nous supposons que les occurrences passées en paramètre sont correctes vis-à-vis des arbres passés en paramètre.

2.3.3 Application : Accès à l'étiquette d'un nœud

Question II.11 *Écrire en CaML une fonction* `consulter` *de type* `int list -> arbre -> int` *telle que l'appel* `(consulter c a)` *renvoie la valeur de l'étiquette du nœud accessible en suivant l'occurrence* `c` *dans l'arbre parfait* `a`. *Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

2.4 Insertion d'un nœud

Deux cas se présentent lors de l'insertion d'un nœud dans un arbre parfait :

- soit l'arbre est complet, auquel cas il faut ajouter un niveau supplémentaire contenant uniquement le nouveau nœud positionné à l'extrême gauche de l'arbre,
- soit le dernier niveau de l'arbre est incomplet, auquel cas le nouveau nœud est inséré à l'extrême droite de ce dernier niveau.

Le chemin conduisant au nouveau nœud est donné en paramètre de la fonction d'insertion pour éviter un parcours en largeur de l'arbre nécessaire au calcul de la position du nouveau nœud.

Pour insérer le nouveau nœud, il faut suivre ce chemin en partant de la racine de l'arbre pour arriver à sa position, puis ajouter ce nœud comme fils du nœud précédent dans le chemin.

Question II.12 *Écrire en CaML une fonction* `inserer` *de type* `int -> int list -> arbre -> arbre` *telle que l'appel* `(inserer v c a)` *retourne un arbre parfait obtenu en insérant dans l'arbre* `a` *le nœud étiqueté par* `v` *et d'occurrence* `c` *(cette occurrence correspond à la position du nœud situé après le dernier nœud dans l'arbre parfait* `a`*). Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question II.13 *Expliquer la fonction* `inserer` *que vous avez proposée pour la question précédente.*

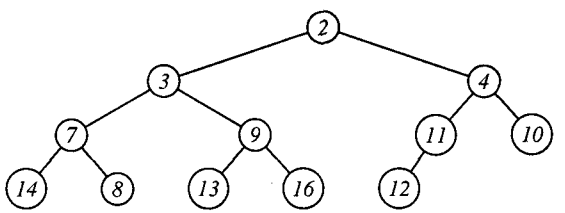
Question II.14 *Calculer une estimation de la complexité de la fonction* `inserer` *en fonction de la taille de l'arbre* `a`*. Cette estimation ne prendra en compte que le nombre d'appels récursifs effectués.*

3 Étude des tas

Lorsqu'il existe une relation d'ordre total entre les étiquettes des nœuds d'un arbre, un arbre est dit partiellement ordonné si les étiquettes contenues dans les nœuds d'un sous-arbre ont toutes une valeur supérieure ou égale à l'étiquette de la racine de celui-ci.

On appelle tas un arbre binaire parfait partiellement ordonné.

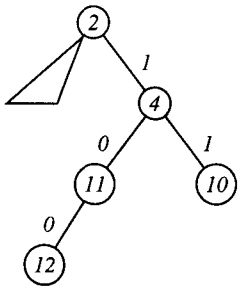
Exemple II.6



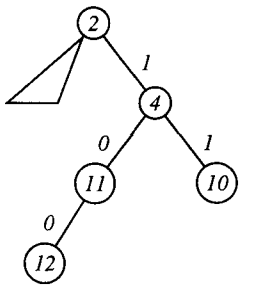
3.1 Insertion d'un nœud dans un tas

Lors de l'insertion d'un nœud, il est nécessaire de préserver l'ordre partiel dans le tas. Pour cela, il faut échanger les étiquettes de certains nœuds dans la branche menant de la racine au nouveau nœud. Ces échanges seront effectués lors du parcours de l'occurrence menant de la racine à la position du nouveau nœud. L'exemple suivant illustre ces échanges lors de l'insertion du nœud étiqueté par la valeur 5 en suivant l'occurrence 1,0,1.

