CCP 2008. Option Informatique.

Corrigé pour serveur UPS par JL. Lamard (jean-louis.lamard@prepas.org)

Partie I. Logique et calcul des propositions.

1) $P_1 = A$, $P_2 = AB$ et $P_3 = \overline{B}$.

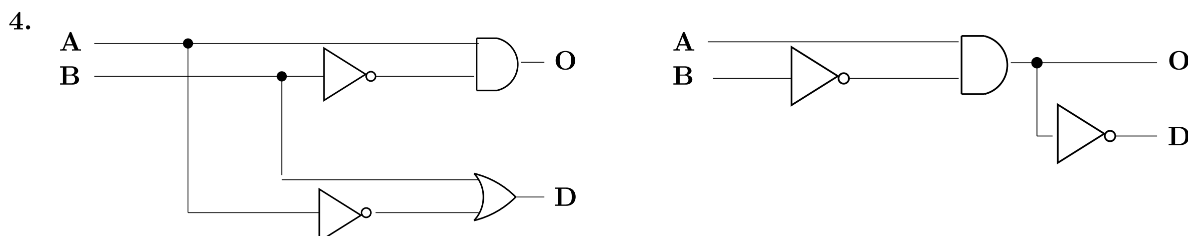
2) $P_1\overline{P_2}P_3 + \overline{P_1}P_2\overline{P_3}$.

3) Or $P_1\overline{P_2}P_3 = A(\overline{AB})\overline{B} = A(\overline{A}+\overline{B})\overline{B} = A\overline{A}\ \overline{B} + A\overline{B}\ \overline{B} = 0 + A\overline{B} = A\overline{B}$ et

$\overline{P_1}P_2\overline{P_3} = \overline{A}ABB = 0$.

Pour ouvrir la porte il faut donc fermer A et laisser B ouvert.

4.



Le premier circuit est de profondeur 2 alors que le second est de profondeur 3.

Remarque : L'énoncé me paraît peu clair. Il est dit que les deux interrupteurs sont ouverts au départ ce qui pourrait signifier que la salle est déjà auto-détruite .... Il faudrait préciser que l'état des entrées A et B ne sera pris en compte qu'après un certain délai ... de réflexion.

Partie II. Automates et langages.

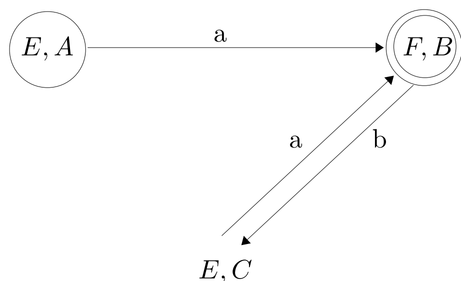
1. Automate fini complet déterministe.

1) Le langage régulier L_1 (théorème de Kleene) reconnu par $\mathcal{E}_1$ peut être décrit par l'expression régulière $L_1 = a(ba+c)^*$

Celui reconnu par $\mathcal{E}_2$ peut être décrit par $L_2 = (a+b+cc^*a)(bc^*a)^*$.

2. Composition d'automates finis complets déterministes.

2) et 3)



Le langage reconnu par $\mathcal{E}_1 \times \mathcal{E}_2$ est $L_1 \times L_2 = a(ba)^* = L_1 \cap L_2$

Notons que l'automate représenté ci-contre n'est pas complet. Il ne représente que les états et transitions utiles de $\mathcal{E}_1 \times \mathcal{E}_2$ qui lui est naturellement complet puisque $\mathcal{E}_1$ et $\mathcal{E}_2$ le sont.

4), 5) et 6)

Évident par définition même de la composition de deux automates finis complets déterministes et de la définition de l'extension δ^* (démonstration par récurrence sur $n = |m|$ pour la question 5).

7) Il découle immédiatement de la question 6) que $L(\mathcal{A}_1 \times \mathcal{A}_2) = L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$

Partie III. Algorithmique et programmation en CaML.

