# Option informatique
# corrigé de la deuxième épreuve

Lycée Louis-le-Grand, Paris

mars 2000

# 1 Arbres et dictionnaires

## 1.1 Profondeur et taille

### 1.1.1 Profondeur

```
1  let rec profondeur = function
2      | Vide -> -1
3      | Feuille -> 0
4      | Nœud(g,d) -> max (profondeur g) (profondeur d)
5      | NœudFeuille(g,d) -> max (profondeur g) (profondeur d) ;;
```

### 1.1.2 Taille

On procède à une démonstration par induction structurelle pour prouver que la fonction $\mathrm{Card} \circ \mu$ dénombre les `Feuille` et `NœudFeuille` d'un arbre-dictionnaire.
On a bien sûr d'abord $\mathrm{Card}\,\mu(\texttt{Vide}) = \mathrm{Card}\,\emptyset = 0$ et $\mathrm{Card}\,\mu(\texttt{Feuille}) = \mathrm{Card}\{\varepsilon\} = 1$.
Puis $\mathrm{Card}\,\mu(\texttt{Nœud}(\texttt{g},\texttt{d})) = \mathrm{Card}(\texttt{G}.\mu(\texttt{g}) \cup \texttt{D}.\mu(\texttt{d})) = \mathrm{Card}(\texttt{G}.\mu(\texttt{g})) + \mathrm{Card}(\texttt{D}.\mu(\texttt{d}))$ car un mot ne peut commencer à la fois par un `G` et un `D`, et l'union considérée est disjointe. Alors finalement on a bien $\mathrm{Card}\,\mu(\texttt{Nœud}(\texttt{g},\texttt{d})) = \mathrm{Card}\,\mu(\texttt{g}) + \mathrm{Card}\,\mu(\texttt{d})$.
De même $\mathrm{Card}\,\mu(\texttt{NœudFeuille}(\texttt{g},\texttt{d})) = \mathrm{Card}(\{\varepsilon\} \cup \texttt{G}.\mu(\texttt{g}) \cup \texttt{D}.\mu(\texttt{d})) = 1 + \mathrm{Card}(\texttt{G}.\mu(\texttt{g})) + \mathrm{Card}(\texttt{D}.\mu(\texttt{d}))$ car un mot ne peut commencer à la fois par un `G` et un `D` ni être vide et avoir une initiale, et l'union considérée est disjointe. Alors finalement on a bien $\mathrm{Card}\,\mu(\texttt{NœudFeuille}(\texttt{g},\texttt{d})) = 1 + \mathrm{Card}\,\mu(\texttt{g}) + \mathrm{Card}\,\mu(\texttt{d})$.

```
6  let rec taille = function
7      | Vide -> 0
8      | Feuille -> 1
9      | Nœud(g,d) -> (taille g) + (taille d)
10     | NœudFeuille(g,d) -> 1 + (taille g) + (taille d) ;;
```

### 1.1.3 Dénombrement

L'arbre peut n'avoir qu'une feuille tout en bas et donc le minorant est $1 \leqslant n$.
Tout nœud de l'arbre peut être un `NœudFeuille`, ce qui fournit la majoration $n \leqslant 2^{p+1} - 1$, correspondant à un arbre complet.
Finalement : $1 \leqslant n \leqslant 2^{p+1} - 1$.

## 1.2 La fonction $\mu$

```
let rec préfixe c = function
    | [] -> []
    | l :: q -> (c :: l) :: (préfixe c q) ;;

let rec mu = function
    | Vide -> []        (* l'ensemble vide *)
    | Feuille -> [ [] ] (* le singleton constitué du mot vide *)
    | Nœud(g,d) ->
        let gl = mu g and dl = mu d
        in
        (préfixe G gl) @ (préfixe D dl)
    | NœudFeuille(g,d) ->
        let gl = mu g and dl = mu d
        in
        [[]] @ (préfixe G gl) @ (préfixe D dl) ;;
```