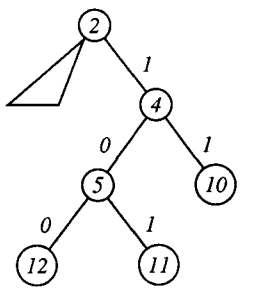
Exemple II.7



Étape n° 1 : liaison entre 2 et 4 : pas d'échange car $5 \geq 2$.



Étape n° 2 : liaison entre 4 et 11 : pas d'échange car $5 \geq 4$.



Étape n° 3 : liaison entre 5 et le nouveau nœud : échange car $5 < 11$.

3.1.1 Programmation de l'insertion

Question II.15 *En modifiant la fonction* `inserer` *de la question II.12, écrire en CaML une nouvelle fonction* `inserer_tas` *de type* `int -> int list -> arbre -> arbre` *telle que l'appel* `(inserer_tas v c a)` *retourne un tas obtenu en insérant dans le tas* `a` *le nœud étiqueté par* `v` *et d'occurrence* `c`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question II.16 *Expliquer la fonction* `inserer_tas` *que vous avez proposée pour la question précédente. En particulier, justifier que l'arbre parfait renvoyé est un tas.*

Question II.17 *Calculer une estimation de la complexité de la fonction* `inserer_tas` *en fonction de la taille de l'arbre* `a`*. Cette estimation ne prendra en compte que le nombre de comparaisons effectuées.*

4 Application au tri par tas

La racine d'un tas contient toujours le minimum des étiquettes des nœuds composant le tas.
Le tri par tas d'une liste d'entiers est donc composé de deux étapes :

- La construction d'un tas contenant les étiquettes de la liste non triée
- L'extraction successive des racines du tas pour construire la liste triée.

Question II.18 *Écrire en CaML une fonction* `construire` *de type* `int list -> arbre` *telle que l'appel* `(construire l)` *renvoie un tas contenant les mêmes éléments que la liste* `l`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question II.19 *Expliquer la fonction* `construire` *que vous avez proposée pour la question précédente. En particulier, justifier que l'arbre engendré par* `construire` *est un tas.*

Question II.20 *Calculer une estimation de la complexité de la fonction* `construire` *en fonction de la taille de la liste* `l`*. Cette estimation ne prendra en compte que le nombre de comparaisons effectuées.*

On dispose de plus d'une fonction CaML `extraire` de type `int list -> arbre -> arbre` telle que l'appel `(extraire c a)` renvoie le tas `a` dans lequel :

- le nœud d'occurence `c` a été enlevé,
- son étiquette remplace celle de la racine,
- les étiquettes ont été permutées pour préserver la structure de tas.

Cette fonction extrait donc le nœud dont l'étiquette est la plus petite en préservant la structure du tas. Le calcul de la fonction `extraire` se termine quelles que soient les valeurs de ses paramètres. On ne demande pas d'écrire cette fonction, dont on admettra que la complexité est un $O(\log_2(\texttt{taille}\ a))$.

Question II.21 *Écrire en CaML une fonction* `aplatir` *de type* `arbre -> int list` *telle que l'appel* `(aplatir a)` *renvoie la liste triée contenant les mêmes éléments que le tas* `a`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question II.22 *Calculer une estimation de la complexité de la fonction* `aplatir` *en fonction de la taille de l'arbre* `a`*. Cette estimation ne prendra en compte que le nombre de comparaisons effectuées.*

Question II.23 *Écrire en CaML une fonction* `trier` *de type* `int list -> int list` *telle que l'appel* `(trier l)` *renvoie la liste triée contenant les mêmes éléments que la liste* `l`*.*

Question II.24 *Calculer une estimation de la complexité de la fonction* `trier` *en fonction de la taille de la liste* `l`*. Cette estimation ne prendra en compte que le nombre de comparaisons effectuées.*

Partie II : Algorithmique et programmation en PASCAL

Cette partie doit être traitée par les étudiants qui ont utilisé le langage PASCAL dans le cadre des enseignements d'informatique. Les fonctions écrites devront être récursives ou faire appel à des fonctions auxiliaires récursives. Elles ne devront pas utiliser d'instructions itératives (`for`, `while`, `repeat`, ...).

