Arbres (deuxième année)

C. Charignon

Table des matières

Première partie
Cours

1 Révisions et compléments

1.1 Parcours en profondeur et en largeur

Le cours a été fait en MPSI. On propose deux exercices de révision : on peut reconstruire un arbre à partir d'une énumération de ses sommets, à condition de connaître le type de parcours employé et le type de chaque noeud (feuille ou noeud intérieur). Voir l'exercice 3.

Et pour un exemple de parcours en largeur : exercice 1.

1.2 Commentaires sur les relations d'ordre

On peut réaliser un `'a` arbre binaire de recherche dès que `'a` est un type muni d'un ordre total (deux éléments peuvent toujours êtres comparés). Les types totalement ordonnés les plus fréquents sont les entiers, les flottants, les chaînes de caractères, et les n-uplets formés à partir de types totalement ordonnés. Ces derniers sont ordonnés par l'ordre lexicographique : on compare d'abord la première coordonnée, en cas d'égalité la seconde, etc...

Par ailleurs, Caml crée par défaut un ordre sur les type définis par l'utilisateur. Les types somme (par exemple `type truc = Machin  Chose | Bidule ;;|`) sont ordonnés selon l'ordre dans lequel on les a tapés. Ainsi, sur l'exemple précédent, |Machin < Chose < Bidule|.

Au final presque n'importe quel type est muni d'une relation d'ordre. Exception notoire : les fonctions.

D'un point de vue théorique, puisque n'importe quel donnée est enregistrée comme une suite de 0 et de 1, il est toujours possible de créer une relation d'ordre en comparant le nombre écrit en base 2 par cette suite de bits.

1.3 Arbres binaires de recherche

Une fonction un peu plus difficile que celles vues en première année : supprimer un élément d'un ABR. Nous reprenons le type :

```
type 'a arbre = Vide | Noeud of ('a arbre * 'a * 'a arbre);;
```

On peut penser à deux stratégies : on peut écrire une fonction auxiliaire `fusionABRDisjoints` qui fusionne deux ABR a et b tels que toutes les étiquettes de a sont inférieures aux étiquettes de b.

Ou alors lors de la suppression de l'étiquette d'un nœud on la remplace par le maximum du fils gauche, ou le minimum du fils droit.

```
let rec fusion_ABR_disjoints a b =
    (* Fusionne a et b, en supposant que toutes les étiquettes de a sont <= à toutes les
    étiquettes de b. *)
    match a, b with
        |Vide, _  -> b
        |_, Vide  -> a
        |Noeud(fg1, e1, fd1), Noeud(fg2, e2, fd2) ->
                (* NB : on a e1 <= e2 par la précondition *)
                Noeud(fg1, e1, Noeud(fusion_ABR_disjoints fd1 fg2, e2, fd2))
;;
```

```
let rec extrait_min = function
  (* renvoie le couple (min de 'labr, 'labr privé de ce min) *)
  |Vide -> failwith "arbre vide"
  |Noeud(Vide, e, fd) -> (e, fd)
  |(Noeud fg, e, fd) ->
      let mini, fg_sans_mini = extrait_min fg in
      (mini, Noeud(fg_sas_mini, e, fd))
;;

let rec fusion_ABR_disjoints a b =
  match a, b with
  |_, Vide  -> a
```

```
  |_ -> let e, fd_sans_e = extrait_min b in
      Noeud(b, e, fd_sans_e )
;;
```

Bien que la complexité de ces deux fonctions soit similaire, l'une des deux est clairement à favoriser... Laquelle ?

D'où le programme principal :

```
let res supprimeABR x a =
    match a with
        |Vide -> Vide
        |Noeud(fg, e, fd) when e<x -> Noeud(fg, e, supprimeABR x fd)
        |Noeud(fg, e, fd) when e>x -> Noeud( supprimeABR x fg, e, fd)
        |Noeud(fg, _, fd)     -> fusion_ABR_disjoints fg fd
```

N.B. Nous supposons que `x` n'apparaît qu'une seule fois dans `a`. Si ce n'était pas le cas, nous n'aurions enlevé que le premier `x` rencontré, et il pourrait en rester d'autres.

Pour l'autre version, commençons par écrire une fonction qui extrait le minimum :

```
let rec extraitMin a=
   (* Renvoie le couple (min de a, reste de a) *)
   match a with
    |Vide-> failwith "arbre Vide"
    |Noeud(Vide, e, fd) -> e, fd
    |Noeud(fg, e, fd)  -> let m, r = extraitMin fg in
        (m, Noeud(r, e, fd))
 ;;
```

D'où la fonction principale :

```
let rec supprimeABR x a=
    match a with
        | Vide -> Vide
        |Noeud(fg, e, fd) when e<x -> Noeud(fg, e, supprimeABR x fd)
        |Noeud(fg, e, fd) when e>x -> Noeud( supprimeABR x fg, e, fd)
        |Noeud(fg, _, fd) ->
            let m, r =extraitMin fd in Noeud(fg, m, r)
;;
```

Dans la même veine, voir l'exercice 5.

2 Tas

On étudie à présent un autre type d'arbre qui aura les avantages suivants :

- La recherche du maximum est très facile ;
- On peut contrôler sa hauteur maximale ;
- On peut réaliser facilement une version mutable de ce type au sein d'un simple tableau.

