TD 8

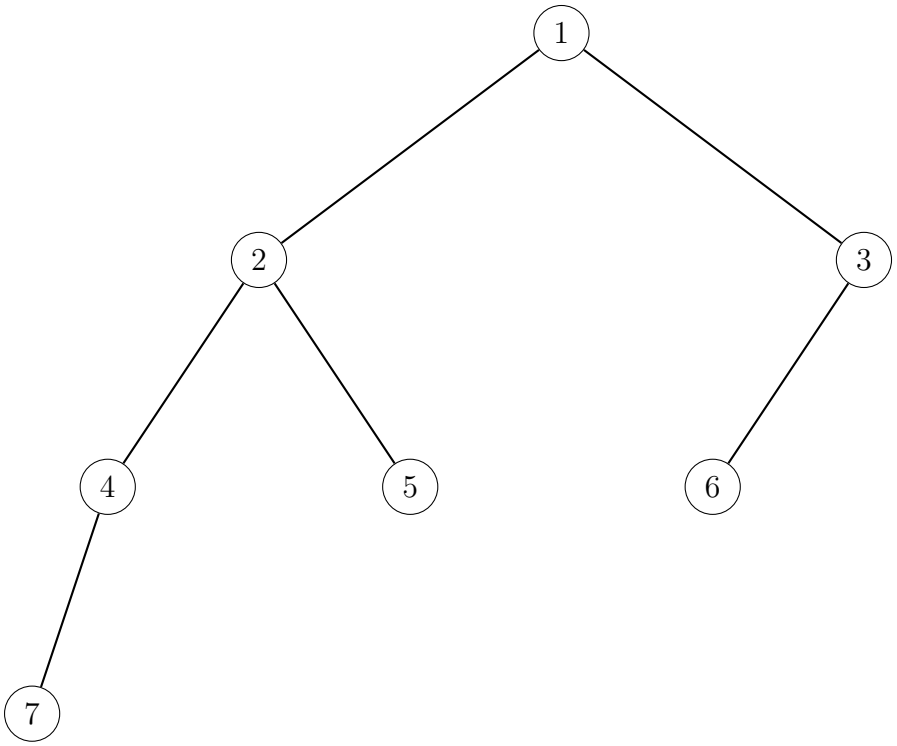
Arbres binaires

Correction

On utilisera le type suivant :

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

1. Définir l'arbre suivant : ex1 :



Correction

Listing 1 – L'implémentaion de l'exemple

```
let ex_1 =
Noeud(Noeud(Noeud(
Noeud(Vide,7,Vide),4,Vide),2,Noeud(Vide,5,Vide)
), 1,Noeud(Noeud(Vide,6,Vide),3,Vide));;
```

2. Écrire une fonction `taille` qui calcule la taille d'un arbre.

```
taille : 'a arbre_bin -> int
```

Correction

Listing 2 – Fonction `taille`

```
let rec taille arbre =
  match arbre with
  |Vide -> 0
  | Noeud(a_g,etiquette,a_d) ->
    taille a_g + taille a_d +1;;
```

3. Écrire une fonction `hauteur` qui calcule la hauteur d'un arbre :

   `hauteur : 'a arbre_bin -> int`

   Correction

Listing 3 – Fonction `hauteur`

```
let rec hauteur arbre =
  match arbre with
  |Vide -> -1
  | Noeud(a_g,etiquette,a_d) ->
    (max (hauteur a_g)  (hauteur a_d)) +1;;
```

4. Écrire une fonction `maxi` qui renvoie la plus grande étiquette d'un arbre.

   `maxi : 'a arbre_bin -> 'a`

   Correction

Listing 4 – Fonction `maxi`

```
let rec maxi arbre =
  match arbre with
  | Vide -> failwith "␣erreur␣maxi␣"
  | Noeud(Vide,e , Vide ) -> e
  | Noeud(a_g,e,Vide) -> let m_a_g = maxi a_g in
      max m_a_g e
  | Noeud(Vide,e,a_d) -> let m_a_d = maxi a_d in
      max m_a_d e
  | Noeud(a_g,e,a_d) -> let m_a_g = maxi a_g in
                  let m_a_d = maxi a_d in
          max m_a_d (max m_a_g e);;
```

5. Écrire une fonction `test_complet` qui vérifie si un arbre est complet.

   `test_complet : 'a arbre_bin -> bool`

   Correction

Listing 5 – Fonction `test_complet`

```
let test_complet arbre =
  let rec aux arbre =
  (* la fonction renvoie le test et la hauteur de l'arbre *)
      match arbre with
      | Vide -> failwith "␣erreur␣arbre␣vide␣"
```

```
      | Noeud(Vide,e , Vide ) -> (true , 0)
      | Noeud(a_g,e,Vide) -> let (test_gauche , hauteur_g ) = aux a_g in
          ( false , hauteur_g +1 )
      | Noeud(Vide,e,a_d) -> let (test_droite , hauteur_d ) = aux a_d in
          ( false , hauteur_d +1 )
      | Noeud(a_g,e,a_d) -> let (test_gauche , hauteur_g ) = aux a_g in
          let (test_droite , hauteur_d ) = aux a_d in
          ( test_gauche && test_droite &&  hauteur_g = hauteur_d,
                1+ max hauteur_g hauteur_d)
    in
  fst( aux arbre);;
```

6. On souhaite construire un arbre binaire équilibré à partir d'un tableau. On propose de prendre pour racine l'élément d'indice 0 du tableau, puis pour un nœud étiqueté par un élément d'indice k, le fils gauche est l'élément d'indice $2k+1$, et le fils droit l'indice $2k+2$, sous réserve d'existence. Écrire la fonction `arbre_of_vect` qui convertit un tableau en arbre binaire équilibré.