Le but de cette partie est l'étude de la structure de « tas » (arbre binaire parfait partiellement trié) et de son application au tri de listes d'entiers. L'intérêt de cette structure est double : elle permet non seulement de réduire le nombre d'opérations de comparaison nécessaires au tri d'une liste mais elle peut également être implantée de manière simple et efficace en utilisant une structure linéaire de type liste ou tableau.

L'algorithme du tri par tas exploite deux opérations élémentaires sur les tas : l'insertion d'un nouveau nœud et l'extraction de la racine. Nous n'étudierons que l'opération d'insertion.

Cette étude sera réalisée en trois étapes : l'étude de l'opération d'insertion dans les structures d'arbre parfait puis de tas, et l'utilisation de la structure de tas dans un algorithme de tri.

1 Représentation des arbres binaires

Un arbre binaire composé de nœuds étiquetés par des entiers est représenté par le type de base `ARBRE`.

Une liste d'entiers est représentée par le type de base `LISTE`.

Nous supposons prédéfinies les constantes et les fonctions suivantes dont le calcul se termine quelles que soient les valeurs de leurs paramètres. Elles pourront éventuellement être utilisées dans les réponses aux questions :

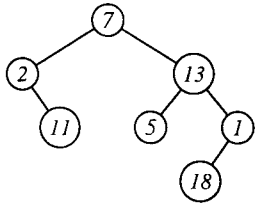
- `Vide` est une constante de valeur `NIL` qui représente un arbre ou une liste vide,
- `FUNCTION Noeud(v:INTEGER; g:ARBRE; d:ARBRE):ARBRE;` est une fonction qui renvoie un arbre dont l'étiquette, le fils gauche et le fils droit de la racine sont respectivement `v`, `g` et `d`,
- `FUNCTION Etiquette(a:ARBRE):INTEGER;` est une fonction qui renvoie l'étiquette de la racine de l'arbre `a`,
- `FUNCTION Gauche(a:ARBRE):ARBRE;` est une fonction qui renvoie le fils gauche de la racine de l'arbre `a`,
- `FUNCTION Droit(a:ARBRE):ARBRE;` est une fonction qui renvoie le fils droit de la racine de l'arbre `a`,
- `FUNCTION Ajouter(i:INTEGER; l:LISTE):LISTE;` est une fonction qui renvoie une liste dont le premier élément est `i` et la liste des autres éléments `l`,
- `FUNCTION Juxtaposer(d:LISTE; f:LISTE):LISTE;` est une fonction qui renvoie une liste résultant de la juxtaposition des éléments des deux listes passées en paramètres, c'est-à-dire dont les premiers éléments sont ceux de la liste `d` et les derniers éléments sont ceux de la liste `f`,
- `FUNCTION Debut(l:LISTE):INTEGER;` est une fonction qui renvoie le premier élément d'une liste `l`,

- `FUNCTION Reste(l:LISTE):LISTE;` est une fonction qui renvoie la liste des éléments suivant le premier dans la liste `l`.

Exemple II.1 *L'appel suivant*

```
Noeud( 7,
   Noeud( 2,
      Vide,
      Noeud( 11, Vide, Vide)),
   Noeud( 13,
      Noeud( 5, Vide, Vide),
      Noeud( 1,
         Noeud( 18, Vide, Vide),
         Vide)))
```

renvoie l'arbre binaire représenté graphiquement par :



Question II.1 *Écrire en PASCAL une fonction `taille(a:ARBRE):INTEGER;` telle que l'appel `(taille a)` renvoie le nombre de nœuds contenus dans l'arbre `a`. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

La profondeur d'un nœud est définie comme le nombre de liaisons entre la racine de l'arbre et ce nœud. La profondeur d'un arbre est égale au maximum des profondeurs de ses nœuds. Un niveau dans l'arbre est composé de tous les nœuds ayant la même profondeur.