2.1 Définition

Définition 2.1. *Soit a un arbre binaire étiqueté. On dit que c'est un tas lorsque pour chaque nœud n, les étiquettes des fils de n sont inférieures à l'étiquette de n.*

On comprend pourquoi il est facile de trouver le maximum : c'est l'étiquette de la racine !

Remarques :

- Ce type de tas est parfois appelé un « tas-max », ou « maximier ». On peut de même définir une notion de « tas-min », ou « minimier » pour lequel c'est le minimum qui serait à la racine.
- En pratique si vous avez sous la main une structure de tas-max mais que vous auriez voulu des tas-min, vous pouvez juste changer le signe de toutes les étiquettes (s'il s'agit de nombres).
- Certains auteurs imposent de plus que les feuilles soient toutes de hauteur h ou $h-1$, et que les feuilles de hauteurs h soient toutes « à gauche ». Ceci sera utile pour la réalisation de tas mutable (voir plus loin). Un arbre vérifiant la condition sur l'ordre des étiquettes mais pas sur le squelette sera alors appelé « arbre de tournoi ».

  C'est l'occasion de rappeler que de nombreuses définitions ne sont pas fixées par le programme officiel, et peuvent varier d'un sujet à l'autre. Soyez donc attentifs à la partie au début du sujet qui donne les définitions.

2.2 Files de priorité

Une file de priorité est une structure de donnée permettant :

- d'insérer une donnée, en lui associant un nombre (ou autre objet d'un type totalement ordonné), auquel on pense comme étant sa « priorité » ;
- et d'extraire l'élément de priorité maximale.

Ainsi, un tas est un bon moyen de réaliser une file de priorité. Il suffit de faire en sorte que la relation d'ordre utilisée soit la comparaison des priorités.

2.3 Tas persistants

Le plus naturel a priori pour programmer une structure de tas est d'utiliser la structure d'arbre binaire persistant que nous connaissons déjà.

Exercice : Programmer une fonction `estUnTas` qui indique si un arbre binaire est en tas.

Pour pouvoir prétendre disposer d'une structure de tas, nous devons programmer les deux fonctions suivantes :

- `insereTas: 'a -> 'a arbreBinaire -> 'a arbreBinaire` ;
- `extraitMax : 'a arbreBinaire -> 'a * 'a arbreBinaire`.

Nous n'allons pas programmer ces fonctions de manière trop naïve : nous allons créer une structure de tas *auto-équilibrés*. Cela signifie qu'un tas construit uniquement à l'aide des fonctions ci-dessus sera toujours équilibré. En particulier, sa hauteur sera fonction logarithmique de son nombre de nœuds, ce qui nous assure que toutes les fonctions ci-dessus auront un coût logarithmique en fonction du nombre de données.

2.3.1 Insertion

Le principe de la fonction `insereTas` est le suivant : pour insérer un élément x dans `Noeud(fg, e, fd)`, nous l'insérons dans le fils gauche, puis nous échangeons les deux fils. Autrement dit, nous renvoyons `Noeud(fd, e, insere x fg)` ou `Noeud(fd, x, insere e fg)` selon que $x < e$ ou l'inverse.

```
let rec insertion x t=
  (* Entrée : un tas t, un élément x
      Sortie : un tas contenant les éléments de t et x
   *)
  match t with
  | Vide -> Noeud(Vide, x, Vide)

  | Noeud(fg, e, fd) when x>e -> (* x est le max du nouveau tas *)
     Noeud(
   fd,
   x,
   (insertion e fg)
       )

  |  Noeud(fg, e, fd) -> (* Et dans ce cas, c'est e le max *)
      Noeud(
    fd,
    e,
    (insertion x fg)
  )
;;

let rec tas_of_list = function
  | [] -> Vide
  | t::q -> insertion t (tas_of_list q)
;;

(* Ou en fonctionnel : *)
let tas_of_list l =
   List.fold_right insertion l Vide;;
```

Proposition 2.2. *Soit $n \in \mathbb{N}$ et t un tas obtenu en insérant $2^{n+1}-1$ élément dans* `Vide` *avec la fonction* `insere`*. Alors t est un arbre parfait de hauteur n.*

Plus généralement, tout tas obtenu en insérant à l'aide de la fonction `insere`*, moins de $2^{n+1}-1$ éléments dans* `Vide` *est un arbre complet de hauteur moins de n.*

Démonstration :

Remarquons que dans la fonction `insere` il y a deux cas, selon que l'élément à insérer doit être mis à la racine ou non. Cependant, dans les deux cas, le squelette du tas obtenu est la même. Il n'y a donc pas lieu ici de distinguer ces cas.

On procède par récurrence.

L'initialisation est claire : le tas obtenu en insérant un élément dans `Vide` est réduit à une feuille, c'est bien un arbre parfait de hauteur 0.

Pour tout arbre a et élément x, notons $x + a$ le résultat de `insere_tas x a`.

Fixons $n \in \mathbb{N}$, et supposons que tout tas obtenu en insérant $2^{n+1}-1$ feuilles dans `Vide` est parfait de hauteur n. Prenons maintenant $2^{n+2}-1$ éléments $(a_0, ..., a_{2^{n+2}-2})$ et insérons les dans `Vide`. Une fois le premier élément inséré, la racine est formée. Ensuite, on insère successivement un élément dans le fils gauche de la racine, puis un élément à droite.