1. Réalisation à base de listes.

1) Le fonction suivante insère v en fin de la liste E si $v \notin E$.

```
let rec insertion_ens v E = match E with
  | [] -> [v]
  | a :: suite -> if (v = a) then E
                  else a :: insertion_ens v suite
;;
insertion_ens : int -> int list -> int list = <fun>
```

2) Soit $T(n)$ la complexité temporelle de cette fonction pour une liste de longueur n. On a au mieux $T(n)$ constant si v est égal au premier élément de E et au pire $T(n) = T(n-1) + a$ donc $T(n) = an$ si v n'appartient à E. □

3)

```
let rec elimination_ens v E = match E with
  | [] -> []
  | a :: suite -> if (v = a) then suite
                  else a :: elimination_ens v suite
;;
elimination_ens : int -> int list -> int list = <fun>
```

4) Comme précédemment on a au mieux $T(n)$ constant si v est le premier élément de E et au pire $T(n) = an$ si v ne figure pas dans E. □

2. Réalisation à base d'arbres binaires de recherche bicolore.

5) Si $a = \varnothing$ alors $|A| = 0$ sinon $|A| = \max\big(|\mathcal{G}(a)|, |\mathcal{D}(a)|\big) + 1$. □

6) Dans toute la suite, pour alléger les notations, on notera p la profondeur et r le rang d'un arbre a bicolore donné.

La profondeur est évidemment maximale et égale à n lorsque chaque nœud interne ne possède qu'un fils (droit ou gauche). La profondeur est alors égale à n. Cela revient exactement à coder par une liste chaînée. □

Un arbre de profondeur p contient au plus $1 + 2 + 2^2 + \ldots + 2^{p-1} = 2^p - 1$ nœuds lorsque toutes les branches sont de longueur p.
Soit alors un arbre ayant n nœuds. Sa profondeur minimale est donc le plus petit entier p tel que $2^p - 1 \geqslant n$ c'est à dire p tel que $2^{p-1} < n + 1 \leqslant 2^p$ i.e. $p - 1 < \log_2(n+1) \leqslant p$. □
La forme d'un tel arbre est telle que tous les étages sauf éventuellement le dernier sont complets. □

7) On a évidemment $r \leqslant p$ puisque r est le nombre de nœuds gris de n'importe quelle branche donc est au plus égal au nombre de nœuds de n'importe quelle branche et en particulier d'une branche maximale.

Par ailleurs il y a au plus $2r + 1$ nœuds au total dans toute branche (et en particulier dans une branche maximale) : un blanc suivi d'un gris suivi d'un blanc ... pour terminer par un gris suivi d'un blanc compte tenu de **P2**.

Ainsi pour un arbre bicolore on a $r \leqslant p \leqslant 2r + 1$. □

Un arbre bicolore de rang r contient le minimum de noeuds lorsque tous ses nœuds sont gris. C'est alors un arbre binaire complet de profondeur r donc qui contient $1 + 2 + \ldots + 2^{r-1} = 2^r - 1$ nœuds.
Donc si n est le nombre de nœuds d'un arbre bicolore on a $2^r - 1 \leqslant n$. □

8) Notons que l'inégalité proposée dans l'énoncé est fausse pour $n = 1$ car alors $p = 1$.
Nous allons prouver qu'en fait $E(\log_2 n) \leqslant p - 1 \leqslant 2E(\log_2 n)$.

- D'après la question 6), on a $n \leqslant 2^p - 1$ donc $\log_2(n) \leqslant \log_2(2^p - 1)$ donc par croissance de la fonction partie entière $E\big(\log_2(n)\big) \leqslant E\big(\log_2(2^p - 1)\big) = p - 1$
- Si a est complet alors $n = 2^p - 1$ donc $E(\log_2 n) = E\big(\log_2(2^p - 1)\big) = p - 1$ donc $2E(\log_2 n) = 2p - 2 \geqslant p - 1$ car $p \geqslant 1$.
  Si a n'est pas complet alors $2^r - 1 < n$ car on a vu précédemment qu'il y a égalité si et seulement si a est complet (et $p = r$). Ainsi $n \geqslant 2^r$ donc $\log_2 n \geqslant r$ d'où $2E(\log_2 n) \geqslant 2r \geqslant p - 1$ d'après la question précédente.
- Ainsi pour un arbre bicolore on a $E(\log_2 n) \leqslant p - 1 \leqslant 2E(\log_2 n)$ donc $p = \Theta(\log_2 n)$. □

9) La fonction suivante renvoie le nombre de nœuds gris de la branche la plus à gauche d'un arbre coloré donc le rang d'un arbre bicolore.

```
let rec rang a = match a with
  | Vide -> 0
  | Noeud (c, fg, v, fd) ->
      let n = rang fg in if (c = Blanc) then n else n+1
;;
rang : arbre -> int = <fun>
```

10) On commence par écrire une fonction auxiliaire `valid_aux` de sorte qu'appliquée à un arbre binaire `a` elle renvoie un couple d'un booléen et d'un entier tel que : si l'arbre est bicolore alors le booléen vaut `vrai` et l'entier est égal au rang de `a` et sinon le booléen vaut `faux` (l'entier étant alors sans signification).

