Arbres

Table des matières

Première partie

Cours

1 Création de type énuméré en Caml

On peut définir un type en énumérant simplement les valeurs possibles :

```
type jour = Lundi | Mardi | Mercredi | Jeudi | Vendredi | Samedi | Dimanche ;;
```

Penser que le « | » signifie un « ou ».

L'utilisation d'un tel type se fait par filtrage. Par exemple :

```
let demain j =
    match j with
        | Lundi -> Mardi
        | Mardi -> Mercredi
        etc...
;;
```

Mais chacune des valeurs de base peut aussi être d'un certain type :

```
type Rbarre = Infini | Minfini | Fini of float;;
```

Exemple de fonction :

```
let addition x y=
    match (x,y) with
        | Infini, Minfini -> failwith "nan"
        | Minfini, Infini -> failwith "nan"
        | Infini,_ -> Infini
        | _ ,Infini-> Infini
        | Minfini,_ -> Minfini
        | _ ,Minfini-> Minfini
        | Fini(a), Fini(b) -> Fini (a+.b)
;;
```

Remarque : « nan » signifie « not a number », c'est ce qu'on obtient dans la plupart des langages lorsqu'on effectue une opération qui correspond à une forme indéterminée.

Exemple important : Reconnaissez-vous le type suivant ?

```
type a' mystere = Vide | Cons of 'a * mystere;;
```

N.B. C'est un type récursif !

Vocabulaire : les éléments de base dans un type somme (dans les exemples précédents il y avait `Lundi, Infini, Fini, Vide` s'appellent les « constructeurs » du type. Ils peuvent être constants ou être des fonctions. En Ocaml, les constructeurs prennent une majuscule.
Important : Les constructeurs sont les seules fonctions qui peuvent être utilisés dans un motif (un objet d'un type construit peut être « déconstruit »).

cf exercice : 2

2 Vocabulaire des arbres

Un graphe est un ensemble de *sommets* reliés par des *arêtes*.

Un arbre est un graphe connexe, ce qui signifie qu'entre deux points existe toujours un chemin, et acyclique, ce qui signifie qu'il n'existe pas de chemin dans l'arbre qui revienne à son point de départ (pas de boucle). De manière équivalente, un arbre est un graphe tel que deux points sont toujours reliés par un unique chemin.

Dans un arbre, les sommets sont généralement appelés des *nœud*. Un nœud qui est relié à un seul autre nœud est souvent appelé une *feuille*.

En informatique, on choisit en général un sommet qu'on appelle la *racine*. Lorsqu'on dessine un arbre, on place la racine en haut, puis une ligne en dessous tous les sommets reliés à la racine, la ligne suivante on dessine les sommets reliés à ces sommets, etc... Les feuilles sont donc dessinées en bas. Un chemin de la racine vers une branche s'appelle une branche.

Remarques :

- On peut préférer appeler « arbre déraciné » un arbre pour lequel on n'a pas fixé de racine. En informatique, nous n'en utiliserons presque jamais.
- Un graphe non connexe, mais sans sans cycle est donc une réunion de plusieurs arbres. On dit parfois qu'il s'agit d'une « forêt ».

La *hauteur* d'un arbre est le nombre maximal d'arêtes entre une feuille et la racine.

Remarque : Certains sujets de concours préfèrent compter le nombre maximal de nœuds entre la racine et une feuille. C'est alors un de plus.

Les sous-arbres situés en dessous d'un nœud sont appelés ses *fils*.

En informatique, on utilise des arbres pour stocker des informations. Selon la situation, on peut attacher une donnée à chaque nœud, ou à chaque feuille. La donnée enregistrée dans un nœud s'appelle son *étiquette*.

Par exemple, soit A un arbre. Supposons que chaque nœud ait soit deux soit aucun fils (c'est-à-dire un nœud qui n'est pas une feuille a deux fils). Soit h la hauteur de A. Alors le nombre de feuilles est au maximum 2^h. Et le nombre total de nœuds est au maximum $2^{h+1} - 1$. Ce maximum est atteint lorsque toutes les branches sont de longueur h.

Remarque : Un tel arbre s'appelle arbre *binaire*, ou parfois binaire entier, si on veut insister sur le fait qu'un nœud ne peut avoir un seul fils.

Ainsi un arbre binaire de hauteur h permet de stocker $2^{h+1} - 1$ données. D'un autre point de vue, pour enregistrer N données, il faudra un arbre dont la hauteur sera de l'ordre de $\log_2(N)$.

3 Type Caml

En pratique pour enregistrer un arbre, le principe est exactement le même que pour les listes chaînées, la seule différence est que pour chaque valeur, on enregistre plusieurs pointeurs : un vers chaque fils.

La manipulation d'arbre sera donc très proche de la manipulation de liste, en particulier adaptée à la programmation récursive.

Signalons enfin que le programme de MPSI limite l'étude des arbres *persistants*.

3.1 Définition du type

En Caml on peut procéder ainsi pour définir un arbre : un arbre peut être

- vide ;
- ou un nœud pouvant avoir plusieurs fils.

Selon l'implémentation choisie, une feuille sera un nœud qui n'a aucun fils, ou un nœud qui n'a que des fils vides.

La manière de créer le type dépend des étiquettes qu'on veut mettre à chaque nœud et du nombre de fils de chaque nœud. Quelques exemples :

- Arbre binaire, où tous les nœuds ont une étiquette :

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

Remarque : Ce sera le type le plus fréquemment utilisé dans les exemples à suivre.

- Arbre binaire, où seules les feuilles ont une étiquette, contenant un élément de type `'a` :

```
type 'a arbref = Feuille of 'a  | Noeud of ('a arbref * 'a arbref);;
```

Remarque : Ce type ne permet pas de créer un arbre vide, ce qui peut poser problème dans certaines situations.