Le fils gauche est alors $Vide + a_1 + a_3 + ... + a_{2^{n+2}-3}$ et le fils droit $Vide + a_2 + ... + a_{2^{n+2}-2}$. Ces deux arbres sont obtenus à partir de `Vide` en insérant $\frac{2^{n+2}-2}{2}$ c'est-à-dire $2^{n+1}-1$ éléments. Par hypothèse de récurrence, ce deux arbres parfaits de hauteur n. Et l'arbre total est bien parfait de hauteur $n+1$.

On prouve la généralisation par une récurrence similaire. □

Corollaire 2.3. *Soit t obtenu par la fonction* `tas_of_list` *à partir d'une liste de N éléments. Alors la hauteur de t est au plus $\log_2(N)+1$.*

En particulier, la complexité de `tas_of_list` *est en $O(N \log N)$.*

2.3.2 Extraction du maximum

Pour supprimer le maximum, on remarque qu'il suffit d'écrire une fonction qui fusionne deux tas.

```
let rec rassemble t1 t2 =
  (* Rassemble deux tas en un nouveau tas *)
  match t1, t2 with
  | Vide, _ -> t2
  | _, Vide -> t1

  | Noeud(fg1, e1, fd1), Noeud(fg2, e2, fd2) when e1 <= e2 ->
     (* le max du nv tas est e2 *)
     Noeud(t1, e2, rassemble fg2 fd2)

  | Noeud(fg1, e1, fd1), Noeud(fg2, e2, fd2) ->
     (* Le nv max est e1 *)
     Noeud( rassemble fg1 fd1, e1, t2)
;;


let max_extrait = function
  | Vide -> failwith "tas vide"
  | Noeud(fg, m, fd) ->
     m, rassemble fg fd
;;
```

Lemme 2.4. *Lors de l'extraction du maximum d'un tas, le nouveau tas obtenu est de hauteur au plus égale à celle du tas initial.*

Démonstration : Récurrence sur la hauteur. □

On ne le prouvera pas, mais on peut montrer que pour n'importe quelle suite d'insertion et de suppression de maximum dans un tas d'au plus N éléments, la complexité amortie de chaque opération est en $O(\log N)$.

2.4 Application : le tri par tas

Partant d'une liste, nous pouvons tout simplement entasser tous ses éléments dans un tas, puis les extraire du plus grand au plus petit.

```
let rec detasse t =
    (* Renvoie la liste des éléments de t, dans l'ordre.*)
  match t with
  | Vide -> []
  | _ -> let max,suiteTas = max_extrait t in
            max::(detasse suiteTas)
;;


let tri_par_tas l=
   let tas = tas_of_list l in

   (* Maintenant il faut détasser le tas *)
  List.rev (detasse tas)
;;
(* Pour éviter le List.rev, on aurait pu utiliser des tas-min *)
```

Notons n le nombre d'éléments à trier. Nous avons vu que la création du tas se fait en $O(n \log n)$. Et nous obtenons un tas de hauteur au plus $\log_2(n) + 1$. Par le lemme 2.4, la hauteur du tas restera toujours inférieure à $\log_2(n) + 1$ par la suite. Dès lors, la complexité de chaque extraction est en $O(\log n)$. Comme il y a n extractions, la complexité de `list_of_tas` est en $O(n \log n)$. Enfin le renversement final est en $O(n)$, négligeable devant l'entassement et le détassement.

Au final, le tri par tas est en $O(n \log n)$. Il s'agit donc d'un tri efficace. Plus compliqué à programmer que le tri fusion, il aura tout de même des intérêts, comme nous le verrons par la suite.

Remarque : On peut faire la même chose avec des ABR au lieu des tas (voir le cours de MPSI). Cependant, comme nous n'avons pas programmé d'insertion dans un ABR qui fasse en sorte que l'arbre reste toujours équilibré, nous

n'avons aucune garantie que les insertions et les extractions soient en $O(\log n)$, de sorte que dans les mauvais cas, le tri obtenu serait en $O(n^2)$. Et ce mauvais cas est obtenu en particulier lorsque la liste initiale était déjà triée, dans l'ordre croissant ou décroissant.

2.5 Tas modifiables

Pour obtenir des tas modifiables, nous allons les enregistrer dans des tableaux, au lieu d'utiliser la structure d'arbre binaire que nous avons déjà programmée.

En plus de permettre d'obtenir une structure de tas mutable, cela va nous permettre d'obtenir toujours des arbres «complets à gauche», c'est-à-dire que tous les niveaux de l'arbre seront complets, sauf éventuellement le dernier, pour lequel toutes les feuilles seront rangées «à gauche».

Ainsi, la hauteur sera toujours la plus petite possible. Précisément, en notant h la hauteur d'un tel arbre et n son nombre de nœuds, on aura $2^h \leqslant n \leqslant 2^{h+1} - 1$. En particulier, $h \leqslant \log_2(n)$, ce qui garantit une complexité logarithmique pour les opérations de base.

L'idée est de numéroter les nœuds de l'arbre selon un parcours en largeur.

Comme l'arbre sera toujours complet gauche, il y a une formule très simple pour obtenir l'indice des fils d'un nœud. Voici la formule si on décide de commencer la numérotation à 1 :

Proposition 2.5. *Soit a un arbre binaire complet gauche dont les nœud sont numérotés par un parcours en largeur en commençant à 1.*

Soit n un nœud de a ayant un fils gauche et i son numéro. Alors le numéro de son fils gauche est $2i$. Si de plus n à un fils droit, alors le numéro de ce dernier est $2i + 1$.