## 1.3 Recherche d'un mot

```
let rec cherche m a = match (m,a) with
    | _ , Vide -> false
    | _ , Feuille -> m = []
    | [] , Nœud(g,d) -> false
    | G::q , Nœud(g,d) -> cherche q g
    | D::q , Nœud(g,d) -> cherche q d
    | [] , NœudFeuille _ -> true
    | G::q , NœudFeuille(g,d) -> cherche q g
    | D::q , NœudFeuille(g,d) -> cherche q d ;;
```

## 1.4 Insertion d'un mot

### 1.4.1 Fonctions auxiliaires

```
let nfon = function
    | Nœud(g,d) -> NœudFeuille(g,d)
    | a -> a ;;

let rec arbre1 = function
    | [] -> Feuille
    | G :: q -> Nœud(arbre1 q,Vide)
    | D :: q -> Nœud(Vide,arbre1 q) ;;

let rec arbre2 a b = match (a,b) with
    | G :: a' , G :: b' -> Nœud(arbre2 a' b',Vide)
    | D :: a' , D :: b' -> Nœud(Vide,arbre2 a' b')
    | G :: a' , D :: b' -> Nœud(arbre1 a',arbre1 b')
    | D :: a' , G :: b' -> Nœud(arbre1 b',arbre1 a')
    | [] , [] -> Feuille
    | [] , b -> nfon (arbre1 b)
    | a , [] -> nfon (arbre1 a) ;;
```

### 1.4.2 La fonction d'insertion

```
let rec insère m = function
    | Vide -> arbre1 m
    | Feuille ->
       begin
           match m with
               | [] -> Feuille
               | _ -> nfon (arbre1 m)
       end
    | Nœud(g,d) ->
       begin
           match m with
               | [] -> NœudFeuille(g,d)
               | G :: m' -> Nœud(insère m' g,d)
               | D :: m' -> Nœud(g,insère m' d)
       end
    | NœudFeuille(g,d) ->
       begin
           match m with
               | [] -> NœudFeuille(g,d)
               | G :: m' -> NœudFeuille(insère m' g,d)
               | D :: m' -> NœudFeuille(g,insère m' d)
       end ;;
```

## 1.5 Suppression d'un mot

```
let rec supprime m a = match a with
    | Vide -> a                    (* le mot ne figurait pas dans l'arbre-dictionnaire *)
    | Feuille ->
       begin
           match m with
               | [] -> Vide
               | _ -> a            (* le mot ne figurait pas dans l'arbre-dictionnaire *)
       end
    | Nœud(g,d) ->
       begin
           match m with
               | [] -> a           (* le mot ne figurait pas dans l'arbre-dictionnaire *)
               | G :: m' -> Nœud(supprime m' g,d)
               | D :: m' -> Nœud(g,supprime m' d)
       end
    | NœudFeuille(g,d) ->
       begin
           match m with
               | [] -> Nœud(g,d)
               | G :: m' -> NœudFeuille(supprime m' g,d)
               | D :: m' -> NœudFeuille(g,supprime m' d)
       end ;;
```

# 2 Arbres et files binomiaux

## 2.1 Arbres binomiaux

### 2.1.1

Dessinons des arbres binomiaux de degrés 0, 1, 2 et 3.

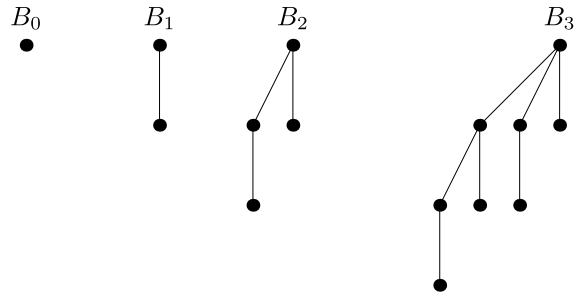


FIG. 1: les premiers arbres binomiaux

### 2.1.2

En détachant la branche gauche d'un arbre binomial de degré $k$, on obtient encore un arbre binomial mais de degré $k-1$. Comme cette branche gauche est elle-même un arbre binomial de degré $k-1$, on a une biejction entre les arbres binomiaux de degré $k$ et les couples d'arbres binomiaux de degré $k-1$.