- Arbre binaire où les arêtes sont étiquetées :

```
type 'a arbrea = Feuille | Noeud of ('a * 'a arbrea * 'a * 'a arbrea
```

Dans l'expression `Noeud(e1,f1, e2,f2)`, on pense que `e1` est l'étiquette de l'arête menant à `f1`.

- Arbres ternaires :

```
type 'a arbre3 = Vide  | Noeud of ( 'a * 'a arbre3 * 'a arbre3 * 'a arbre3 );;
```

- Arbres où chaque nœud peut avoir un nombre quelconque de fils, et contient une étiquette :

```
type 'a arbreg = Noeud of 'a * ('a arbreg) list;;
```

Il n'est pas besoin de prévoir un cas de base (arbre vide ou feuille) car le type `list` en contient un. En pratique, une feuille est représentée par `Noeud (x, [])`, où `x` est l'étiquette de la feuille. Ce type ne permet pas d'enregistrer un arbre vide : si besoin rajouter un constructeur `Vide` exprès.

3.2 Manipulation

Les types d'arbres que nous avons présentés sont tous des types somme (c'est-à-dire « construits »). On va donc les manipuler par filtrage (c'est-à-dire en les « déconstruisant »).

Rappel : un motif de filtrage peut être construit avec, et uniquement avec :

- des constructeurs ;
- des identificateurs libres (c'est-à-dire non utilisées jusque là).

Dans les exemples précédents, les constructeurs étaient `Noeud, Vide, Feuille`.

Remarque : Dans les filtrages utilisés depuis le début de l'année, on avait utilisé principalement les constructeurs des listes `::` et `[]`, mais aussi la virgule qu'on peut voir comme le constructeur des couples (ou n-uplets plus généralement).

Traitons quelques exemples :

- Calculer le nombre de nœuds d'un arbre ;
- Calculer le nombre de feuilles d'un arbre ;
- Calculer la hauteur d'un arbre ;
- Calculer la somme des étiquettes d'un arbre ;
- Tester si un élément est un étiquette d'un arbre.

4 Arbre binaire de recherche

Dans cette partie, nous considérons des arbres binaires définis par le type

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

On suppose que le type `'a` est muni d'une relation d'ordre.

Remarque : La notion de relation d'ordre est très utile en informatique. Le fait de pouvoir utiliser un ABR est un exemple très parlant. En conséquence, presque tous les types sont munis de relation d'ordre, notamment grâce à l'ordre lexicographique.

On notera $\mathcal{B}$ l'ensemble des arbres de ce type.

4.1 Description

Définition 4.1. *Soit $a \in \mathcal{B}$.*

On dit que a est un arbre binaire de recherche (ABR) lorsque pour tout nœud n de a,

les étiquettes du fils gauche de n sont inférieures à l'étiquette de n elle même inférieure aux étiquettes de son fils droit.

4.2 Programmation

La fonction la plus importante est celle permettant de tester si un élément est présent dans un ABR.

```
let rec appartient x a=
  (* entrée : un ABR a
              un élément x
     sortie : le booléen « x est une étiquette de a » *)
  match a with
  | Vide -> false
  | Noeud(fg, e, fd) when e=x -> true
  | Noeud(fg, e, fd) when e < x ->  (*Poursuivre la recherche à droite *)
     appartient x fd
  | Noeud(fg, e, fd) -> (* e > x : poursuivre la recherche à gauche *)
     appartient x fg
;;
```

Programmons aussi une fonction permettant d'ajouter un élément dans un ABR.

```
let rec insertion x a =
    (* Entree : un ABR a
        Sortie : un ABR contenant les elements de a ainsi que x. Si x y etait deja, on y met
            ↪ un nouvel exemplaire*)
    match a with
    |Vide -> Noeud(Vide,x,Vide)
    |Noeud(fg,e,fd) when x>=e -> Noeud(fg, e, insertion x fd)
    |Noeud(fg,e,fd) -> Noeud(insertion x fg, e, fd)
;;
```

On déduit immédiatement, une fonction pour ranger le contenu d'une liste dans un ABR :

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

Version fonctionnelle en une ligne :

```
let abr_of_list l=
  List.fold_right insertion l Vide ;;
```

Remarque : Pour la différente entre `fold_left` et `fold_right` :

- `List.fold_right op [a0…; ;an] e` renvoie $a_0 op ....(a_{n-2} op(a_{n-1} op(a_n ope)))$
- Alors que :
  `List.fold_left op e [a0…; ;an]` renvoie $(((e opa_0) opa_1 *) * a_2)....$
  Noter que l'ordre des arguments n'est pas le même...

Pour les autres fonctions élémentaires, voir l'exercice 15.

4.3 Complexité

On constate que la complexité de cette fonction est en $O(h_a)$. Par conséquent, on aura intérêt à ce que la hauteur de nos arbres soit la plus faible possible.

Définition 4.2. *On dit que a est équilibré lorsque pour tout nœud n, en notant g le fils gauche et d le fils droit de n, on a $|h_g - h_d| \leq 1$ (les hauteurs des fils gauche et droit diffèrent au plus de 1).*

Remarque : Un arbre « équilibré » est un intermédiaire entre le peigne et l'arbre complet (ou parfait). Il n'est pas optimal en terme de hauteur, mais plus facile à obtenir.

Proposition 4.3. *Pour tout arbre a équilibré, $h_a \leq \frac{3}{2}\log_2(N_a + 1)$.*

Démonstration : Pour tout $h \in \mathbb{N}$, posons $P(h)$: « Pour tout arbre a de hauteur h, $h \leqslant \frac{3}{2}\log_2(N_a + 1)$. ».
Et remarquons que l'inégalité $h_a \leq \frac{3}{2}\log_2(N_a + 1)$ équivaut à $N_a \geqslant 2^{\frac{2}{3}h_a}$.