Dans l'exemple précédent, l'arbre est de profondeur 3 et les différents niveaux contiennent les nœuds suivants :

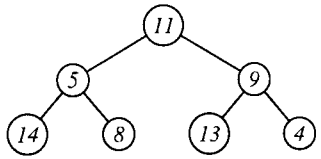
Profondeur	Contenu
0	7
1	2,13
2	1,5,11
3	18

2 Étude des arbres binaires parfaits

2.1 Arbres binaires complets

Un arbre binaire complet est un arbre binaire dont tous les niveaux sont complets, c'est-à-dire que tous les nœuds d'un même niveau ont deux fils sauf les nœuds du niveau le plus profond qui n'ont aucun fils. Autrement dit, le niveau de profondeur p contient 2^p nœuds.

Exemple II.2

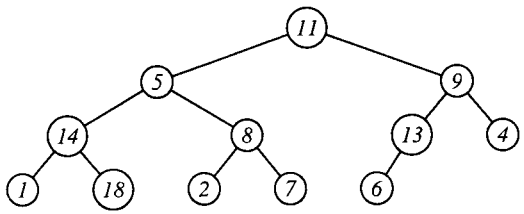


Question II.2 *Calculer le nombre de nœuds d'un arbre binaire complet de profondeur p.*

2.2 Arbres binaires parfaits

Un arbre binaire parfait est un arbre binaire dont tous les niveaux sont complets sauf le niveau le plus profond qui peut être incomplet auquel cas ses nœuds sont alignés à gauche de l'arbre.

Exemple II.3



On remarque qu'il s'agit d'un sous-arbre complet étendu par un dernier niveau partiel contenant les nœuds les plus profonds.

On remarque également que : si l'on complète la partie droite du niveau partiel d'un arbre parfait, on obtient un arbre complet.

Question II.3 *Déterminer un encadrement du nombre n de nœuds dans un arbre parfait en fonction de la profondeur p de cet arbre.*

Question II.4 *En déduire la profondeur d'un arbre parfait contenant n éléments.*

2.3 Numérotation et occurrence des nœuds

L'algorithme naïf d'insertion d'un nœud dans un arbre parfait repose sur un parcours en largeur de l'arbre pour trouver la position du dernier nœud.

Ce parcours coûteux (complexité de l'ordre de $O(\texttt{taille}(arbre))$) peut être évité en utilisant une représentation du chemin menant de la racine à la position où doit être inséré le nouveau nœud.

L'algorithme d'insertion consiste ensuite à parcourir l'arbre en suivant le chemin jusqu'à atteindre la position puis à insérer le nouveau nœud.

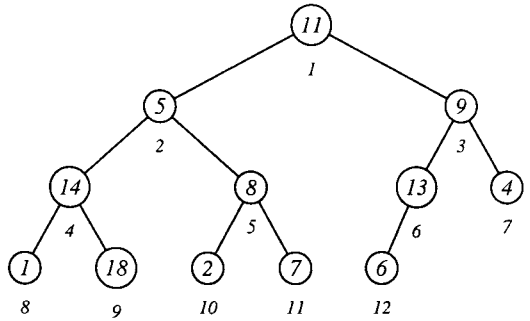
Les questions suivantes permettent de construire cette représentation du chemin.

2.3.1 Numérotation hiérarchique des nœuds

La numérotation hiérarchique des nœuds d'un arbre parfait consiste à les numéroter par un parcours en largeur en partant de la racine (numéro 1) et en parcourant chaque niveau de gauche à droite.

Dans les exemples suivants, le numéro de chaque nœud sera noté en-dessous de son étiquette.