### 2.1.3

Il existe un unique squelette d'arbre binomial de degré $k$. Appelons $N(k)$, $H(k)$, $NS(k,p)$ et $D(k)$ respectivement le nombre de noeuds, la hauteur (profondeur maximale d'une feuille), le nombre de noeuds à la profondeur $p$ et le nombre maximal de fils d'un noeud d'un arbre binomial de degré $k$. On a donc $N(0)=1$, $H(0)=0$, $NS(p,0)=1$ si $p=0$, 0 sinon, $D(0)=0$. De plus, la définition récursive des arbres binomiaux nous montre que l'on a les relations de récurrence

$$\begin{aligned} N(k) &= N(k-1)+N(k-2)+\ldots+N(0)+1 \\ H_k &= \max(H(k-1),\ldots,H(0))+1 \\ NS(p,k) &= NS(p-1,k-1)+NS(p-1,k-2)+\ldots+NS(p-1,0) \text{ si } p\geq 1 \\ NS(0,k) &= 1 \\ D(k) &= \max(k,D(k-1),\ldots,D(1),D(0) \end{aligned}$$

Nous allons donc montrer par récurrence que $N(k)=2^k$, $H(k)=k$, $NS(p,k)=C_k^p$ (avec la convention habituelle si $p>k$), $D(k)=k$. Supposons ces formules vraies pour tous les arbres binomiaux de degré inférieur ou égal à $k-1$. On a alors

$$\begin{aligned} N(k) &= 2^{k-1}+2^{k-2}+\ldots+2+1+1=(2^k-1)+1=2^k \\ H_k &= \max(k-1,\ldots,1,0)+1=k-1+1=k \\ D(k) &= \max(k,k-1,\ldots,1,0)=k \\ NS(0,k) &= 1 \end{aligned}$$

et enfin, si $p\geq 1$

$$NS(p,k)=C_{k-1}^{p-1}+C_{k-2}^{p-1}+\ldots+C_0^{p-1}=C_k^p$$

cette dernière égalité résultant immédiatement de

$$C_k^p=C_{k-1}^{p-1}+C_{k-1}^p=C_{k-1}^{p-1}+C_{k-2}^{p-1}+C_{k-1}^{p-1}+C_{k-2}^p=\ldots$$

## 2.2 Arbre binomial croissant

On a écrit

```
1  type abinom = Nœud of int * abinom list;;
```

Pour obtenir la liste des étiquettes des noeuds d'un arbre (qu'il soit binomial ou non) ou d'une liste d'arbres binomiaux (ce que l'on appellera ensuite une file), on peut écrire (en utilisant l'opérateur `@` de concaténation de listes)

```
2  let rec liste_elts_arbre = function
3      | Nœud (x,[]) -> [x]
4      | Nœud (x,l) -> x :: (liste_elts_file l)
5  and liste_elts_file = function
6      | [] -> []
7      | t :: r -> (liste_elts_arbre t) @ (liste_elts_file r) ;;
```

On pourrait aussi utiliser la fonction `it_list`, en écrivant

```
let liste_elts_arbre = function
       | Nœud(x,l) -> it_list (fun a b -> a @ (liste_elts_arbre b)) [ x ] l ;;
```

Pour fusionner deux arbres binomiaux croissants $A_1$ et $A_2$ de degré $k-1$, de racines respectives $a_1$ et $a_2$, on commence par repérer celui qui a la plus petite racine (par exemple $a_1 \leq a_2$) ; on construit un nouvel arbre binomial croissant en prenant comme racine $a_1$, en mettant comme branche gauche l'arbre $A_2$ et comme autres branches les branches de $A_1$. On obtient clairement par induction structurelle un nouvel arbre binomial croissant : la racine $a_1$ est plus petite que son fils gauche $a_2$ et que ses autres fils (qui étaient déjà ses fils dans $A_1$). Ceci conduit à la fonction

```
8  let fusion a1 a2 = match (a1,a2) with
9          Nœud (x1,l1), Nœud (x2,l2) ->
10             if x1 <= x2 then Nœud (x1, Nœud (x2,l2) :: l1)
11             else Nœud (x2, Nœud (x1,l1) :: l2) ;;
```