- *Initialisation* : Soit a un arbre de hauteur 0. Il est donc réduit à sa racine et $N_a = 1$. Ainsi $\frac{3}{2}\log_2(N_a+1) = 0$ et la formule proposée est vraie.
- *Hérédité* :Soit $h \in N$ tel que $\forall k \in [\![0,h]\!]$, $P(k)$. Soit a un arbre de hauteur $h+1$. Soient f_g et f_d les fils de la racine. Du fait que a est équilibré, f_g et f_d sont de hauteur h ou $h-1$. De plus l'un d'eux est de hauteur h.
  Ainsi,

$$\begin{aligned} N_a =& 1 + N_{f_g} + N_{f_g} \\ \geqslant& 2^{\frac{2}{3}h} + 2^{\frac{2}{3}(h-1)} - 1 \end{aligned}$$

  La formule souhaité sera alors vraie si (et pas « seulement si » !) $2^{\frac{2}{3}h} + 2^{\frac{2}{3}(h-1)} - 1 \geqslant 2^{\frac{2}{3}(h+1)} - 1$. Notons $(*)$ cette inégalité, et procédons par équivalence :

$$\begin{aligned} (*) \Leftrightarrow& 2^{\frac{2}{3}h} + 2^{\frac{2}{3}(h-1)} \geqslant 2^{\frac{2}{3}(h+1)} \\ \Leftrightarrow& 1 + 2^{-\frac{2}{3}} \geqslant 2^{\frac{2}{3}} \end{aligned}$$

  On vérifie par un calcul approché dans une calculette ou dans Caml que cette dernière inégalité est vraie. Donc $(*)$ l'est aussi. Donc $P(h+1)$.

□

bonus : Calculer la constante optimale pouvant remplacer le $\frac{3}{2}$ dans la formule ci-dessus.

Ainsi, lorsque l'arbre est équilibré, la complexité de la recherche d'un élément est logarithmique. C'est un sujet d'étude classique que de voir comment créer et maintenir des arbres équilibrés.

On peut considérer que l'utilisation d'un arbre de recherche est l'équivalent récursif de l'utilisation du tri dichotomique dans un vecteur trié. Comparons les avantages et inconvénients des deux :

- L'arbre est plus facile à manipuler, moins de risque d'erreur ;
- On peut rajouter un élément dans un ABR en $O(\log h)$ contre $O(n)$ pour un tableau trié (programmer l'insertion si pas déjà fait) ;
- Lorsque l'arbre est parfait, la complexité de la recherche est la même que dans un tableau. Lorsqu'il est seulement équilibré, il y a un facteur multiplicatif. Lorsqu'on n'a pas de certitude sur son squelette, on ne peut pas savoir.
- Il n'est pas évident de maintenir le caractère équilibré d'un ABR (il existe plusieurs méthodes, qui ne sont pas au programme : arbres bicolores, arbres AVL, B-arbres (pas binaires, qui combinent en fait tableaux triés et ABR)...)

cf exercice : 15.

4.4 Supprimer un élément

Voici un exemple de fonction un peu moins évidente.

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 17.

4.5 Dictionnaires

Les arbres binaires de recherche sont un bon moyen de créer des dictionnaires persistants. Il suffit d'enregistrer à chaque nœud un couple (clef, valeur).

On peut utiliser le type suivant :

```
type ('a, 'b) dico = Vide | Noeud of (('a, 'b) dico * 'a * 'b * ('a, 'b) dico);;
```

Les fonctions `est_une_clef` et `assoc` sont de simple variantes de la fonction de recherche dans un ABR classique. Pour la fonction d'insertion, il y a un choix à faire : si une clef est déjà présente, on peut supprimer l'ancienne valeur, ou bien la masquer, c'est-à-dire la descendre plus bas dans l'arbre, auquel cas l faut s'assurer que la fonction `assoc` s'arrête à la première occurrence rencontrée. Ceci présente l'avantage de permettre de supprimer une association en faisant alors « reapparaître » l'ancienne, comme dans une liste d'association (il faudra alors être soigneux sur la programmation de la suppression...).

5 Arbres de valence non bornée

Voyons maintenant comment utiliser des arbres dont chaque nœud peut avoir un nombre quelconque de fils.

On enregistrera alors pour chaque nœud la *liste* de ses fils. Dans le type suivant, on décide en outre de munir chaque nœud d'une étiquette. Cependant on peut aussi munir chaque arête d'une étiquette : dans ce cas on enregistre pour chaque nœud la liste des couples (fils, étiquette de l'arête y menant).

```
1 type 'a arbre3 =   Noeud of 'a * ('a arbre3 list);;
```

N.B. Ici une feuille est un nœud dont la liste des fils est vide. Ce type ne permet pas un arbre vide.

On rappelle qu'une liste d'arbres est parfois appelée une «forêt». Ainsi, chaque nœud est muni d'une étiquette et d'une forêt : la liste de ses fils.

Pour parcourir un tel arbre, on écrit en général deux fonctions mutuellement récursives :

1. Une fonction conçue pour balayer un arbre ;
2. et une fonction conçue pour balayer une forêt.

Il est souvent possible de s'épargner la seconde fonction, à l'aide de programmation fonctionnelle (Par exemple, on pourra mapper la première fonction à la liste des fils).

cf exercice : 10

6 Induction structurelle

6.1 Première approche de la définition inductive

Redonnons la définition suivante de l'ensemble $\mathbb{N}$ en Caml :

```
type entier_nat = Zero | Succ of entier_nat;;
```

La définition inductive d'un ensemble consiste à donner :

- des éléments de base de cet ensemble ;
- des règles pour créer de nouveaux éléments.

L'ensemble ainsi défini est alors l'ensemble de tous les éléments pouvant être obtenus à partir d'un nombre fini d'éléments de base et d'application des règles.

