TD 9

Arbres binaires de recherche

Correction

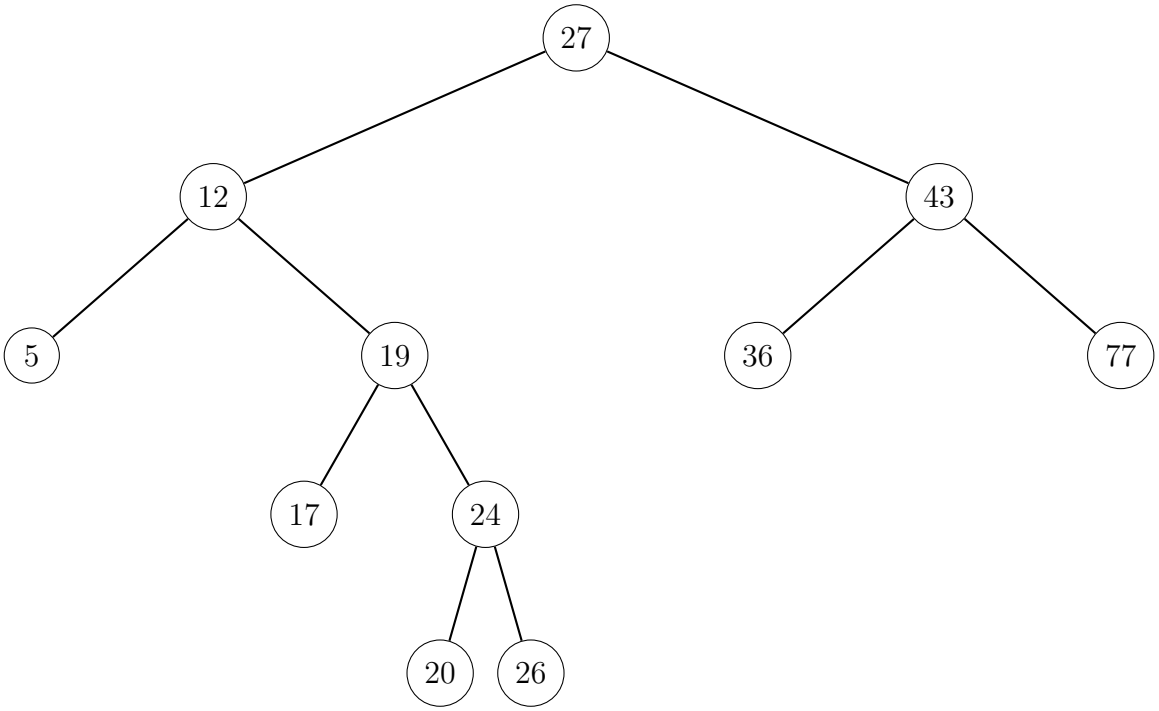
Un arbre binaire non vide étiqueté est appelé arbre binaire de recherche si les étiquettes appartiennent à un ensemble totalement ordonné et s'il vérifie l'une des deux conditions équivalentes suivantes :

- La liste des étiquettes en ordre infixe est croissante au sans large.
- Pour tout nœud x d'étiquette e, les étiquettes des éléments de la branche gauche de x sont inférieures ou égales à e et les étiquettes des éléments de la branche droite de x sont supérieures ou égales à e.

On utilisera le type suivant :

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

1. Définir l'arbre binaire de recherche suivant :`ex_1` :



Pour comparer, on utilise le type :

```
type comparaison = Inf | Egal | Sup;;
```

On peut ainsi créer une fonction pour comparer deux entiers :

```
let compare_int x y =
  if x<y then Inf else if x=y then Egal else Sup;;
```

2. Écrire la fonction `recherche` qui, pour un x donné, ainsi qu'une fonction de comparaison donnée vérifie si x est l'étiquette d'un arbre de recherche donné :

   `recherche : 'a arbre -> 'b -> ('b -> 'a -> comparaison) -> bool`

   Correction

Listing 1 – La fonction `recherche`

```
let rec recherche arbre element compare =
  match arbre with
  | Vide -> false
  | Noeud(a_g,etiquette,a_d) -> let equiv = compare element etiquette in
    equiv = Egal || (if equiv = Inf then
        recherche a_g element compare else
        recherche a_d element compare);;
```

3. Écrire une fonction qui retourne l'étiquette du nœud le plus à gauche, ainsi que l'étiquette du nœud le plus à droite. La fonction renvoie un message d'erreur si l'arbre est vide.

   `min_gauche_max_droit : 'a arbre -> 'a * 'a`

   Correction

Listing 2 – La fonction `min_gauche_max_droit`

```
let rec min_gauche_max_droit arbre =
  match arbre with
  | Vide -> failwith "Arbre␣vide␣"
  | Noeud( Vide , etiquette ,Vide ) -> ( etiquette , etiquette )
  | Noeud( Vide , etiquette ,a_d ) -> (etiquette , snd(
min_gauche_max_droit a_d))
    | Noeud( a_g , etiquette ,Vide ) ->(fst(min_gauche_max_droit a_g) ,
etiquette)
  | Noeud( a_g , etiquette ,a_d ) -> (fst(min_gauche_max_droit a_g),
     snd(min_gauche_max_droit a_d));;
```

   En déduire la fonction `verif_arb_rech` qui retourne vrai si l'arbre est un arbre de recherche, et faux sinon.