Pour faciliter sa rédaction, écrivons une fonction `col_rac` qui renvoie la chaîne "V" si l'arbre est vide, "B" si la racine est blanche et "G" si elle est grise.

```
let col_rac a = match a with
  | Vide -> "V"
  | Noeud (c, fg, v, fd)  -> if (c = Blanc) then "B" else "G"
;;
col_rac : arbre -> string = <fun>
```

puis :

```
let rec valid_aux a = match a with
  | Vide -> (true, 0)
  (*| Noeud (c, Vide, _, Vide) -> if (c = Gris) then (true, 1) else (true,0) *)
  | Noeud (c, fg, v, fd) -> match c with
      | Gris ->  let (bool_g, nb_g) = valid_aux fg
                 and (bool_d, nb_d) = valid_aux fd in
                   (bool_g & bool_d & (nb_g = nb_d), succ nb_g)
      | Blanc -> if (col_rac fg = "B") || (col_rac fd = "B")
                   then (false,0)
                 else let (bool_g, nb_g) = valid_aux fg
                        and (bool_d, nb_d) = valid_aux fd in
                   (bool_g & bool_d & (nb_d = nb_g), nb_g)
;;
valid_aux : arbre -> bool * int = <fun>
```

et enfin

```
let validation_bicolore a =
  let (bool, n) = valid_aux a in bool
;;
validation_bicolore : arbre -> bool = <fun>
```

11) Notons $T(n)$ la complexité temporelle pour un arbre de longueur n. On a, compte-tenu de l'algorithme `valid_aux`, $T(n) = T(p) + T(q) + \Theta(1)$ avec $p + q = n - 1$.
Par une récurrence immédiate, il en découle que $T(n) = \Theta(n)$. □

12) On commence par écrire (voir page suivante) une fonction `valid_aux_abr` qui appliquée à un arbre coloré NON VIDE renvoie un triplet $(bool, mini, maxi)$ tel que $bool = vrai$ si et seulement si l'arbre est de recherche et avec $mini$ (resp. $maxi$) égal au plus petit (resp. plus grand) entier qui soit étiquette d'un nœud de l'arbre.
L'écriture de la fonction `valisation_abr` est alors immédiate.

```
let validation_abr a =
  if (a = Vide) then true
  else let (bool, mini, maxi) = valid_aux_abr a in bool
;;
validation_abr : arbre -> bool = <fun>
```

On peut remarquer que la complexité de `valid_aux` est encore en $\Theta(n)$ par la même démonstration que ci-dessus.

```
let rec valid_aux_abr a = match a with
  | Vide -> failwith "arbre vide non atteint es partant d'un arbre non vide"
  | Noeud(c, Vide, v, Vide) -> (true, v, v)
  | Noeud(c, Noeud (cg, fgg, vg, fgd), v, Vide) ->
     let (bool, mini, maxi) =  valid_aux_abr (Noeud (cg, fgd, vg, fgd)) in
       (bool & (maxi < v), min v mini, max v maxi)
  | Noeud(c, Vide, v, Noeud (cd, fdg, vd, fdd)) ->
     let (bool, mini, maxi) =  valid_aux_abr  (Noeud (cd, fdg, vd, fdd)) in
       (bool & (mini > v), min v mini, max v maxi)
  | Noeud(c, Noeud (cg, fgg, vg, fgd), v, Noeud(cd, fdg, vd, fdd)) ->
     let (boolg, minig, maxig) =  valid_aux_abr (Noeud (cg, fgg, vg, fgd))
     and (boold, minid, maxid) =  valid_aux_abr (Noeud (cd, fdg, vd, fdd)) in
       (boolg & boold & (maxig < v) & (minid > v),
        min (min minig minid) v,
        max (max maxig maxid) v)
;;
valid_aux_abr : arbre -> bool * int * int = <fun>
```

13) L'écriture récursive est immédiate :

```
let rec insertion_abr v a = match a with
  | Vide -> Noeud (Blanc, Vide, v, Vide)
  | Noeud (c, fg, n, fd) -> match 0 with
      | _ when v = n -> a
      | _ when v < n -> Noeud (c, insertion_abr v fg, n, fd)
      | _            -> Noeud (c, fg, n, insertion_abr v fd)
;;
insertion_abr : int -> arbre -> arbre = <fun>
```

14) Dans le meilleur des cas la complexité est à temps constant lorsque l'entier à insérer est égal à l'étiquette de la racine (ou lorsque l'arbre est vide !).