On peut prouver que c'est aussi le plus petit (pour $\subset$) ensemble contenant les éléments de base et stable par les règles indiquées.

On reviendra un peu plus précisément sur ce concept dans le chapitre sur la logique.

6.2 Définition récursive des arbres

Une définition récursive de l'ensemble des arbres non vides pourrait alors être :

- Une feuille est un arbre ;
- un nœuds associé à un ensemble d'arbres est un arbre.

Remarque : Le point gênant est que nous n'avons pas vraiment défini la notion de feuille ni de nœud ici. L'astuce consiste en général à définir choisir un ensemble de symboles Σ (ici on pourrait prendre, si on ne met pas d'étiquettes à nos arbres, $\Sigma = \{N, F, (,)\}$ ainsi que la virgule), puis à définir nos éléments comme étant des suites de symboles. La définition deviendrait :

- le symbole F est un arbre,
- pour tout $n \in \mathbb{N}$ et tous arbres $a_1, ..., a_n$, la suite de symboles $N(a_1, ..., a_n)$ est un arbre.

Ainsi un arbre est juste un empilement des symboles N, F, de parenthèses et de virgules selon certaines règles.

6.3 Deuxième approche de la définition par induction structurelle

Si on veut formaliser un peu : soit X un ensemble (en général on fixe un ensemble de symbole Σ, et prend pour X l'ensemble des suites finies de symboles. Noté Σ^* d'ailleurs.). On appelle « constructeur » une fonction C telle qu'il existe $n \in \mathbb{N}$ tel que $C : X^n \to X$. L'entier n s'appelle alors l'arité de C. Soit $\mathcal{B} \in \mathcal{P}(X)$. Soit $\mathcal{C}$ un ensemble de constructeurs.

L'ensemble E définit par induction à l'aide des cas de base $\mathcal{B}$ et des constructeurs $\mathcal{C}$ est le plus petit ensemble tel que :

- $\forall x \in \mathcal{B}$, $x \in E$;
- $\forall C \in \mathcal{C}$, en notant n l'arité de C, pour tout $(x_1, ..., x_n) \in E^n$, $C(x_1, ..., x_n) \in E$.

6.4 Rappel : Le théorème de la récurrence sur $\mathbb{N}$

6.5 Le théorème de l'induction structurelle

Le théorème de la récurrence se généralise immédiatement à tout ensemble défini récursivement :

Théorème 6.1. *Soit E un ensemble défini par induction. Soit $\mathcal{B}$ l'ensemble des éléments de base, et c l'ensemble des constructeurs correspondant. Soit P un prédicat sur E. On suppose :*

- $\forall x \in \mathcal{B}$, $P(x)$;
- $\forall C \in \mathcal{C}$, *d'arité* n, $\forall (x_1, ..., x_n) \in E^n$, $\bigwedge i = 1^n P(x_i) \Rightarrow P\big(C(x_1, ..., x_n)\big)$.

Alors $\forall \in E$, $P(x)$.

7 Arbres binaires

Revenons un peu plus précisément sur les arbres binaires.

7.1 Vocabulaire

Voici le vocabulaire précis employé par le programme :

- Arbre binaire strict : chaque nœud a zéro ou deux fils. Autrement dit es nœuds internes ont deux fils. Type Caml :

```
type 'a arbreBinStrict = Feuille of 'a | Noeud of ('a arbreBinStrict * 'a * 'a
    ↪ arbreBinStrict);;
```

(On ne prévoit pas un cas pour l'arbre vide ici.)

- Arbre binaire : chaque nœud à zéro, un ou deux fils. Type Caml

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

7.2 Nombre de nœuds maximal et minimal

Soit a un arbre binaire, donc chacun de ses nœuds a zéro ou un fils. Soit h sa hauteur. Pour tout $p \in [\![0, h]\!]$, il y a au maximum 2^p et au minimum un nœud(s) de profondeur p.

En sommant pour tous les p dans $[\![0, h]\!]$, cela fait au minimum $h+1$ et au maximum $2^{h+1}-1$ nœuds dans a.

Le minimum est atteint lorsque chaque nœud a un seul fils. On dit alors que a est « un peigne ». Cette image s'explique dans les cas où le fils non vide est toujours du même côté et si on représente aussi les nœuds vides.

Le maximum est atteint lorsque chaque nœud, sauf ceux de profondeur p a deux fils. On dit dans ce cas que a est « complet » ou « parfait » (selon les auteurs...).

D'un autre point de vue : soit a un arbre binaire, h sa hauteur, et n son nombre de nœuds. On vient de voir que

$$h+1 \leqslant n \leqslant 2^{h+1}-1$$

d'où on déduit :

$$\log_2(n+1) - 1 \leqslant h \leqslant n-1$$

Ainsi la hauteur maximale d'un arbre à n nœuds est $n-1$ et sa hauteur minimale est $\log_2(n+1)-1$ (retenir que c'est de l'ordre de $\log(n)$).

Démontrons tout ceci en détail par induction structurelle.

7.3 Nombre de nœuds et de feuilles

Pour tout arbre a, notons $NI(a)$ sont nombre de nœuds internes (c'est-à-dire pas des feuilles) et $NF(a)$ son nombre de feuilles.

Proposition 7.1. *Pour tout arbre binaire strict non vide a, $NF(a) = NI(a) + 1$.*

Remarque : Le nombre total de nœuds est donc $N = NI + NF = 2NI + 1$.

Démonstration :
Notons ici $\mathcal{A}$ l'ensemble des arbres binaires strictes.
Pour tout $a \in \mathcal{A}$, soit $P(a)$: « $NF(a) = NI(a) + 1$.»