   `verif_arb_rech : 'a arbre -> ('a -> 'a -> comparaison) -> bool`

   Correction

Listing 3 – La fonction `verif_arb_rech`

```
let verif_arb_rech arbre compare =
  let rec aux arbre =
    match arbre with
    | Vide -> failwith "Arbre␣vide␣"
    | Noeud( Vide , etiquette ,Vide ) -> ( etiquette , etiquette , true)
    | Noeud( Vide , etiquette ,a_d ) -> let ( min_d, max_d,bool_d ) = aux
 a_d in
          ( etiquette , max_d , bool_d && (compare etiquette min_d = Egal
 ||
```

```
            compare etiquette min_d = Inf))
    | Noeud( a_g , etiquette ,Vide ) -> let ( min_g, max_g,bool_g ) = aux
 a_g in
          ( min_g ,etiquette , bool_g && (compare etiquette max_g = Egal
||
            compare etiquette max_g = Sup))
    | Noeud( a_g , etiquette ,a_d ) ->
      let ( min_d, max_d,bool_d ) = aux a_d in
      let ( min_g, max_g,bool_g ) = aux a_g in
          ( min_g ,max_d , bool_d&& bool_g && (compare etiquette max_g =
Egal ||
            compare etiquette max_g = Sup) &&
            (compare etiquette min_d = Egal ||
            compare etiquette min_d = Inf) )
    in
    let (min_g,max_d,bool) = aux arbre in
    bool;;
```

4. La deuxième méthode consiste à donner la liste des étiquettes dans l'ordre infixe et à vérifier que la liste est triée.
   Écrire une fonction `verif_tri` qui vérifie si une liste donnée est triée par une relation d'ordre donné.

   ```
   verif_tri : 'a list -> ('a -> 'a -> comparaison) -> bool
   ```

   Correction

Listing 4 – La fonction `liste_bien_triee`

```
let rec liste_bien_triee liste compare =
  match liste with
    | [] -> true
    | [a] -> true
    | a::b::reste ->((compare a b = Egal)||
            (compare a b = Inf)) && (
            liste_bien_triee (b::reste) compare );;
```

En retrouvant la fonction infixe, en déduire la fonction `verif_arb_rech2`.

```
infixe : 'a arbre -> 'a list
verif_arb_rech2 : 'a arbre -> ('a -> 'a -> comparaison) -> bool
```

Correction

Listing 5 – La fonction `verif_2`

```
let verif_2 arbre compare =
  let l = infixe arbre in
  liste_bien_triee l compare;;
```

On désire insérer un nouvel élément dans un arbre binaire de recherche sans changer sa caractéristique d'arbre de recherche.

5. **Insertion aux feuilles :**
Écrire une fonction qui place un nouvelle élément sur une branche vide de l'arbre ou renvoie cet arbre si l'élément était déjà présent.

```
insere_feuille:'a arbre->'a ->('a->'a->comparaison)->'a arbre
```

Correction

Listing 6 – La fonction `insere_feuille`

```
let rec insere_feuille arbre element compare =
  match arbre with
    | Vide -> Noeud(Vide, element , Vide )
    | Noeud(a_g, etiquette , a_d ) ->
    if compare element etiquette = Sup
      then
      begin
        let a_d_2 = insere_feuille a_d element compare in
          Noeud(a_g,etiquette , a_d_2)
      end
  else
      begin
        let a_g_2 = insere_feuille a_g element compare in
          Noeud(a_g_2,etiquette , a_d)
      end ;;
```

6. **Insertion à la racine :**
Écrire une fonction qui place un nouvelle élément à la racine du nouvel arbre, ou renvoie cet arbre si l'élément était déjà présent.
L'idée est de découper l'arbre en deux, et de le recoller avec la nouvelle racine :

```
decoupe:'a arbre->'b->('b->'a->comparaison)->'a arbre * 'a arbre
insere_racine:'a arbre->'a->('a->'a->comparaison)->'a arbre
```

Correction

Listing 7 – La fonction `insere_racine`

```
let rec decoupe arbre element compare =
  match arbre with
  | Vide -> (Vide,Vide)
  | Noeud(a_g,etiquette,a_d) ->
    if compare element etiquette = Inf
    then
    let ( a_g_inf , a_g_sup) = decoupe a_g element compare in
      (a_g_inf , Noeud( a_g_sup , etiquette , a_d))
    else
    let ( a_d_inf , a_d_sup) = decoupe a_d element compare in
      ( Noeud( a_g , etiquette , a_d_inf) , a_d_sup);;


let insere_racine  arbre element compare =
  let (a_g,a_d )= decoupe arbre element compare in
    Noeud(a_g,element  , a_d );;
```

7. Écrire une fonction `construction_feuille` qui utilise la fonction `insere_feuille` pour construire un arbre de recherche composé des éléments d'une liste donnée.

   `construction_feuille:'a list->('a->'a->comparaison)->'a arbre`

   Correction

Listing 8 – La fonction `construction_feuille`

```
let rec construction_feuille liste compare=
  match liste with
    | [] -> Vide
    | a::reste ->
      let arbre_reste = construction_feuille reste compare in
      insere_feuille arbre_reste  a compare ;;
```

Écrire une fonction `construction_racine` qui utilise la fonction `insere_racine` pour construire un arbre de recherche composé des éléments d'une liste donnée.

`construction_racine:'a list->('a ->'a->comparaison)-> 'a arbre`

Correction

Listing 9 – La fonction `construction_racine`

```
let rec construction_racine liste compare=
  match liste with
    | [] -> Vide
    | a::reste ->
      let arbre_reste = construction_racine reste compare in
      insere_racine arbre_reste  a compare ;;
```

Comparer les arbres obtenus avec les listes :

```
let liste_ex_1 = [1;2;3;4;5;6;7;8;9];;
let liste_ex_2 = [1;3;5;7 ;9 ;8;6;4;2];;
```