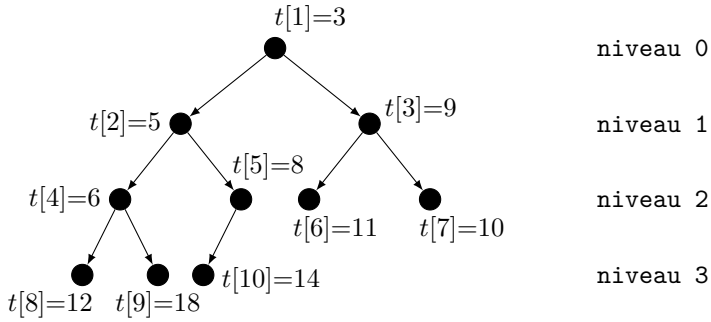
1

Arborescences

« *Un père, des fils. Une racine, des noeuds, des feuilles.* »

1.1 Tas

Un arbre binaire complet est un graphe comme celui de la figure : tous les niveaux sont remplis de gauche à droite, sauf éventuellement le dernier niveau. On numérote les sommets de 1 à n, niveau par niveau, et pour chaque niveau, de gauche à droite. S'il y a une flèche de i vers j, on dit que i est le père de j, et que j est un fils de i. Le sommet d'en haut qui n'a pas de père (niveau 0), est la racine. Les sommets du dernier niveau, qui n'ont pas de fils, sont des feuilles. À chaque sommet numéro i, on associe un attribut entier. On représente alors un arbre binaire complet de taille n par un tableau de taille $n + 1$ dont la première case contient le nombre d'éléments du tas.



Question 1

Pour i donné, $1 \leq i \leq n$, quel est son père, et quels sont ses fils (s'ils existent) ? Un tas est un arbre binaire complet dans lequel tout père est plus petit que ses fils (comme

pour la figure). Le minimum est donc à la racine. Écrire une fonction OCaml qui vérifie si un arbre binaire complet est un tas.

Question 2

Écrire une fonction OCaml qui supprime la racine d'un tas à n sommets et qui renvoie un tas composé des $n-1$ sommets restants (indication : on pourra mettre la dernière feuille à la place de la racine et la faire descendre).

Question 3

Comment insérer un nouvel élément dans un tas à n sommets ?

CORRIGÉ

Question 1

Le père du sommet i est $\lfloor i/2 \rfloor$ pour $2 \leq i \leq n$ (le père du sommet 1 n'est pas défini).
Les fils de i sont $2i$ et $2i+1$ si ces valeurs sont $\leq n$. En particulier :

- si n est le nombre de sommets et $2i = n$, alors i n'a qu'un fils, à savoir n
- si $i > \lfloor n/2 \rfloor$, alors i est une feuille

Pour la fonction demandée, on vérifie les attributs des fils des sommets qui ne sont pas des feuilles, et on sort dès qu'il y a un échec :

```
let verif (t:int array) : bool =
  let n = t.(0) in
  try
    for i=1 to n/2 do
      if t.(i) > t.(2*i) then raise Exit;
      if 2*i+1 <= n && t.(i) > t.(2*i+1) then raise Exit;
    done;
    true
  with Exit -> false
```

Question 2

Une idée naturelle serait d'essayer de remonter le plus petit des fils de la racine à sa place, puis de continuer ainsi... mais on aboutit à un déséquilibre de l'arbre, qui n'est plus complet. Comme l'indication le suggère, il est plus simple de placer le dernier élément à la racine puis de le descendre à la bonne place (percolation basse) :

```
let fils_min (t:int array) (i:int) : int =
  let n = t.(0) in
  let a_fils_droit = 2*i+1 <= n in
  if a_fils_droit then
    if t.(2*i) < t.(2*i+1) then
      2*i else 2*i+1
  else 2*i
```

```
let rec percolation_basse (t:int array) (i:int) : unit =
  let n = t.(0) in
  let a_fils_gauche = 2*i <= n in
  if a_fils_gauche then
    let fils = fils_min t i in
    if t.(fils) < t.(i) then (
      let tmp = t.(i) in
      t.(i) <- t.(fils);
      t.(fils) <- tmp;
      percolation_basse t fils
    )

let supprimer_racine (t:int array) : int =
  let n = t.(0) and rac = t.(1) in
  t.(1) <- t.(n);
  t.(0) <- n-1;
  percolation_basse t 1;
  rac
```

La procédure a une complexité au pire proportionnelle au nombre de niveaux de l'arbre soit $\lceil \log_2 n \rceil$.