Exemple II.4



Question II.5 *Soit un nœud de numéro n dans le niveau de profondeur p, calculer le nombre de nœuds qui se trouvent à sa gauche dans le niveau de profondeur p.*

Question II.6 *En déduire le nombre de nœuds, dans le niveau de profondeur $p+1$, qui se trouvent à la gauche des fils du nœud de numéro n (n faisant parti du niveau de profondeur p).*

Question II.7 *Soit un nœud de numéro n, calculer les numéros de ses fils gauche et droit.*

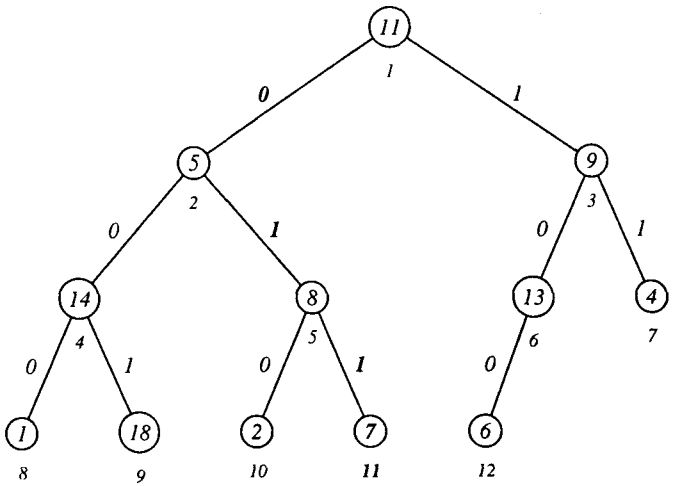
Question II.8 *Déduire de la question précédente, le numéro du père du nœud de numéro n.*

Pour définir plus simplement les opérations de manipulation des nœuds d'un arbre parfait, nous utiliserons systématiquement les numéros des nœuds.

2.3.2 Occurrence d'un arbre

Pour accéder à un nœud, nous devons représenter le chemin dans l'arbre allant de la racine au nœud. Pour cela, nous étiquetons la liaison entre un nœud et son fils gauche par l'entier 0 et la liaison entre un nœud et son fils droit par l'entier 1. Nous appelons alors occurrence, ou chemin, du nœud de numéro n la liste des étiquettes suivies pour aller de la racine à ce nœud. Dans l'exemple suivant, l'occurrence du nœud de numéro 11 étiqueté par la valeur 7 est 0,1,1.

Exemple II.5



Question II.9 *Écrire en PASCAL une fonction* `occurrence(n:INTEGER):LISTE;` *telle que l'appel* `occurrence(n)` *renvoie une liste d'entiers (parmi 0 et 1) représentant le chemin menant de la racine au nœud de numéro* `n` *avec* $n > 0$. *Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question II.10 *Expliquer la fonction* `occurrence` *définie dans la question précédente.*

Nous utiliserons systématiquement les occurrences des nœuds dans toutes les opérations de manipulation des arbres parfaits. Nous supposons que les occurrences passées en paramètre sont correctes vis-à-vis des arbres passés en paramètre.

2.3.3 Application : Accès à l'étiquette d'un nœud

Question II.11 *Écrire en PASCAL une fonction* `consulter(c:LISTE; a:ARBRE):INTEGER;` *telle que l'appel* `consulter( c, a)` *renvoie la valeur de l'étiquette du nœud accessible en suivant l'occurrence* `c` *dans l'arbre parfait* `a`. *Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

2.4 Insertion d'un nœud

Deux cas se présentent lors de l'insertion d'un nœud dans un arbre parfait :

- soit l'arbre est complet, auquel cas il faut ajouter un niveau supplémentaire contenant uniquement le nouveau nœud positionné à l'extrême gauche de l'arbre,
- soit le dernier niveau de l'arbre est incomplet, auquel cas le nouveau nœud est inséré à l'extrême droite de ce dernier niveau.