- **Initialisation :** Soit a une feuille. Donc $NI(a) = 0$ et $NF(a) = 1$. La formule fonctionne. Donc $P(a)$.
- **Hérédité :** Soit $(f_g, f_d) \in \mathcal{A}^2$ tel que $P(f_g)$ et $P(f_d)$. Donc $NF(f_g) = NI(f_g) + 1$ et $NF(f_d) = NI(f_d) + 1$. Soit `a = Noeud(fg, fd)`.
  Comptons alors le nombre de nœuds internes de a : $NI(a) = NI(f_g) + NI(f_d) + 1$ (le $+1$ c'est la racine).
  Et le nombre de feuilles : $NF(a) = NF(f_g) + NF(f_d)$.
  On rassemble le tout :

$$\begin{aligned} NF(a) &= NF(f_g) + NF(f_d) \\ &= NI(f_g) + 1 + NI(f_d) + 1 \\ &= NI(a) + 1 \end{aligned}$$

  D'où $P(a)$.

En conclusion, pour tout $a \in \mathcal{A}$, $P(a)$. □

7.4 Une application : complexité maximale d'un tri

Fixons $n \in \mathbb{N}^*$. Le but est d'estimer la complexité minimale pour trier une suite de n éléments en les comparant deux à deux.

Soit $\mathcal{A}$ un algorithme de tri. On suppose que $\mathcal{A}$ est basé sur des comparaisons deux à deux des éléments à trier. On compare deux éléments, selon le résultat on en compare deux autres, etc. jusqu'à ce qu'on puisse renvoyer le résultat.

Considérons l'arbre de décision a associé : chaque nœud correspond un test de type "si $u_i < u_j$", chaque feuille est le résultat final (la liste triée), donc une permutation des éléments de départ.

Il y a $n!$ manière d'ordonner n les n éléments de départ, donc $n!$ résultats possible, donc au moins $n!$ feuilles.

Chaque exécution de l'algorithme consiste à suivre un chemin dans l'arbre.

7.4.1 Complexité au pire

La hauteur de a est donc le nombre maximal de comparaisons effectuées (c'est-à-dire la complexité au pire). Notons-la h_n.

Enfin, a est un arbre binaire. On a donc $n! \leq 2^h$, d'où :

$$h_n \geq \log_2(n!).$$

Or, $\log_2(n!) \geqslant \log_2\left(\left(\frac{n}{2}\right)^{n/2}\right) = \frac{n}{2}\log_2(\frac{n}{2})$.

On en déduit que :

$$n\log(n) = O_{n\to\infty}(h_n)$$

Remarque : Si vous l'avez vu en maths, vous pouvez aussi utiliser que $\ln(n!) \underset{n\to\infty}{\sim} n\ln(n)$.

Ce qui signifie que la complexité au pire est au moins de l'ordre de $n\log(n)$.

En particulier, la méthode de tri fusion est (asymptotiquement) optimale. On peut éventuellement l'améliorer un peu, mais on ne peut pas trouver de méthode de complexité négligeable devant la sienne.

Remarque : On peut aussi concevoir des tris plus adaptés à telle ou telle situation : si les données sont déjà presque triées, si on veut un tri "en place" (dans un vecteur, sans avoir à utiliser d'autre vecteur)...

7.4.2 Complexité moyenne

Menons la même étude pour la complexité moyenne. Celle-ci est égale à la moyenne des hauteurs des feuilles de a. (En considérant que les feuilles d'arrivées sont équiprobables.)

Pour tout feuille F, on notera $h(F)$ sa hauteur. On note aussi NF le nombre de feuilles. Enfin, on notera $M(a)$ la hauteur moyenne des feuilles, c'est-à-dire la complexité en moyenne de $\mathcal{A}$.

Supposons qu'il existe une feuille F_1 de hauteur $\leq h_a - 2$. Alors on peut améliorer l'algorithme ! En effet : soient F_2 et F_3 deux feuilles "soeurs" de hauteur h_a. On les déplace et les colle sous F_1 (cf dessin...). On obtient un nouvel arbre qu'on note a', qui correspond donc à un nouvel algorithme $\mathcal{A}'$.

Dans le calcul de sa complexité moyenne, il y aura $h_a - 1 + 2.(h(F_1) + 1)$, et dans celle de $\mathcal{A}$ $2.h_a + h(F_1)$. Comme $h(F_1) \leq h_a - 2$, la complexité moyenne de $\mathcal{A}'$ est meilleure.

Soit b l'arbre obtenu après toutes les optimisations possibles. Donc $M(a) \geq M(b)$, et dans b toutes les feuilles sont de hauteur h_b ou $h_b - 1$.

Remarque : Donc b est équilibré, mais même mieux : il est complet.

Alors $M(b) \geq h_b - 1 \geq \log_2(NF) - 1$.

Puis comme avant...

8 Parcours d'arbres

Il arrive que l'ordre dans lequel on parcourt un arbre soit important.

8.1 Parcours en profondeur

Dans un parcours en profondeur, on visite chaque branche entièrement avant de passer à une autre branche. C'est le type de parcours le plus naturel à programmer, et c'est celui que nous avons employé jusqu'ici.

Admettons qu'on traite toujours le fils gauche avant le fils droit d'un nœud. On peut alors encore distinguer trois variantes selon le moment où l'étiquette sera traitée. Si l'étiquette est traitée avant les fils, le parcours est dit « préfixe », si elle est traités après, on dira que le parcours est « postfixe ». Sinon, il s'agit d'un parcours « infixe ».

Deuxième partie
Exercices

MPSI, option informatique
Arbres

1 Création d'un type somme

Exercice 1. * Listes

1. Créer un type pour représenter des listes chaînées.
2. Programmer le calcul de la longueur d'une liste, puis un prédicat pour tester si une liste est triée.