Question 3

Comme précédemment, une idée simple est d'ajouter la nouvelle valeur en dernière position, puis de la remonter tant que besoin est (percolation haute) :

```
let rec percolation_haute (t:int array) (i:int) : unit =
  let parent = i/2 in
  if i <> 1 && t.(parent) > t.(i) then (
    let tmp = t.(i) in
    t.(i) <- t.(parent);
    t.(parent) <- tmp;
    percolation_haute t parent
  )

let ajouter (t:int array) (e:int) : unit =
  assert(Array.length t > t.(0) + 1);
  t.(0) <- t.(0) + 1;
  let n = t.(0) in
  t.(n) <- e;
  percolation_haute t n
```

Ici encore, la fonction a une complexité proportionnelle au nombre de niveaux de l'arbre, donc en $O(\lceil \log_2 n \rceil)$.

Signalons que la structure de tas est à la base d'une méthode de tri très performante : le tri par tas. L'idée est simple : on part du tableau `a` de n éléments à trier, et on construit un tas par ajout successif des n éléments. On retire ensuite toutes les racines et on obtient ainsi les éléments dans l'ordre :

```
let tri_par_tas (a:int array) : unit =
  let n = Array.length a in
  let tas = Array.make (n+1) 0 in
  Array.iter (ajouter tas) a;
  for i=0 to n-1 do
    a.(i) <- supprimer_racine tas
  done
```

Le coût de chaque insertion ou suppression est proportionnel à la hauteur du tas courant ($\log_2 p$ pour p éléments), et donc le coût total est de l'ordre de :

$$\sum_{p=1}^{n} \log_2 p = \log_2 n! = O(n \log_2 n)$$

(d'après la formule de Stirling ou par comparaison avec $\int_1^n \log x \, \mathrm{d}x$). Le tri par tas est donc asymptotiquement très rapide.

⋆⋆⋆⋆⋆⋆⋆⋆

1.2 Arbre bicolore

Question 1

Un arbre binaire étiqueté est défini par une racine, des nœuds et des feuilles. Un nœud est défini par un père, un fils gauche, un fils droit, un étiquette (entier positif) et une couleur (noir ou rouge). La racine est un nœud sans père et une feuille est un nœud vide (autrement dit, fils vide d'un nœud interne).

Expliquer quelle structure de donnée utiliser pour représenter un arbre binaire étiqueté en OCaml.

Question 2

Un arbre binaire est de recherche (ABR) si chaque nœud est tel que : tout nœud de sa sous arborescence droite ne contient que des valeurs d'étiquettes supérieures et tout nœud de sa sous-arborescence gauche des valeurs d'étiquettes inférieures.

Donne une fonction permettant d'insérer une nouvelle valeur en conservant la structure d'ABR.

Question 3

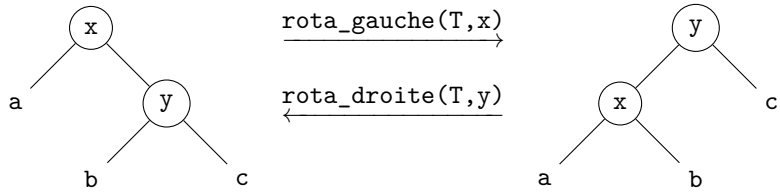
Un arbre bicolore est un arbre binaire de recherche qui vérifie les propriétés suivantes : chaque nœud est soit noir, soit rouge, chaque feuille est considérée comme noire, si un nœud est rouge alors ses deux fils sont noirs et tous les chemins d'un nœud (x) à une feuille de sa descendance contiennent le même nombre $\mathbf{np}(x)$ de nœuds noirs (non-compris le nœud lui-même). Ce nombre est appelé la **noire-profondeur** du nœud.

Nous définissons la **noire-profondeur d'un arbre bicolore** comme étant la noire-profondeur de sa racine.

Montrer que la profondeur (longueur de la plus longue branche, *racine* → *feuille*) d'un arbre bicolore composé de n nœuds internes (i.e qui ne sont pas des feuilles) est d'au plus $2\log_2(n+1)$.

Question 4

Nous désirons effectuer les rotations suivantes, où T représente l'arbre :



Écrire la fonction `rota_gauche`.

Montrer que cette fonction préserve la structure d'arbre binaire de recherche. En est-il de même de la structure d'arbre rouge-noir ?

Question 5

Comment insérer un nouvel élément dans un arbre bicolore en conservant sa structure ?

CORRIGÉ

Question 1

On représente un arbre bicolore avec le type récursif suivant :

```
type arn =
  | Nil
```

Question 2

Pour insérer un nouvel élément dans un arbre binaire de recherche en conservant la propriété fondamentale des ABR, nous parcourons l'arborescence en partant de la racine avec comme critère de choix pour le nœud suivant le fils droit si l'étiquette du nouvel élément est plus grande que celle du nœud courant et le fils gauche sinon.