Le chemin conduisant au nouveau nœud est donné en paramètre de la fonction d'insertion pour éviter un parcours en largeur de l'arbre nécessaire au calcul de la position du nouveau nœud.

Pour insérer le nouveau nœud, il faut suivre ce chemin en partant de la racine de l'arbre pour arriver à sa position, puis ajouter ce nœud comme fils du nœud précédent dans le chemin.

Question II.12 *Écrire en PASCAL une fonction* `inserer(v:INTEGER; c:LISTE; a:ARBRE):ARBRE;` *telle que l'appel* `inserer( v, c, a)` *retourne un arbre parfait obtenu en insérant dans l'arbre* `a` *le nœud étiqueté par* `v` *et d'occurrence* `c`. *Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question II.13 *Expliquer la fonction* `inserer` *que vous avez proposée pour la question précédente.*

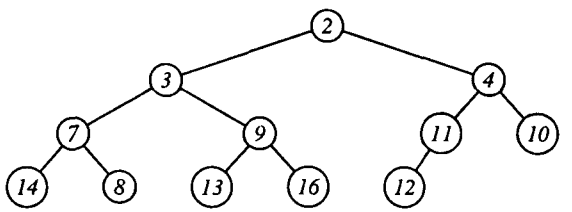
Question II.14 *Calculer une estimation de la complexité de la fonction* `inserer` *en fonction de la taille de l'arbre* `a`. *Cette estimation ne prendra en compte que le nombre d'appels récursifs effectués.*

3 Étude des tas

Lorsqu'il existe une relation d'ordre total entre les étiquettes des nœuds d'un arbre, un arbre est dit partiellement ordonné si les étiquettes contenues dans les nœuds d'un sous-arbre ont toutes une valeur supérieure ou égale à l'étiquette de la racine de celui-ci.

On appelle tas un arbre binaire parfait partiellement ordonné.

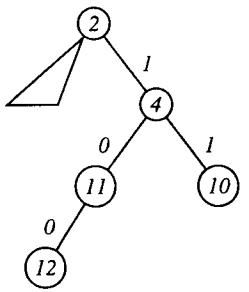
Exemple II.6



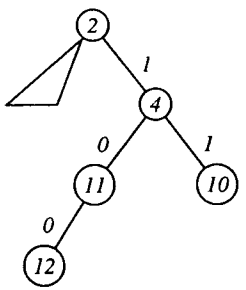
3.1 Insertion d'un nœud dans un tas

Lors de l'insertion d'un nœud, il est nécessaire de préserver l'ordre partiel dans le tas. Pour cela, il faut échanger les étiquettes de certains nœuds dans la branche menant de la racine au nouveau nœud. Ces échanges seront effectués lors du parcours de l'occurrence menant de la racine à la position du nouveau nœud. L'exemple suivant illustre ces échanges lors de l'insertion du nœud étiqueté par la valeur 5 en suivant l'occurrence 1,0,1.

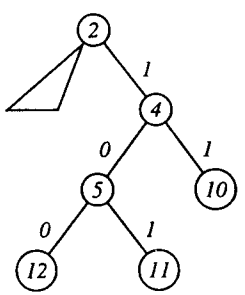
Exemple II.7



Étape n° 1 : liaison entre 2 et 4 : pas d'échange car $5 \geq 2$.



Étape n° 2 : liaison entre 4 et 11 : pas d'échange car $5 \geq 4$.



Étape n° 3 : liaison entre 5 et le nouveau nœud : échange car $5 < 11$.