Démonstration : Notons p la profondeur de n. Au étages précédents, il y a donc $2^p - 1$ nœuds. Donc le premier nœud de l'étage p porte le numéro 2^p. Donc le nombre de nœuds de l'étage p situés à gauche de n est $i - 2^p$.

Comme n à un fils gauche, et que l'arbre est complet gauche, tous les nœuds à gauche de n ont deux fils, ce qui fait $2 \times (i - 2^p)$ fils. Notons fg le fils gauche de n. Il y a donc $2 \times (i - 2^p)$ nœuds à gauche de fg.

Comme il y a $2^{p+1} - 1$ nœuds aux étages 0 à p (inclus), le numéro de i est $2^{p+1} - 1 + 2 \times (i - 2^p) + 1$, soit i.

Et le fils droit de n est le nœud suivant dans la numérotation, c'est donc le numéro $2i + 1$. □

Corollaire 2.6. *Avec les notations précédente, soit n un nœud différent de la racine et i son numéro. Alors le numéro de son père est $\left\lfloor \frac{i}{2} \right\rfloor$.*

N.B. On voit pourquoi il va être facile de maintenir le fait que l'arbre est complet gauche : si n est le nombre d'éléments du tas, le prochain emplacement à utiliser correspond à la case d'indice n du tableau. Et si on veut supprimer un élément, il faudra vider la case d'indice $n - 1$ du tableau.

La numérotation à partir de 1 présente une particularité intéressante : si i est le numéro d'un nœud, nous venons de voir que son fils gauche a pour numéro $2i$: en base 2 il suffit d'ajouter 0. De même, pour obtenir son fils droit il suffit d'ajouter 1 à son écriture en base 2.

Soit encore i le numéro d'un nœud. Dans la représentation d'un tas selon un arbre binaire, on peut alors accéder à l'élément i en lisant simplement son écriture en base 2 : on élimine le premier 1 (numéro de la racine), et ensuite lorsqu'on lit un 1 on descend dans la branche gauche, et lorsqu'on lit un 1, on descend dans la branche droite. Voir l'exercice 12 où cette remarque permet de construire un tas, enregistré sous forme d'arbre binaire, qui soit toujours complet gauche.

Voyons maintenant les formules si on numérote les nœuds à partir de 0. Soit n un nœud, i son numéro dans la numérotation commençant par 1, et j son numéro en commençant par 0. Notons i_g, i_d, j_g, j_d les numéros des fils gauche et droit dans chaque numéro. On a donc $j = i - 1$, puis $i_g = 2i = 2j + 2$. Et enfin $j_g = i_g - 1 = 2j + 1$. Et de même, $j_d = 2j + 2$. Et la formule pour retrouver le père est $j = \left\lfloor \frac{j-1}{2} \right\rfloor$.

Cette numérotation donne des formules un rien plus compliquées mais permet de mieux coller à la numérotation habituelle des tableaux dans Caml et Python.

En général, on utilise un tableau plus grand que le nombre d'éléments du tas. En particulier, ceci permet d'ajouter ultérieurement des éléments. Si on veut garder la longueur du tas sous la main, il suffit d'utiliser un enregistrement :

```
type 'a tasMutable = { longueur : int; donnees : 'a array };;
```

En outre, les tas ainsi programmés auront une taille maximale. Si on veut régler ce problème, on peut utiliser des tableaux redimensionnables, comme le type `list` de Python.

On va programmer les opérations suivantes :

- `estUnTas : 'a tasMutable -> bool`
- `nouveauTas : unit -> 'a tasMutable`
- `estVide : 'a tasMutable -> bool`
- `maxTas : 'a tasMutable -> 'a`
- `extraitMax : 'a tasMutable -> 'a`
- `entasse : 'a -> 'a tasMutable -> unit`

Les fonctions principales sont `extraitMax` et `entasse` ; elles sont basées sur les deux procédures clef `remonteASaPlace` et `descendASaPlace` (en anglais « sifdown » et « siftup »). Les deux prennent en entrée un tableau contenant presque un tas : seul un élément est mal placé. Pour la première fonction, cet élément est trop bas, pour la deuxième il est trop haut. Dans les deux cas il sera amené à la place adéquate pour que l'arbre représenté par la tableau devienne un tas.

```
type 'a tasMutable= {
  mutable longueur : int;
  donnees : 'a array
  };;

(* Rema : une limitation : il y a un maximum au nb de données qu'on peut enregistrer
dans le tas : c'est la longueur de t.donnees
 Pour régler ce problème, utiliser un tableau redimensionable.
 *)

let nouveauTas n x =
  (* Renvoie un nouveau tas, avec de la place pour n éléments, dont le type devra être
  le type de x.*)
  {longueur=0; donnees = Array.make n x };;
```

Passons à la fonction pour remonter un élément, de laquelle on déduit la fonction d'insertion d'un nouvel élément. Je la trouve plus pratique en récursif. Pour les spécifications, j'utilise la notation Python avec les « : » pour décrire des tranches de tableaux.

```
let transpose tab i j=
    let tmp=tab.(i) in
    tab.(i) <- tab.(j);
    tab.(j) <- tmp
;;

let rec remonteASaPlace tab k =
    (* Entrée : un tableau qui représente un arbre binaire.
       Précondition : la portion tab.(0:k) représente un tas.
       Effet : après exécution, tab.(0:k+1) représente un tas.
       *)
      if k=0 || tab.(k) <= tab.( (k-1)/2) then () (* l'élément k est bien placé *)
      else begin
          transpose tab k ((k-1)/2); (* On échange avec son père *)
          remonteASaPlace tab ((k-1)/2)
          end
;;

let insere x t =
    let n = t.longueur in
    t.donnees.(n) <- x; (* On remplit la prochaine case vide *)
    t.longueur <- t.longueur +1;
    remonteASaPlace t.donnees n
;;
```