Exercice 2. ** Les entiers

Voici la définition mathématiques des entiers positifs (en théorie des ensembles) :

```
type entier = Zero | Succ of entier;;
```

1. Définir les entiers 2 et 3.
2. Programmer le test d'égalité et la relation d'ordre sur les entiers, puis l'addition et la multiplication. Vérifier à titre d'exemple que $2 + 3 = 3 + 2$.

2 Arbres binaires

Exercice 3. * Différents types d'arbre

Proposer un type Caml pour représenter les arbres dans chacune des situations suivantes. Pour chacun, programmer le calcul de la hauteur.

1. Arbres binaire strict où seules les feuilles ont une étiquette, sans arbre vide.
2. Arbres ternaires : chaque nœud a trois fils. Vide autorisé.
3. Arbres avec deux types de nœuds : des nœuds ayant deux fils, et des nœuds en ayant un seul.

Exercice 4. *! Fonctions élémentaires sur les arbres binaires

Dans cet exercice, on utilise le type suivant :

```
type 'a arbreBin = Vide | Noeud of 'a arbreBin * 'a * 'a arbreBin
```

1. Tester l'égalité de deux arbres. On fera la distinction entre le fils gauche et le fils droit.
2. Tester si deux arbres ont le même « squelette », c'est-à-dire sont égaux aux contenus des étiquettes près.
3. Écrire une fonction `mapArbre` qui prend en entrée une fonction f et un arbre a et qui renvoie l'arbre obtenu en appliquant f à chaque étiquette de a.
4. De même écrire une fonction `fold` prenant en entrée un arbre a, une lci o et son neutre e et qui compose toutes les étiquettes de a.
5. (**!) Tester si un arbre est « équilibré » , c'est-à-dire si pour chaque nœud, les hauteurs des deux fils différent d'au plus 1.
6. Prendre une liste et en ranger le contenu dans un « peigne », c'est-à-dire un arbre dont chaque nœud a au moins un fils vide.
7. (**) Prendre une liste et en ranger le contenu dans un arbre équilibré. Plusieurs méthodes sont envisageables.

Exercice 5. ** Arbre des appels pour le tri fusion

Écrire une fonction qui prend en entrée une liste, la trie selon l'algorithme du tri fusion, et renvoie en même temps l'arbre des appels effectués lors de ce tri.

On utilisera le type suivant :

```
type arbreAppels = Feuille of int list | Noeud of arbreAppels * int list * arbreAppels
```

L'expression `Feuille l`, où `l` est singleton ou vide représente un cas de base. L'expression `Noeud fg l fd` , représente un appel pour trier la liste `l`, `fg` et `fd` étant les arbres des deux appels récursifs effectués.

Exercice 6. * Dessiner un arbre binaire**

On souhaite écrire une fonction pour dessiner un arbre d'entiers. La fonction prendra en argument, outre l'arbre à représenter, un entier représentant la taille que prendra chaque feuille et la hauteur entre deux lignes, ainsi que la fonction d'affichage d'une étiquette (`draw_string` pour un arbre de chaînes, `fun x-> draw_string (string_of_int x)` pour un arbre d'entiers...)

Pour simplifier, on considérera des arbres binaires.

On pourra créer une fonction auxiliaire prenant comme arguments supplémentaires abscisse et ordonnée du coin haut gauche du rectangle où il faudra tracer l'arbre, et renvoyant comme valeur supplémentaire l'abscisse de l'extrémité droite de ce rectangle.

3 Variantes sur les arbres binaires

Exercice 7. * Arbre syntaxique d'une formule

Il est très pratique de représenter une formule à l'aide d'un arbre. A chaque nœud on place une opération, et à chaque feuille un nombre.

1. Dessiner l'arbre correspondant à la formule $-(2+1)*3+2/4$.
2. On propose le type suivant pour implémenter un arbre de formule :

```
type formule =
   Feuille of int
  | Somme of formule*formule
  | Produit of formule*formule
  | Quotient of formule*formule
  | Oppose of formule
;;
```

   Écrire une fonction `calcule` prenant en entrée une formule et calculant son résultat.
3. (Avec le cours sur les piles) Écrire une fonction prenant une expression algébrique postfixée et renvoyant l'arbre de cette formule.

Exercice 8. * Analyse syntaxique**

Cet exercice est la suite logique de 7 : le but est de lire une chaîne de caractères représentant une formule, et de créer l'arbre correspondant pour pouvoir calculer le résultat. Vous aurez alors programmé votre propre calculette !

Pour simplifier, on ne prendra pas en compte la priorité des opérations : on supposera que l'utilisateur mets des parenthèses partout.

Remarque : En comparaison avec la notation polonaise inversée, nous avons là une méthode plus compliquée à programmer, mais plus agréable pour un utilisateur.

1. Écrire une fonction `parentheseCorrespondante` qui prend en entrée une chaîne m, un indice i tel que $M.[i]$ soit une parenthèse ouvrante, et qui renvoie l'indice de la parenthèse fermante correspondante.

   On lèvera une erreur si au cours du procédé on constate que m était mal parenthésé.
2. Écrire une fonction `arbre_of_string` qui prend la chaîne et la traduit en une formule selon l'exercice 7.

   En déduire une fonction pour évaluer une formule tapée par l'utilisateur. Lui donner si possible un nom d'animal, serpents et chameaux étant déjà pris.

Exercice 9. ** Une énigme On dispose de 12 boules numérotées et d'une balance Roberval. Les boules sont toutes de même poids sauf une.

1. Proposer un algorithme pour déterminer quelle boule a un poids différent des autres en trois pesées.
2. En déduire que $12 \leqslant 27$.
3. Proposer une méthode analogue pour démontrer que $27 \leqslant 27$.

4 Arbres de valence non bornée

Exercice 10. *! Fonctions élémentaires pour un arbre quelconque

On utilise ici le type :

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

Écrire des fonctions pour :

1. tester si un arbre est une feuille.
2. Calculer la somme des étiquettes.
3. Tester la présence d'un élément x parmi les étiquettes.
4. Calculer le maximum des étiquettes.
5. Calculer la hauteur d'un arbre.
6. Appliquer une fonction `f int -> int` à toutes les étiquettes d'un arbre.