3.1.1 Programmation de l'insertion

Question II.15 *En modifiant la fonction* `inserer` *de la question II.12, écrire en PASCAL une nouvelle fonction* `inserer_tas(v:INTEGER; c:LISTE; a:ARBRE):ARBRE` *telle que l'appel* `(inserer_tas v c a)` *retourne un tas obtenu en insérant dans le tas* `a` *le nœud étiqueté par* `v` *et d'occurrence* `c`. *Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question II.16 *Expliquer la fonction* `inserer_tas` *que vous avez proposée pour la question précédente. En particulier, justifier que l'arbre parfait renvoyé est un tas.*

Question II.17 *Calculer une estimation de la complexité de la fonction* `inserer_tas` *en fonction de la taille de l'arbre* `a`. *Cette estimation ne prendra en compte que le nombre de comparaisons effectuées.*

4 Application au tri par tas

La racine d'un tas contient toujours le minimum des étiquettes des nœuds composant le tas.
Le tri par tas d'une liste d'entiers est donc composé de deux étapes :

- La construction d'un tas contenant les étiquettes de la liste non triée
- L'extraction successive des racines du tas pour construire la liste triée.

Question II.18 *Écrire en PASCAL une fonction* `construire(l:LISTE):ARBRE;` *telle que l'appel* `construire(l)` *renvoie un tas contenant les mêmes éléments que la liste* `l`. *Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question II.19 *Expliquer la fonction* `construire` *que vous avez proposée pour la question précédente. En particulier, justifier que l'arbre engendré par* `construire` *est un tas.*

Question II.20 *Calculer une estimation de la complexité de la fonction* `construire` *en fonction de la taille de la liste* `l`. *Cette estimation ne prendra en compte que le nombre de comparaisons effectuées.*

On dispose de plus d'une fonction PASCAL `extraire(c:LISTE; a:ARBRE):ARBRE;` telle que l'appel `extraire(c, a)` renvoie le tas `a` dans lequel :

- le nœud d'occurence `c` a été enlevé,
- l'étiquette de la racine a été remplacée par l'étiquette ce nœud,
- les étiquettes ont été permutées pour préserver la structure de tas.

Le calcul de la fonction `extraire` se termine quelles que soient les valeurs de ses paramètres. On ne demande pas d'écrire cette fonction, dont on admettra que la complexité est un $O(\log_2(\texttt{taille}(a)))$.

Question II.21 *Écrire en PASCAL une fonction* `aplatir(a:ARBRE):LISTE;` *telle que l'appel* `aplatir(a)` *renvoie la liste triée contenant les mêmes éléments que le tas* `a`. *Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question II.22 *Calculer une estimation de la complexité de la fonction* `aplatir` *en fonction de la taille de l'arbre* `a`. *Cette estimation ne prendra en compte que le nombre de comparaisons effectuées.*

Question II.23 *Écrire en PASCAL une fonction* `trier(l:LISTE):LISTE;` *telle que l'appel* `trier(l)` *renvoie la liste triée contenant les mêmes éléments que la liste* `l`.

Question II.24 *Calculer une estimation de la complexité de la fonction* `trier` *en fonction de la taille de la liste* `l`. *Cette estimation ne prendra en compte que le nombre de comparaisons effectuées.*

Partie III : Automates et langages

Le but de cet exercice est l'étude des propriétés de l'opération $\ominus$ de composition de deux automates finis déterministes.

Soit l'alphabet X, un ensemble de symboles, soit Λ (parfois noté abusivement ϵ) le symbole représentant le mot vide ($\Lambda \notin X$), $X^\star$ l'ensemble des mots composés de symboles de X ($\Lambda \in X^\star$), un automate fini déterministe sur X est un quintuplet $A = (Q,X,i,T,\delta)$ composé de :

- Un ensemble d'états : Q,
- L'état initial : $i \in Q$,
- Un ensemble d'états terminaux : $T \subseteq Q$,
- Une fonction de transition : $\delta : X \times Q \to Q$, définie a priori sur une partie de $X \times Q$.

Les valeurs de la fonction de transition δ seront représentées par un graphe dont les nœuds sont les états. Un état initial sera entouré d'un cercle (i) et un état final sera entouré d'un double cercle $((t))$.

Un automate déterministe est dit « complet » si la fonction δ est totale (définie sur $X \times Q$ tout entier).

Dans cette partie, les automates finis considérés seront complets.