La procédure pour descendre un élément suit le même principe ; elle est juste un peu plus compliquée car il y a trois cas à considérer : l'élément peut être bien placé, ou devoir être descendu à gauche, ou devoir être descendu à droite. Attention à l'ordre des tests pour obtenir un code clair...

```
let rec descendASaPlace tab k n =
    (*
    tab est un tableau tel que tab.(0:n) représente un arbre binaire.
```

```
    Précondition : Les sous-arbres enracinnés en 2k et en 2k+1 sont des tas.
    Effet : à la fin, l'arbre enraciné en k est un tas.
    *)

    (* cas 1 : on doit descendre tab.(k) vers son fils droit *)
    if 2*k+2 < n &&  tab.(2*k+1) <= tab.(2*k+2) && tab.(2*k+2) > tab.(k) then begin
        transpose tab k (2*k+2);
        descendASaPlace tab (2*k+2) n
    end

    (* cas 2 : on descend à gauche *)
    else if 2*k+1<n && tab.(k) < tab.(2*k+1) then begin
        transpose tab k (2*k+1);
        descendASaPlace tab (2*k+1) n
    end

    (* cas 3 : tab.(k ) est bien placé *)
    else ()
;;

let extraitMax t =
    let n = t.longueur in
    if n=0 then failwith "tas vide"
    else begin
      let maxi = t.donnees.(0) in
      t.donnees.(0) <- t.donnees.(n-1);
      t.longueur <- t.longueur -1;
      descendASaPlace t.donnees 0 (n-1);
      maxi
   end

;;
```

2.6 Tri par tas en place

On peut programmer grâce aux tas mutable une version du tri par tas « en place », c'est-à-dire où les éléments à trier ne sortiront pas du tableau initial.

Notons t le tableau à trier et n sa longueur.

2.6.1 Entasser

Pour que `t` devienne effectivement un tas, on peut imaginer deux stratégies :

- Parcourir le tableau de gauche à droite en appliquant `remonteASaPlace` à chaque case. On utilise alors un indice k qui augmente de 1 en 1, et l'invariant de boucle sera « `t.(0 :k)` est un tas » (ce qui nous autorisera à utiliser `remonteASaPlace t k`).
- Parcourir le tableau de droite à gauche en appliquant `descendASaPlace` à chaque case. On utilise un indice k qui diminue de 1 en 1, et l'invariant de groupe sera « Tous les sous-arbres enracinés en une case $\geqslant k$ sont des tas » (ce qui nous autorisera à utiliser `descendASaPlace t k`).

Après réflexion, la deuxième stratégie est à préférer. En effet, nous pourrons alors initialiser k au milieu du tableau, puisque tous les sous-arbres enracinés dans la deuxième moitié du tableau n'ont qu'un seul nœud et donc sont déjà des tas. Faisons le calcul précis :

$$\begin{aligned}
&\text{Le sous-arbre enraciné en } k \text{ est une feuille}\\
\Leftrightarrow &2k+1 \geqslant n\\
\Leftrightarrow &k \geqslant \tfrac{n-1}{2}\\
\Leftrightarrow &k \geqslant \left\lfloor \tfrac{n}{2} \right\rfloor
\end{aligned}$$

(Traiter les deux cas selon que k est pair ou impair pour la dernière équivalence.)

```
let entasse t =
  (* Entrée : tableau t
     Sortie : tas contenant les éléments de t
   *)
  let n= Array.length t in

  for k = n/2 -1 downto 0 do
    descendASaPlace t k n
  done;

  {donnees=t; longueur=n}
;;
```

2.6.2 Détasser

L'idée est qu'à chaque instant, nous aurons un entier l indiquant la taille du tas. Les l premières cases de t contiendront alors le tas lui-même, tandis que les $n-l$ cases suivantes contiendront les $n-l$ plus grands éléments de t, dans l'ordre.

```
let tri_par_tas t =
  (* Entrée : un tableau t
     Effet : à l'issue de cette procédure, t est trié
  *)
  let n = Array.length t in
  let t_tas = entasse t in
  (* NB : on a t=t_tas.donnees. Les modifications de l'un modifient aussi l'autre, ce sont le
   ↪  même tableau. *)
  for l = n downto 2 do
    (* En entrée de boucle : t.(0:l) est un tas, et t.(n:) contient les plus grands éléments de t,
     ↪  déjà triés. *)
    let x = extraitMax t_tas in
    t.(l-1) <- x
  done
;;
```

3 Interlude : gestion des bibliothèques

Vous commencez à avoir un certain nombre de fichiers .ml qui implémentent diverses structures de données vues en cours. Voici des commandes pour les charger simplement dans le toplevel de Caml :

- `#directory "repertoire";;` ajoute un répertoire à la liste des répertoires dans lesquels Caml va chercher les fichiers demandé.
- `#use "monFichier";;` Cette commande exécute le contenu du fichier externe comme si vous l'aviez tapé dans votre fichier principal.