Exercice 11. ** Modèle de Galton-Watson

Le modèle de Galton-Watson étudie la probabilité d'extinction d'une espèce animale. On considère que chaque individu a la probabilité 1/8 d'avoir 3 descendants, 3/8 d'avoir 2 descendants, 3/8 d'avoir 1 descendant, et 1/8 de n'avoir aucun descendant.

La fonction `Random.int` permet d'obtenir un entier aléatoire.

1. Écrire une fonction prenant en entrée un entier n et qui génère l'arbre généalogique issu d'un individu sur au plus n générations.
2. Modifier la fonction pour qu'elle renvoie en plus de l'arbre un booléen indiquant si la lignée s'est éteinte, c'est-à-dire si à la n-ème génération, il n'y a plus aucun descendant.
3. On peut démontrer que la probabilité d'extinction de la lignée est $\sqrt{5}-2$. Écrire une fonction permettant de vérifier ce chiffre.

Exercice 12. * ! Arbre des permutations d'une liste**

Étant donné une liste l on demande de construire un arbre dont les feuilles sont toutes les permutations de l.

On pourra reprendre la structure vue en cours : une fonction pour traiter un nœud, et une fonction pour visiter tous ses fils.

Ces deux fonctions prendront en arguments supplémentaire :

- une liste `dejaPlaces` contenant les éléments de l déjà placés ;
- une liste `elementsRestants` contenant les éléments de j pas encore placés.

L'idée étant que ces deux données permettent de savoir à quel nœud de l'arbre des permutation nous sommes.

Exercice 13. ** Afficher les fichiers d'un répertoire

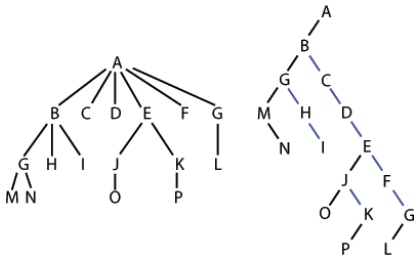
Écrire une fonction prenant une chaîne de caractères qui représente un répertoire, et renvoyant la liste des fichiers présents dans ce répertoire, y compris les fichiers dans les sous-répertoires, les sous-sous-répertoires, etc.

Les commandes utiles sont dans le module `Sys`.

- `Sys.chdir` : changer le répertoire courant ;
- `Array.to_list (Sys.readdir chemin)` : renvoyer la liste des fichiers et répertoires du répertoire désigné par `chemin`.
- `Sys.is_directory chemin` indique si `chemin` désigne un répertoire.

Exercice 14. * Transformer un arbre quelconque en un arbre binaire**

Écrire une fonction pour transformer un arbre quelconque en un arbre binaire, en suivant le principe illustré ci-dessous :



Que penser de la complexité pour atteindre un nœud dans la version binarisé de l'arbre comparée à celle dans l'arbre initial ?

5 Arbres binaires de recherche

Exercice 15. *! Fonctions élémentaires sur les ABR

Gardez un fichier `abr.ml` *facilement accessible en cas de besoin l'an prochain.*

On garde le même type Caml qu'à l'exercice 4.

Pour tout arbre a, on notera h_a sa hauteur et N_a son nombre de nœuds (feuilles incluses).

1. Dessiner un arbre binaire de 7 nœuds de hauteur minimale, puis de hauteur maximale.
2. Rechercher le minimum.
3. Renvoyer l'arbre privé de son minimum.
4. Ranger le contenu d'une liste dans un ABR.
5. Renvoyer la liste des étiquettes contenues dans un ABR, dans l'ordre croissant.
6. Combiner les fonctions précédentes pour obtenir un tri. Quelle est sa complexité au pire ? Dans quel cas le pire se produit-il ?

   Ce tri est isomorphe au « tri par segmentation » qu'on étudiera, mais pour les tableaux, en seconde année.
7. (**) Ranger le contenu d'une liste triée dans un ABR.
8. Renvoyer l'arbre privé de tous les éléments inférieurs à un élément donné dans un ABR.
9. (**!) Écrire une fonction pour tester si un arbre binaire est un arbre de recherche.
10. (**) Renvoyer l'arbre privé d'un élément.

Exercice 16. ** Dédoublonnage

Écrire une fonction pour dédoublonner une liste, en utilisant un ABR pour enregistrer les éléments déjà rencontrés.
N.B. On utilise ici un ABR pour enregistrer un ensemble. En effet, cette structure permet d'implémenter efficacement l'ajout d'un élément et le test d'appartenance, qui sont les deux opérations élémentaires sur les ensembles. On peut donc dire que les ABR sont une manière d'implanter les ensembles.

Exercice 17. *! 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 18. ** Créer un ABR à partir d'une liste triée

1. Pourquoi la fonction `abr_of_list` vue en cours 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 19. ** 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 caractères 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.

6 Parcours d'arbre

Dans les exercices de cette partie, l'ordre dans lequel on parcourt l'arbre est important.

Exercice 20. ** Arbre généalogique « ascendant »

1. Proposer un type pour modéliser un arbre généalogique « ascendant » : chaque nœud représente un individu, et ses deux fils sont le père et la mère de celui-ci. Pour chaque individu, on enregistrera deux valeurs : le nom et le prénom de celui-ci.
2. Écrire une fonction prenant un arbre généalogiue ascendant ainsi qu'un prénom et renvoyant tous les ancêtres de la racines portant ce prénom.
3. Écrire une fonction prenant les même entrées et renvoyant un ancêtre le plus proche de la racine portant le prénom indiqué. On ne parcourra que la partie nécessaire de l'arbre.