Soit $\delta^\star$ l'extension de δ à $X^\star \times Q$ définie par :

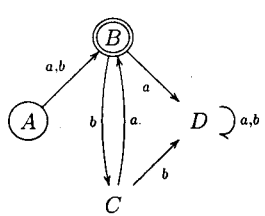
$$\begin{cases} \forall q \in Q, \delta^\star(\Lambda,q) = q \\ \forall x \in X, \forall q \in Q, \delta^\star(x,q) = \delta(x,q) \\ \forall m \in X^\star, \forall x \; in X, \forall q \in Q, \delta^\star(m.x,q) = \delta(x,\delta^\star(m,q)) \end{cases}$$

Le langage sur $X^\star$ reconnu par cet automate fini est :

$$L(A) = \{m \in X^\star \mid t \in T, \delta^\star(m,i) = t\}$$

Étude de l'exemple $\mathcal{E}_1$

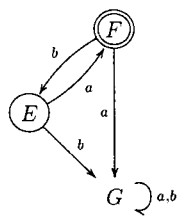
Soit l'automate fini déterministe complet $\mathcal{E}_1 = (\{A,B,C,D\},\{a,b\},A,\{B\},\delta_{\mathcal{E}_1})$ dont la fonction de transition est définie par :



Question III.1 *Donner sans la justifier une expression régulière ou ensembliste représentant le langage reconnu par $\mathcal{E}_1$.*

Étude de l'automate $\mathcal{E}_2$

Soit l'automate fini déterministe complet $\mathcal{E}_2 = (\{E,F,G\},\{a,b\},E,\{F\},\delta_{\mathcal{E}_2})$ dont la fonction de transition est définie par :



Question III.2 *Donner sans la justifier une expression régulière ou ensembliste représentant le langage reconnu par $\mathcal{E}_2$.*

Composition des automates : $\mathcal{E}_1 \ominus \mathcal{E}_2$

Soit l'opération interne $\ominus$ sur les automates finis déterministes définie par :

Déf. III.1 (Composition d'automates) *Soient $A_1 = (Q_1,X,q_1,T_1,\delta_1)$ et $A_2 = (Q_2,X,q_2,T_2,\delta_2)$ deux automates finis déterministes, l'automate $A = A_1 \ominus A_2$ est défini par :*

$$A = (Q_1 \times Q_2, X, (q_1,q_2), T_1 \times (Q_2 \setminus T_2), \delta_{1\ominus 2})$$
$$\delta_{1\ominus 2}(a,(x,y)) = (x',y') \text{ si } \delta_1(a,x) = x' \text{ et } \delta_2(a,y) = y'$$

On remarquera que les états de A sont des paires d'états de A_1 et A_2.

Question III.3 *Construire l'automate $\mathcal{E}_1 \ominus \mathcal{E}_2$ (seuls les états et les transitions utiles, c'est-à-dire accessibles depuis l'état initial, devront être construits).*

Question III.4 *Caractériser le langage reconnu par $\mathcal{E}_1 \ominus \mathcal{E}_2$ par une expression régulière ou ensembliste. Comparer ce langage avec les langages reconnus par $\mathcal{E}_1$ et $\mathcal{E}_2$.*

Étude de l'automate « composé »

Question III.5 *Montrer que : si A_1 et A_2 sont des automates finis déterministes complets, alors $A_1 \ominus A_2$ est un automate fini déterministe complet.*

Question III.6 *Montrer que $\delta^\star_{1\ominus 2}(m,(q_1,q_2)) = (\delta^\star_1(m,q_1),\delta^\star_2(m,q_2))$*

Question III.7 *Montrer que : $m \in L(A_1 \ominus A_2) \Leftrightarrow (m \in L(A_1) \wedge m \notin L(A_2))$.*

Question III.8 *Quelle relation liant les langages reconnus par les automates A_1, A_2 et $A_1 \ominus A_2$ peut-on en déduire ?*

Fin de l'énoncé