Deuxième partie
Exercices

Exercices : arbres

1 Révisions sur les arbres

Par défaut, les arbres seront du type suivant : `type 'a arbre = Vide | Noeud of ('a arbre * 'a* 'a arbre)`.

Exercice 1. ** ! Arbre complet, équilibré, parfait

Attention : les définitions données dans cet exercice peuvent varier d'un sujet de concours à l'autre...
Nous dirons dans cet exercice qu'un arbre a de hauteur h est :

- *complet* lorsque tous ses « Vide » sont de profondeur h ou $h+1$;
- *complet gauche* lorsqu'il est complet et que toutes ses feuilles de hauteur h sont rangées « à gauche » de l'arbre.
- *parfait* lorsque tous ses Vide sont de profondeur $h+1$.
- *équilibré* lorsque pour chaque nœud n, la hauteur des deux fils de n diffère d'autre plus 1.

1. Quels sont les liens logiques entre ces trois notions ? Dessiner un arbre équilibré mais pas complet.
2. Écrire une fonction qui teste si un arbre est complet.
3. Écrire une fonction qui teste si un arbre est complet gauche.
4. Écrire une fonction qui teste si un arbre est parfait.
5. Écrire une fonction qui teste si un arbre est équilibré en un seul parcours.
6. (CCP) On définit le type `catégorieArbre` ainsi :
   `type categorieArbre = Parfait | Complet | CompletGauche | Quelconque`.
   Écrire une fonction qui prend un arbre et renvoie sa catégorie. Lorsque deux possibilités sont vraies, on renverra la plus précise (par exemple si un arbre est parfait, ne pas se contenter de dire qu'il est complet).

Exercice 2. ** ! Caractérisation d'un arbre parfait

1. Montrer par récurrence qu'un arbre binaire de hauteur h a au plus $2^{h+1}-1$ nœuds.
2. Soit a un arbre binaire de hauteur h contenant n nœuds. Montrer qu'il est parfait si et seulement si $n = 2^{h+1}-1$. La définition de parfait étant ici celle de l'exercice 1.

Exercice 3. * ! Reconstruire un arbre à partir de la liste de ses nœuds**

Le but est de reconstruire un arbre à partir de la liste de ses étiquettes et de ses feuilles vides selon un certain parcours.

On définit le type suivant : `type 'a morceauDArbre = V | E of 'a`. La liste qu'on prendra en entrée sera de type `'a morceauDArbre list`. Le constructeur `V` correspond à une feuille vide, et `E n` correspond à un nœud intérieur avec n comme étiquette.

1. Écrire la fonction correspondante dans le cas d'un parcours en profondeur postfixé.
2. Écrire la fonction correspondante dans le cas d'un parcours en largeur.

Exercice 4. ** Numérotation

On manipulera ici des arbres binaires d'entiers.

1. Écrire une fonction qui numérote les étiquettes d'un arbre, en faisant en sorte de ne jamais mettre deux fois la même étiquette à deux nœuds différents. On prendra en entrée un arbre binaire et on renverra l'arbre ayant le même squelette mais tel que pour tout nœud n, l'étiquette de n sera son numéro.
2. Même question mais avec une contrainte supplémentaire : la numérotation doit être effectuée selon un parcours en largeur.

1.1 Arbres binaires de recherche

Exercice 5. * ! Union et intersection**

Pour utiliser des arbres binaires de recherche pour réaliser une structure d'ensemble, il serait naturel de disposer de fonctions pour calculer des unions et des intersections.

Dans ce contexte, nous supposons que les arbres ne contiennent pas d'élément en double, et nous voulons maintenir cette propriété.

1. Écrire une fonction `segmente` prenant en entrée un arbre binaire de recherche a et un élément x et renvoyant le triplet (un ABR contenant les éléments de $a < x$, un ABR contenant les éléments $> x$, le booléen $x \in a$).

2. Écrire une fonction `union`.
3. Écrire une fonction `union_abr_disjoints` qui renvoie la réunion de deux ABR a et b tels que toutes les étiquettes de a sont strictement inférieures aux étiquettes de b.
4. Programmer alors une fonction `intersection`.
5. Et enfin une fonction `différence`.

Exercice 6. **! Créer un ABR à partir d'une liste triée

(Vu en MPSI normalement)

1. Pourquoi la fonction `abr_of_list` vue en première année est-elle à proscrire pour créer un arbre binaire de recherche à partir d'une liste triée ?
2. Écrire une fonction prenant en entrée une liste triée et renvoyant un arbre binaire de recherche équilibré contenant les éléments de cette liste.
3. Quelle est la complexité de cette fonction ?

Exercice 7. ** 5/2 Vérification d'orthographe

1. Le fichier `liste.de.mots.francais.txt` contient la liste des mots du français. Écrire une fonction permettant de les charger dans un ABR.
   Commandes utiles :
   - Pour ouvrir un fichier : `let fichierEntree = open_in "nom du fichier";`
   - Pour lire une ligne : `input_line fichierEntree`.

   Dans Caml, pour lire la totalité des lignes d'un fichier, on est obligé de faire des `input_line` jusqu'à obtenir l'erreur `End_of_file` (utiliser un `try ... with`).
2. En déduire une fonction prenant en entrée une chaîne de caractère et indiquant s'il s'agit d'un mot français.
3. *bonus :* prendre en entrée un texte, le découper en mot selon les espaces, et tester si chaque mot est français. On pourra par exemple renvoyer la liste des mots mal orthographiés.