## 2.3 File binomiale

### 2.3.1

L'écriture en base 2 d'un entier $n$ étant unique, on peut écrire $n$ de façon unique sous la forme

$$n = 2^{k_1} + 2^{k_2} + \ldots + 2^{k_p}$$

avec $0 \leq k_1 < k_2 < \ldots < k_p$ ce qui montre qu'il existe un unique squelette de file binomial de taille $n$, composé d'arbres binomiaux de degrés $k_1, k_2, \ldots, k_p$.

### 2.3.2

L'algorithme d'insertion d'un arbre binomial dans une file binomiale est tout à fait analogue à l'algorithme d'addition en base 2, dans le cas où le nombre ajouté est une puissance de 2, donc de la forme $10\ldots0$. On garde tous les chiffres à droite du 1 que l'on ajoute. Si le 1 se trouve en face d'un zéro, on se contente de l'écrire ; s'il se trouve en face d'un 1, on met un zéro à la place (ici en fait on fusionnera les deux arbres) et on cherche à ajouter $100\ldots0$ au nombre obtenu (on propage la *retenue* qui est ici l'arbre fusionné). Nous aurons besoin de comparer les degrés de deux arbres binomiaux, ce qui nous conduit à introduire une fonction auxiliaire qui calcule ce degré.

```
12 let rec deg_bin = function
13     | Nœud (_, []) -> 0
14     | Nœud (_, t :: _) -> deg_bin t + 1 ;;
```

```
let rec insère abin = function
    | [ ] -> [ abin ]
    | t :: r as l
         -> let d1 = deg_bin abin and d2 = deg_bin t
            in
            if d1 < d2 then abin :: l
            else if d2 < d1 then t :: (insère abin r)
            else insère (fusion abin t) r ;;
```

La fusion de deux files se fait alors de manière évidente

```
let rec fusion_file l1 l2 = match l2 with
    | [] -> l1
    | t2 :: r2 -> fusion_file (insère t2 l1) r2 ;;
```

### 2.3.3

Il suffit bien évidemment d'insérer dans la file l'arbre binomial de degré 0 ayant cet unique élément

```
let ajoute file n = insère (Nœud (n,[])) file ;;
```

### 2.3.4

L'élément minimal de la file binomiale est évidemment la plus petite racine de tous les arbres qui la composent. On peut procéder en deux temps : tout d'abord on recherche la plus petite racine, puis on recherche le premier arbre qui la contient, on ôte cet arbre, on lui retire sa racine puis on réinsère chacune des branches dans la file raccourcie ; la liste des branches étant elle-même une file binomiale à l'envers, il suffit d'utiliser la fonction de fusion précédemment définie. Notre fonction renverra le couple formé de l'élément minimal et de la file réduite.

```
let enlève_min file =
    let rec cherche_min = function
        | [ ] -> failwith "liste vide"
        | [ Nœud(x,_) ] -> x
        | Nœud(x,_) :: r -> min x (cherche_min r)
    and enlève_elt n = function
        | [] -> failwith "liste vide"
        | Nœud(x, l) :: r -> if n <> x then insère (Nœud(x,l)) (enlève_elt n r)
                             else fusion_file r (rev l)
    in
    let n = cherche_min file in (n , enlève_elt n file) ;;
```

### 2.3.5

Il suffit évidemment pour trier une liste d'entiers de les insérer un par un dans une file binomiale, puis de les retirer un par un. On obtient

```
let tri_binomial l =
    let rec liste_en_file = function
          | [] -> []
          | t :: r -> insère (Nœud(t,[])) (liste_en_file r)
    and file_en_liste = function
          | [] -> []
          | file -> let (min,nouvelle_file) = enlève_min file
                    in min :: (file_en_liste nouvelle_file)
    in file_en_liste (liste_en_file l) ;;
```