Dans le pire des cas il y a autant d'appels récursifs que de nœuds dans la branche la plus longue i.e. la profondeur p. Or à chaque appel récursif, outre le temps de l'appel, il y a une seule opération à temps constant (la construction du nouvel arbre par le constructeur de type `Noeud`). Ainsi dans ce cas $T(n) = \alpha p$.

On peut donc retenir que dans tous les cas $T(n) = O(p)$. □

Dans la cas d'un arbre en outre bicolore, on a vu (question 8) que $p = \Theta(\log_2 n)$ donc $T(n) = O(\log_2 n)$ □

Par contre si l'arbre de recherche n'est pas bicolore alors il peut fort bien être constitué d'une seule branche : suite d'entiers strictement décroissante avec uniquement des fils gauches. Alors l'insertion d'un entier plus petit que celui de profondeur minimum aura une complexité linéaire.
On voit donc bien l'intérêt d'avoir un arbre de recherche bicolore.

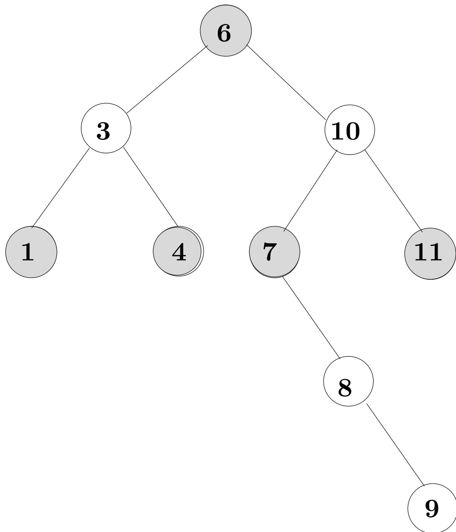
15) La contrainte **P3** est évidemment conservées car on insère nœud blanc ce qui ne modifie pas le nombre de nœuds gris de chaque branche. Par contre **P2** n'est pas forcément conservée : le père du nœud inséré ou un de ses fils peut être blanc. □

16) Notons B_1 l'arbre transformé résultant de la correction d'un arbre B d'un type de l'énoncé.
Remarquons que B vérifie **P3** avec $r = n + 1$ en notant r le nombre de nœuds gris de chaque branche de B.
Comme F_1, F_2, F_3 et F_4 sont bicolores, il est immédiat que la contrainte **P2** est vérifiée par l'arbre B_1.
Comme en outre les arbres F_i ont le même rang n, la propriété **P3** est bien vérifiée : toutes les branches de B_1 contiennent exactement $n + 1$ nœuds gris.
Ainsi B_1 est bien un arbre bicolore de rang $n + 1$. □

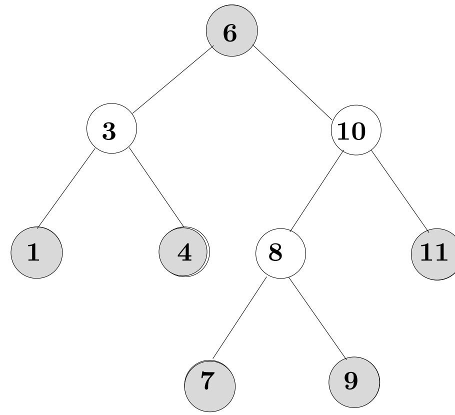
Remarque 1 : si l'arbre initial est en outre de recherche alors on vérifie immédiatement qu'il en va de même de l'arbre tranformé.

Remarque 2 : Soit un arbre A vérifiant **P3** et présentant un sous-arbre B du type de l'énoncé et A_1 l'arbre obtenu en remplaçant B par sa correction B_1. Alors ce qui précède prouve que A_1 vérifie encore **P3** avec la même valeur que A.