2 Tas

Exercice 8. **! Vérifier qu'un arbre est un tas

Écrire une fonction pour vérifier si un arbre est un tas :

1. pour un arbre persistant ;
2. pour un arbre modifiable enregistré dans un vecteur.

Exercice 9. **! Conversion entre tableau et arbre binaire

Écrire une fonction prenant en entrée un arbre représenté par un vecteur et renvoyant l'arbre correspondant enregistré de la manière habituelle.

Exercice 10. **! Créer un tas

1. *Fusion avec colle :* Soient t_1 et t_2 deux tas et x un élément du même type que ceux dans t_1 et t_2. Le tas obtenu en « fusionnant t_1 et t_2 en utilisant x comme colle » et le tas obtenu à partir de `noeud(t1, x, t2)` en faisant descendre x jusqu'à une place adéquate, comme vu en cours sur la version impérative des tas.
   Programmer la fonction `fusionAvecColle` correspondante.
2. *Transformer un arbre binaire quelconque en un tas :* Écrire une fonction prenant en entrée un arbre binaire, et renvoyant un tas contenant les mêmes éléments.
3. Une méthode « diviser pour régner » pour créer un tas à partir d'une liste :
   (a) Découper la liste en un élément et deux listes de longueur égales à au plus un près ;
   (b) Transformer les deux listes en tas ;
   (c) Rassembler le tout à l'aide de `fusionAvecColle`.

   Donner la complexité de cette méthode. (On ne demande pas de programmer cette méthode.)
4. *(***) Créer un tas à partir d'une liste en temps linéaire (construction de Floyd) :* On désire créer un tas à partir d'une liste l. On suppose pour simplifier qu'il existe $p \in \mathbb{N}$ tel que la longueur de l soit $2^p - 1$. On découpe alors la liste en deux listes `lTas` de longueur 2^{p-1} et `lColle` de longueur $2^{p-1} - 1$.
   On convertit les éléments de `lTas` en tas de un seul élément. On va ensuite les fusionner deux par deux en utilisant les éléments de `lColle` comme colle, jusqu'à obtenir une liste de un seul tas.
   Programmer cette méthode.

5. Calculer la complexité de cette dernière fonction.

Exercice 11. ** Application : k-fusion

(En anglais « K-way merge ».)

Dans l'algorithme du tri fusion, on fusionne deux listes triées. Dans certaines circonstances, on peut avoir besoin d'en fusionner un plus grand nombre. On fixe $k \in \mathbb{N}^*$, et on va étudier une méthode pour fusionner k listes triées.

1. Donner, en fonction de k et de la longueur maximale des k listes à fusionner, la complexité de l'algorithme naïf consistant à rechercher à chaque étape le plus petit élément parmi les k têtes de listes pour le placer dans le résultat.
2. On propose de maintenir un tas-min pour contenir les k têtes de liste. Quelle sera alors la complexité ?
3. Programmer cette méthode.

Exercice 12. * Construction de tas complet gauche (CCP 2015)**

Il est possible de programmer l'insertion et l'extraction dans un tas persistant implémenté dans un type d'arbre binaire classique en faisant en sorte que le tas soit toujours complet gauche.

Les fonctions prendront en argument supplémentaire la taille du tas. Dès lors, la décomposition en base deux de cet entier indique le chemin à suivre dans le tas pour arriver à la dernière feuille...

Quelques indications

1 1.
2. Écrire une fonction qui renvoie les profondeurs minimale et maximale d'un arbre.
3. Utiliser un parcours en largeur.
4. Utiliser une fonction auxiliaire qui renvoi une valeur supplémentaire : la hauteur de l'arbre.
5. Même indication que pour la question précédente !

3 1. Le principe est le même que pour évaluer une expression algébrique postfixée (c'est-à-dire en notation polonaise).
2. Pareil, mais avec une file d'attente. Il faudra au préalable retourner la liste issue du parcours en largeur.

4 2) s'inspirer de l'exercice 3.

5 Faire des dessins !

6 Commencer par écrire une fonction qui découpe une liste en deux par le milieu. (Au préalable, déterminer le rôle précis de cette fonction.)

12 Dessiner un arbre complet gauche, et écrire pour chaque nœud son numéro selon un parcours en largeur en base 2.

Quelques solutions

1

3

4

5

```
let rec segmente x = function
  (* Renvoie un triplet (abr des éléments <x, abr des éléments >x, bool x est dans a) *)
  | Vide -> Vide, Vide, false
  | Noeud(fg, e, fd) when x=e ->
    fg, fd, true
  | Noeud(fg, e, fd) when x<e ->
    let sg, sd, b = segmente x fg in
    sg, Noeud(sd, e, fd), b
  | Noeud(fg, e, fd)  -> (* cas où x >e *)
    let sg, sd, b = segmente x fd in
    Noeud(fg, e, sg), sd, b
;;


let rec union a b =
    match a, b with
    |Vide, _  -> b
    |_, Vide  -> a

    |Noeud(ag, e, ad), Noeud(bg, f, bd)  when e<f  ->
      let adg, add, _ = segmente f ad  and bgg, bgd, _ = segmente e bg in
         Noeud( union ag bgg,
                e,
                Noeud( union adg bgd,
                       f,
                       union add bd
    )
   )

    |Noeud(ag, e, ad), Noeud(bg, f, bd)  when e=f  ->
        Noeud( union ag bg, e, union ad bd)

    |_   -> union b a (* L'astuce du fainéant *)
;;
```

```
let rec extrait_min = function
  (* renvoie le couple (min de 'labr, 'labr privé de ce min) *)
  |Vide -> failwith "arbre vide"
  |Noeud(Vide, e, fd) -> (e, fd)
  |Noeud (fg, e, fd) ->
      let mini, fg_sans_mini = extrait_min fg in
      (mini, Noeud(fg_sans_mini, e, fd))
;;

let rec fusion_ABR_disjoints a b =
  match a, b with
  |_, Vide  -> a
  |_ -> let e, fd_sans_e = extrait_min b in
      Noeud(b, e, fd_sans_e )
;;



let rec inter a b=
    match a, b with
```

```
    |Vide, _  -> Vide
    |_, Vide  -> Vide
    |Noeud(ag, e, ad), _    ->
      let sbg, sbd, e_dans_b = segmente e b in
      if e_dans_b then
  Noeud( inter ag sbg, e, inter ad sbd)
      else
  fusion_ABR_disjoints (inter ag sbg) (inter ad sbd)
;;
```