En OCaml, on peut écrire :

```
  | N of arn * int * arn

let rec insere_arn (t:arn) (e1:int) : arn = match t with
  | Nil -> R (Nil, e1, Nil)
  | R (fg, e2, fd) ->
```

Question 3

L'arbre bicolore minimal pour une noire-profondeur np donnée n'est composé que de nœuds noirs. Comme chaque branche partant de la racine a exactement np nœuds noirs, nous avons un arbre complet équilibré de profondeur np. Chaque niveau i de cet arbre avant tout binaire contient 2^i nœuds.

Ainsi, un arbre bicolore de noire-profondeur np est composé d'au moins $2^{np} - 1$ nœuds noirs.

Si n est le nombre de nœuds d'un arbre bicolore de noire-profondeur np, nous avons

$$n \geq 2^{np} - 1$$

autrement dit,

$$\log_2(n+1) \geq np$$

Si l'on tient compte du fait que, sur une branche, au plus un nœud sur deux est rouge, la profondeur d'un arbre bicolore de noire-profondeur np est au plus $2np$ c'est à dire d'au plus $2\log_2(n+1)$.

Question 4

Pour alléger les fonctions et puisque la coloration n'apporte rien ici, on définit un type d'arbre binaire sans couleurs :

```
type abr = Nil | Noeud of abr * int * abr
```

On propose les fonctions de rotations gauche et droite autour de la racine de `t` (la transformation n'est pas effectuée si elle est impossible) :

```
let rotate_right (t:abr) : abr = match t with
  | Noeud(Noeud(t1, u, t2), v, t3) -> Noeud(t1, u , Noeud(t2, v, t3))
  | _ -> t

let rotate_left (t:abr) : abr = match t with
  | Noeud(t1, u, Noeud(t2, v, t3)) -> Noeud (Noeud(t1, u, t2), v, t3)
  | _ -> t
```

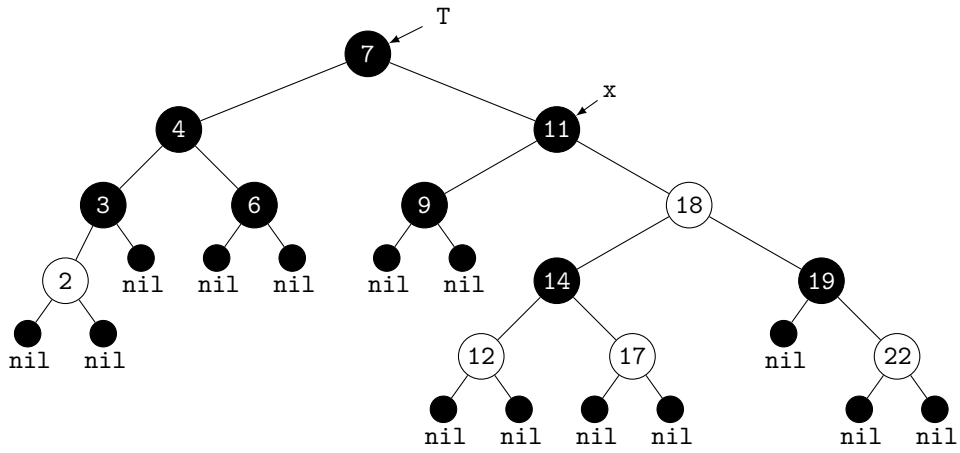
Si l'on souhaite effectuer une rotation autour d'un nœud en particulier, il suffit de descendre jusqu'à celui-ci de manière récursive puis de le remplacer, similairement à ce qui est fait pour l'insertion :

```
let rec rotate_node_left (t:abr) (x:int) : abr = match t with
  | Noeud(t1, y, t2) -> if y = x then rotate_left t
    else if x < y then Noeud(rotate_node_left t1 x, y, t2)
    else Noeud(t1, y, rotate_node_left t2 x)
  | _ -> t
```

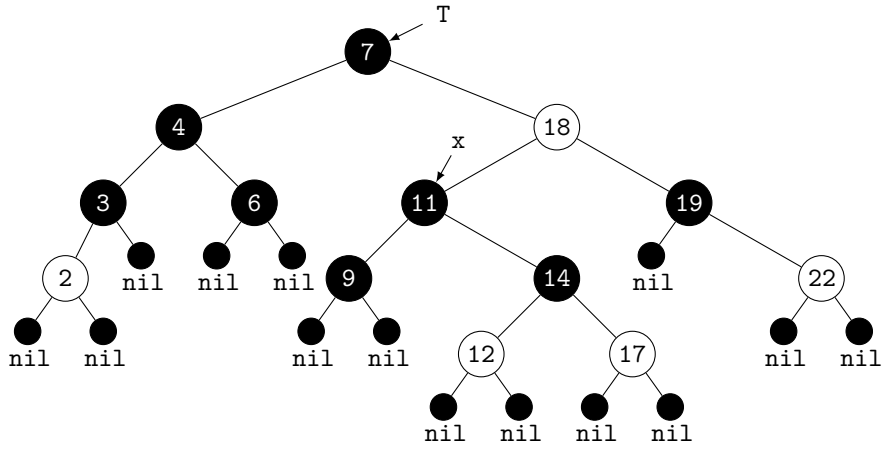
La fonction `rotate_node_right` s'écrit de manière analogue.