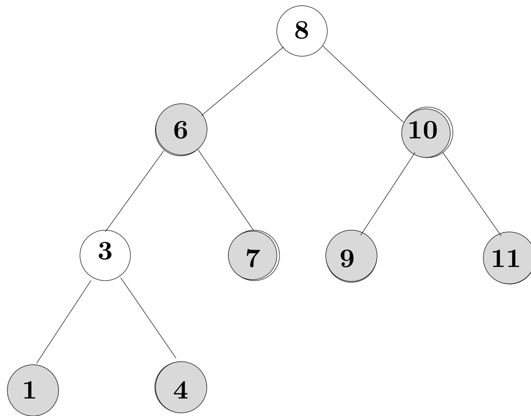
17)



Après l'insertion de 9



Après la première correction.



Après la seconde correction
C'est un arbre bicolore de recherche.

18) et 19)

Soit A un arbre de recherche bicolore, de profondeur p, de rang r et A_0 le résultat de l'insertion d'une nouvelle valeur c.

Remarque 1 L'inégalité stricte $nb_c < p - r$ demandée en question 19) est fausse (nb_c désignant le nombre de corrections nécessaires à l'obtention d'un arbre bicolore après l'insertion).
Prenons un arbre A de profondeur $p = 2$, de racine grise étiquetée 1, de fils gauche vide et de fils droit blanc étiqueté par 2. Il est bien bicolore de recherche avec $r = 1$. Insérons 3. Il se rajoute en fils droit blanc de 2. Une correction est nécessaire pour obtenir un arbre bicolore et $1 = p - r$.
!! On se propose de prouver que d'une manière générale $nb_c \leqslant p - r$.

Remarque 2 Les 4 types de correction proposés ne permettent pas toujours d'obtenir un arbre bicolore après insertion.
Prenons l'arbre A ayant un seul nœud blanc étiqueté par 1. Il est bien bicolore de recherche. Insérons 2 qui se rajoute en fils droit de la racine. A_0 n'est pas bicolore et n'est pas justifiable d'un des 4 types de correction proposés. Pour le rendre bicolore dans ce cas il suffit de griser la racine.
!! Aussi avant d'insérer une nouvelle valeur dans A commencera-t-on par griser la racine si elle est blanche. Cela préserve évidemment la structure bicolore de recherche (cela augmente simplement d'une unité le rang de A).

Soit alors A un tel arbre de recherche bicolore (racine grise), de profondeur p, de rang r et A_0 le résultat de l'insertion d'une nouvelle valeur c.
Il s'agit, on le sait, d'un arbre de recherche.

- S'il est bicolore, rien à faire : A_0 est un arbre de recherche bicolore.
- Sinon, la branche de profondeur ℓ sur laquelle a été greffé c présente forcément un sous-arbre B_1 d'un type de l'énoncé :

En effet on remarque qu'une insertion ne se produit jamais en la racine. Considérons le père étiqueté b de la valeur ajoutée.
- Soit b est blanc (on le note alors bl_1) et ce père n'est pas la racine de A_0 (car la racine de A_0 est celle de A donc grise) donc il admet lui-même un père étiqueté a qui est forcément gris puisque A est bicolore. On a donc un des 4 types car il est clair alors que les F_i de l'énoncé sont bicolores de même rang car A est bicolore et le nœud ajouté est blanc et ne modifie pas le comptage des nœuds gris.
- Soit b est gris mais alors c admet forcément un fils direct non vide noté bl_1 blanc sinon A_0 serait bicolore. Donc là encore on a un des 4 types (les F_i sont bien bicolores de même rang).

Notons que cela impose $\ell - r \geqslant 1$ puisque cette branche contient au moins 1 nœud blanc bl_1 provenant de l'arbre initial. Donc a fortiori $p - r \geqslant 1$.
Soit A_1 l'arbre obtenu en remplaçant le sous-arbre B_1 par sa correction blanche C_1, ce sous-arbre C_1 étant bicolore de recherche d'après la question 16). Notons que A_1 est toujours un arbre de recherche.