   `arbre_of_vect : 'a array -> 'a arbre_bin`

   Correction

Listing 6 – Fonction `arbre_of_vect`

```
let arbre_of_vect tab =
  let taille = Array.length tab in
  let rec aux k =
      if k >= taille then Vide
      else
      let a_g = aux (2*k+1) in
      let a_d = aux ( 2*k+2) in
      Noeud( a_g , tab.(k) , a_d )
  in
  aux 0;;
```

Parcours d'un arbre

7. Parcours en pré-ordre (préfixe) :
Définition : Parcourir un arbre binaire non vide en pré-ordre consiste à traiter la racine de cet arbre, puis à parcourir en pré-ordre l'arbre fils gauche de cette racine et enfin, à parcourir en pré-ordre l'arbre fils droit de cette racine. (parcourir un arbre binaire vide en pré-ordre est une opération nulle (pas de traitement)).

Écrire une fonction `prefixe` qui prend en argument un arbre et rend la liste de ses arguments, donnée en parcourant l'arbre de façon préfixe.

`prefixe : 'a arbre_bin -> 'a list`

Correction

Listing 7 – Fonction `prefixe`

```
let rec prefixe arbre =
    match arbre with
    | Vide -> []
    | Noeud ( a_g,e ,a_d) ->
        let l_g = prefixe a_g in
        let l_d = prefixe a_d in
        e :: l_g @ l_d;;
```

8. **Parcours en ordre (infixe) :**

Définition : Parcourir un arbre binaire non vide en ordre (ou de façon infixe) consiste à parcourir en ordre l'arbre fils gauche de cet arbre, puis à traiter la racine, et enfin, à parcourir en ordre l'arbre fils droit de cette racine. (parcourir un arbre binaire vide en pré-ordre est une opération nulle (pas de traitement)).

Écrire une fonction `infixe` qui prend en argument un arbre et rend la liste de ses arguments, donnée en parcourant l'arbre de façon infixe.

`infixe : 'a arbre_bin -> 'a list`

Correction

Listing 8 – Fonction `infixe`

```
let rec infixe arbre =
    match arbre with
    | Vide -> []
    | Noeud ( a_g,e ,a_d) ->
        let l_g = infixe a_g in
        let l_d = infixe a_d in
        l_g @ (e::l_d)    ;;  (*  l_g @ [e] @ l_d  *)
```

9. **Parcours en post-ordre (postfixe) :**
Définition : Parcourir un arbre binaire non vide en post-ordre (ou de façon postfixe) consiste à parcourir en post-ordre l'arbre fils gauche de cette arbre, puis à parcourir en

post-ordre l'arbre fils droit, et enfin à traiter la racine. (parcourir un arbre binaire vide en pré-ordre est une opération nulle (pas de traitement)).

Écrire une fonction `postfixe` qui prend en argument un arbre et rend la liste de ses arguments, donnée en parcourant l'arbre de façon infixe.

`postfixe : 'a arbre_bin -> 'a list`

Correction

Listing 9 – Fonction `postfixe`

```
let rec postfixe arbre =
    match arbre with
    | Vide -> []
    | Noeud ( a_g,e ,a_d) ->
        let l_g = postfixe a_g in
        let l_d = postfixe a_d in
        l_g @ l_d @ [e]     ;;
```

10. **Parcours militaire :**
Il faut ajouter l'ordre militaire, qui consiste à lire profondeur par profondeur, de gauche à droite. C'est un parcours en largeur d'abord.

Écrire une fonction `militaire` qui prend en argument un arbre et rend la liste de ses arguments, donnée en parcourant l'arbre de façon militaire.

`militaire : 'a arbre_bin -> 'a list`

Correction

Listing 10 – Fonction `parcours_militaire`

```
(* recuperation des files du TD sur les files  *)

type 'a file ={mutable file :'a list};;


let cree_file () = {file = []};;


let file_vide fil = fil.file = [];;


let ajoute fil element = fil.file <- fil.file @ [element];;

let retire fil = if file_vide fil then failwith "erreur retire"
  else
    ( let a::reste = fil.file in
      fil.file <- reste ;
      a);;
```

```
let parcours_militaire arbre =
  let f = cree_file() in

  let rec aux l =
    if file_vide f then l
    else
      begin
        let a = retire f in
        match a with
        | Vide -> aux l
        | Noeud(a_g,e,a_d) ->
            begin
              ajoute f a_g ;
              ajoute f a_d ;
              aux (l@[e])
            end
      end
    in
    ajoute f arbre ;
    aux [];;
```