6

```
let rec decoupe l n=
    (* Renvoie le triplet (liste des n premiers éléments de l, le n+1ème élément, suite de l)
       ↪  *)
    if n=0 then [], hd l, tl l
    else
        let deb, mediane, fin = decoupe (tl l) (n-1) in
            hd l ::deb, mediane, fin
;;

let rec abr_of_list_triee_aux l n=
    (* n est la longueur de l. Cette fonction auxiliaire permet d'éviter de recalculer n à
       ↪ chaque appel*)
    match n with
        |0  -> Vide
        |1  -> feuille (hd l)
        |_  -> let deb, mediane, fin = decoupe l (n/2) in
                Noeud( abr_of_list_triee_aux deb (n/2), mediane, abr_of_list_triee_aux fin
                    ↪ (n-n/2 - 1) )
;;

let abr_of_list_triee l=
    abr_of_list_triee_aux l (list_length l)
;;
(*
abr_of_list_triee [1;2;3;4;5;5;7];;*)
```

Voyons la complexité de cette fonction :

- `decoupe` en une complexité en $O(n)$, si n est la longueur de la liste à découper.
- Si nous notons, pour tout $n \in \mathbb{N}$, C_n la complexité de `abr_of_list_triee_aux` pour une liste de longueur n, nous avons :

$$\begin{cases} C_0 = 0 \\ C_1 = 1 \\ \forall n \in [\![2, \infty[\![,\ C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lfloor \frac{n+1}{2} \rfloor} + O(n) \end{cases}.$$

D'où $C_n = \Theta_{n\to\infty}\left(n\log(n)\right)$. (Même relation de récurrence que pour le tri fusion, ou directement le master théorème.)

7

8

9

10

1.
2.
3. Je note pour tout $n \in \mathbb{N}$ C_n le nombre maximal de comparaison pour entasser une liste de au plus n éléments. La suite C est croissante à cause du « au plus » dans la définition. On a pour tout $n \in \mathbb{N}^*$, $C_n = C_{\lfloor \frac{n}{2} \rfloor} + C_{\lfloor \frac{n-1}{2} \rfloor} + O(n) \leqslant 2C_{\lfloor \frac{n}{2} \rfloor} + O(n)$.
   Avec les notations du cours, $a+b=2$, $\alpha = 1$ et $\beta = 1$. Le master théorème donne une complexité en $O(n\log n)$.
4. La complexité de `fusionAvecColle` est en $O(h)$ si h est le maximum des hauteurs des arbres fusionnés.
   Soit $p \in \mathbb{N}^*$ et l une liste de longueur $2^p - 1$. On va obtenir comme tas un arbre parfait de hauteur $p-1$.
   Pour tout $i \in [\![0, p-1]\!]$, la création de l'étage i nécessite 2^i fusions d'arbres de hauteur $p-2-i$ (l'arbre vide étant considéré ici de hauteur -1).

La complexité totale est donc

$$\begin{aligned}\sum_{i=0}^{p-1} 2^i O\left((p-2-i)\right) &= \sum_{i=0}^{p-1} O\left(2^i(p-i)\right) \\ &= O\left(\sum_{i=0}^{p-1} 2^i(p-i)\right) \\ &= O\left(p(2^p-1) - \sum_{i=1}^{p-1} i2^i\right)\end{aligned}$$

Série divergente à termes positifs

Pour la deuxième somme, on peut utiliser la fonction $f : x \mapsto \sum_{i=0}^{p-1} x^i$ qui vaut aussi $x \mapsto \frac{1-x^p}{1-x}$ et dont la dérivée est $x \mapsto \sum_{i=1}^{p-1} ix^{i-1}$ qui vaut donc aussi $x \mapsto \frac{(p-1)x^p - px^{p-1}+1}{(1-x)^2}$.

En particulier, évaluant en 2, il vient $\sum_{i=1}^{p-1} i2^{i-1} = (p-1)2^p - p2^{p-1} + 1$. Revenons au calcul de la complexité :

$$\begin{aligned}O\left(p(2^p-1) - \sum_{i=1}^{p-1} i2^i\right) &= O\left(p(2^p-1) - (p-1)2^{p+1} + p2^p - 2\right) \\ &= O\left(2^{p+1} - p - 2\right) \\ &= O\left(2^p\right) \\ &= O\left(|l|\right)\end{aligned}$$

12