- Si A_1 est bicolore, c'est terminé et on effectué 1 correction et donc $nb_c \leqslant p - r$ car $p - r \geqslant 1$ comme noté précédemment.
- Si A_1 n'est pas bicolore alors comme C_1 est bicolore c'est forcément que la racine de C_1 n'est pas la racine de A_1 (i.e. $A_1 \neq C_1$) et que son père bl_2 (qui existe donc) est blanc donc n'est pas la racine de A_1 qui est toujours celle de A donc grise. Donc le père e de d existe et est gris
  En outre d'après la remarque 2 de la question 16), A_1 vérifie **P3** avec comme valeur r.
  Il en résulte qu'une des branches du sous-arbre B_2 de racine e contient au moins 2 nœuds blancs provenant de A : bl_1 et bl_2 (donc $p - r \geqslant 2$) et B_2 est de l'un des 4 types.
  Soit A_2 l'arbre obtenu en remplaçant B_2 par sa correction C_2.
  - Si A_2 est bicolore c'est terminé et $nb_c = 2 \leqslant p - r$.
  - Si A_2 n'est pas bicolore alors alors on se trouve exactement dans la même situation qu'avec A_1 mais avec forcément $p - r \geqslant 3$.

    L'itération est claire. Supposons qu'à la fin de la $k^{\text{ième}}$ correction, l'arbre A_k obtenu ne soit pas bicolore. Alors $p - r \geqslant k + 1$. Donc au pire à la fin de la $(p - r)^{\text{ième}}$ correction on obtient un arbre bicolore.

Conclusion finale : Soit A un arbre bicolore de recherche dont on a éventuellement grisé la racine, de profondeur p et de rang r. Il suffit après l'insertion d'une nouvelle valeur d'effectuer au plus $p - r$ corrections blanches du type de l'énoncé pour obtenir un nouvel arbre bicolore de recherche (dont la racine peut parfaitement être blanche). ☐

20) La fonction `correction` appliquée à un arbre a renvoie `a` si `a` n'est pas d'un des 4 types et l'arbre corrigé sinon.

```
let correction a = match a with
  | Noeud (Gris, Noeud (Blanc, Noeud (Blanc, f1, v1, f2), v2, f3), v3, f4)
    -> Noeud (Blanc, Noeud (Gris, f1, v1, f2), v2, Noeud (Gris, f3, v3, f4))
  | Noeud (Gris, f1, v1, Noeud (Blanc, f2, v2, Noeud (Blanc, f3, v3, f4)))
    -> Noeud (Blanc, Noeud (Gris, f1, v1, f2), v2, Noeud (Gris, f3, v3, f4))
  | Noeud (Gris, Noeud (Blanc, f1, v1, Noeud (Blanc, f2, v2, f3)), v3, f4)
    -> Noeud (Blanc, Noeud (Gris, f1, v1, f2), v2, Noeud (Gris, f3, v3, f4))
  | Noeud (Gris, f1, v1, Noeud (Blanc, Noeud (Blanc, f2, v2, f3), v3, f4))
    -> Noeud (Blanc, Noeud (Gris, f1, v1, f2), v2, Noeud (Gris, f3, v3, f4))
  | _ -> a
;;
correction : arbre -> arbre = <fun>
```

21) et 22))
La modification de la fonction d'insertion est alors facile, le fonctionnement de la récursion étant clair.
Compte-tenu de ce qui précède, cette fonction insérera une nouvelle valeur dans un arbre bicolore de recherche dont la racine est grise en respectant à la fois la structure de recherche et celle de coloration. On rappelle que l'arbre produit n'aura pas forcément une racine grise.

```
let rec insertion_abr_bicol v a = match a with
  | Vide -> Noeud (Blanc, Vide, v, Vide)
  | Noeud (c, fg, n, fd) -> match 0 with
      | _ when v = n -> a
      | _ when v < n -> correction (Noeud (c, insertion_abr_bicol v fg, n, fd))
      | _            -> correction (Noeud (c, fg, n, insertion_abr_bicol v fd))
;;
insertion_abr_bicol : int -> arbre -> arbre = <fun>
```

Comme déjà noté le nombre d'appels récursifs est un $O(p)$ et le nombre de corrections effectives (rappelons que la fonction `correction` ne fait éventuellement rien) est majoré par $p - r$.

Vérifions cette fonction avec l'exemple proposé :

```
insertion_abr_bicol 9 exemple;;
- : arbre =
 Noeud
  (Blanc,
   Noeud
    (Gris,
     Noeud
      (Blanc, Noeud (Gris, Vide, 1, Vide), 3, Noeud (Gris, Vide, 4, Vide)),
     6, Noeud (Gris, Vide, 7, Vide)),
   8,
   Noeud
    (Gris, Noeud (Gris, Vide, 9, Vide), 10, Noeud (Gris, Vide, 11, Vide)))
```

C'est bien l'arbre obtenu en question 17.

FIN