La structure d'arbre binaire de recherche est conservée : x et y sont du même côté par rapport au père de x et les étiquettes de la sous-arborescence gauche de y sont supérieures à celle de x et inférieures à celle de y, elle peut devenir la sous-arborescence droite de x sans que l'arbre ne perde sa propriété d'arbre binaire de recherche.

En revanche, la propriété d'arbre bicolore n'est pas forcément conservée. Si nous appliquons l'algorithme précédent à l'arbre ci-dessous où les nœuds noirs sont colorés en noir et les nœuds rouges sont représentés en blanc :



Nous obtenons l'arbre suivant :



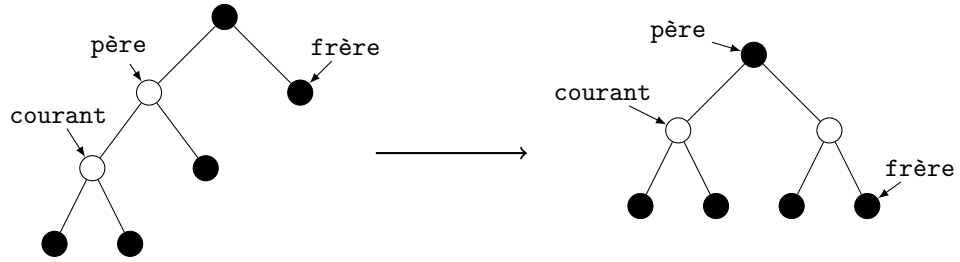
Cet arbre ne vérifie plus la propriété d'arbre bicolore, le nombre de nœuds noirs n'est plus le même sur toutes les branches partant de la racine.

Question 5

L'insertion dans un arbre bicolore peut se faire selon le schéma suivant :
On insère le nouvel élément dans l'ABR en utilisant la fonction de la question 2 ; on obtient donc un arbre binaire de recherche.
Pour que cet arbre vérifie la propriété de bicolore, nous donnons à ce nouveau nœud la couleur rouge. Il devient le nœud courant. Si son père est noir, l'arbre est bicolore et c'est terminé. Sinon, si le frère de ce père est rouge, ce père et son frère deviennent noirs et leur père devient rouge. On recommence avec le père commun. Par contre, si le

frère du père du nœud courant est noir, il y a au moins un élément de plus du côté du nœud courant. Dans ce cas, nous plaçons le nœud courant et son père suivant le même type de filiation que le père et le grand-père : le père devient noir et le grand-père rouge puis nous effectuons une rotation dans le sens du père vers son grand père.

On représente ici l'une des quatre situations où une rotation est nécessaire :



Nous obtenons ainsi les fonctions OCaml suivantes où l'on n'effectue pas de rotation de manière « explicite » mais où on utilise une fonction de *pattern-matching* pour gérer les cas où deux nœuds rouges se suivent :

```
let corrige_rouge (t:arn) : arn = match t with
  | N (R (R (a, x, b), y, c), z, d)
  | N (R (a, x, R (b, y, c)), z, d)
  | N (a, x, R (R (b, y, c), z, d))
  | N (a, x, R (b, y, R (c, z, d)))
    -> R (N (a, x, b), y, N (c, z, d))
  | t -> t

let rec insere_aux (t:arn) (x:int) : arn =
  match t with
  | Nil -> R (Nil, x, Nil)
  | R (fg, y, fd) ->
    if x = y then t
    else if x > y then corrige_rouge (R(fg, y, insere_aux fd x))
    else corrige_rouge (R(insere_aux fg x, y, fd))
  | N (fg, y, fd) ->
    if x = y then t
    else if x > y then corrige_rouge (N(fg, y, insere_aux fd x))
    else corrige_rouge (N(insere_aux fg x, y, fd))
```

Notons que pour éviter de répéter deux fois la même chose, on pourrait définir une fonction constructeur comme suit :

```
let cons (t:arn) (fg:arn) (x:int) (fd:arn) : arn = match t with
  | N _ -> N(fg, x, fd) | R _ -> R(fg, x, fd ) | _ -> t
```

Nous disposons d'un arbre binaire de recherche dont la profondeur est d'au plus $2\log_2(n+1)$, avec des algorithmes d'insertion et de suppression qui parcourent un nombre donné de fois une branche de l'arborescence. Nous disposons ainsi d'une structure de données où toutes les opérations peuvent se faire en un temps logarithmique suivant la taille de cette base de données.

⋆⋆⋆⋆⋆⋆⋆⋆