Exercice 21. * Arbre généalogique « descendant »**

Dans cet exercice, on utilise des arbres où chaque nœud représente un individu, et où pour chacun on enregistre nom, dates de naissance et décès, et la liste de ses fils. Le type est défini ainsi :

```
type arbreGenealogique = noeud of string * int * int * (arbreGenealogique list);;
```

On conviendra de plus de donner les enfants dans l'ordre de leur naissance.

On considère une famille royale. Les règles de succession sont les suivantes : au décès du roi, son fils aîné prend sa place. Mais s'il n'a pas d'enfant, ou qu'ils sont déjà morts, on va chercher dans l'ordre : son frère, ses neveux, ses cousins, ses petits neveux, etc... Entre deux frères, c'est toujours l'aîné qui sera prioritaire.

Écrire une fonction prenant en entrée l'arbre généalogique d'une lignée royale, et renvoyant la liste des rois qui se sont succédé, ainsi que les dates de leurs règnes.

Exercice 22. * ! Des chiffres et pas des lettres, le retour**

1. (Révision) Écrire une fonction prenant un entier n et une liste d'entiers positifs l et renvoyant une liste d'éléments de l dont la somme vaut n. On peut employer plusieurs fois chaque élément de l.

   Le but est à présent de calculer la solution utilisant le moins d'éléments de l.

2. On considère l'arbre suivant :
   - À chaque nœud figure un entier k qui représente la somme restant à faire.
   - Pour un nœud étiqueté par un entier k, il y a un fils pour chaque élément x de l tel que $k - x \geqslant 0$ ce fils sera étiqueté par $k - x$.

   (a) Dessiner l'arbre obtenue pour une valeur de k et de l de votre choix.

   (b) Comment identifie-t-on dans l'arbre une suite d'éléments de l dont la somme vaut n ?

   (c) Comment parcourir l'arbre pour obtenir une solution minimale au problème ?

   (d) Programmer une fonction Caml correspondante.

Exercice 23. * ! 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.

Quelques indications

4 Pour la dernière question : une méthode est de placer la moitié de la liste dans chaque fils. On peut aussi utiliser le principe « j'insère à gauche puis j'échange les fils ».

6

8 Une formule doit être de la forme `bloc1 opérateur bloc2`, `opérateur` étant `+,*,-` ou `/`, et `bloc1` et `bloc2` pouvant être un bloc parenthésé ou un nombre.

Ainsi il s'agit de parcourir la formule, de chercher dans un premier temps un nombre ou une parenthèse, puis de chercher un opérateur, puis encore un nombre ou une parenthèse. Les chiffres sont les caractères de code ASCII dans $[\![48, 57]\!]$.

9 1.

2. Compter les feuilles.
3. Prendre plus de boules, mais en sachant à l'avance que la boule différente des autres est plus lourde.

11 1. Écrire une fonction qui renvoie 0, 1, 2, ou 3 selon les probas données par l'énoncé.
Ensuite écrire deux fonctions mutuellement récursives : l'une pour renvoyer un arbre aléatoire, et l'autre une forêt aléatoire.

12 Pour la fonction qui parcours les fils, rajouter encore deux arguments :

- `a_voir` : liste des fils restant à voir ;
- `vus` : liste des fils déjà vus.

14 La complexité sera la même.

15 5) Bien sûr, ne pas renvoyer un peigne... Il est préférable de mettre la médiane à la racine.
Pour tester si un arbre est un ABR, écrire une fonction auxiliaire qui renvoie en plus le min et le max de l'arbre.

16 Écrire une fonction auxiliaire prenant en argument supplémentaire un ABR contenant les élément déjà vus de la liste.

17 Faire des dessins !

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

21 Faire une fonction auxiliaire qui d'une part prend en argument supplémentaire la date de début de règne de la lignée étudiée, et d'autre part renvoie en valeur renvoyée supplémentaire la date de fin de règne de cette lignée.

23 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.

2

4

5

6 Attention : ceci est encore du Caml light...

```
#load "graphics.cma";;

#open "Graphics";; (* Pour éviter de devoir précéder les commandes du nom du module

let rec dessinAux a deb hauteur taille=
    match a with
        |feuille n  ->  lineto (hauteur*taille) deb;
                        draw_string (string_of_int n);
                        deb + taille
        |noeudBin( n, g,d) ->
            begin
              let milieu = dessinAux g deb (hauteur +1) taille in
               (* on dessine le fils gauche, et on recupère l abscisse de sa fin*)
               lineto (hauteur * taille) milieu;
               draw_string (string_of_int n);
               let fin = dessinAux d milieu (hauteur +1) taille in
                (*on dessine le fils droit et on recupère l abscisse de sa fin*)
                   lineto (hauteur*taille) milieu; (*retour au noeud de départ*)
                   fin
            end
;;


let dessin a taille = dessinAux a 0 0 taille;;

open_graph " 800x600";;
dessin exempleBin 100;;
```

7

8

10

11

12

15 6) : $\log_2(\frac{\sqrt{5}-1}{2})$.

16

17

```
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)
;;
```

18

```
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.)

19

21

23

7 Autres exercices

Exercices non présents dans la feuille de TD.

Exercice 24. * Numérotation Sosa**

Le numérotation Sosa binaire d'un arbre binaire a consiste à attribuer un entier à chaque nœud de la manière suivante :

- La racine est numérotée 1
- Si un nœud porte le numéro k, son fils gauche (si il existe) est numéroté $2k$ et son fils droit (si il existe) $2k+1$.

1. Écrire une fonction prenant un arbre binaire (sans étiquette pour simplifier) et renvoyant l'arbre ayant le même squelette mais étiqueté selon la numérotation Sosa.
2. (***) Écrire une fonction prenant en entrée la liste des étiquettes de Sosa d'un arbre et reconstruisant cet arbre.