TD 9

Arbres binaires de recherche

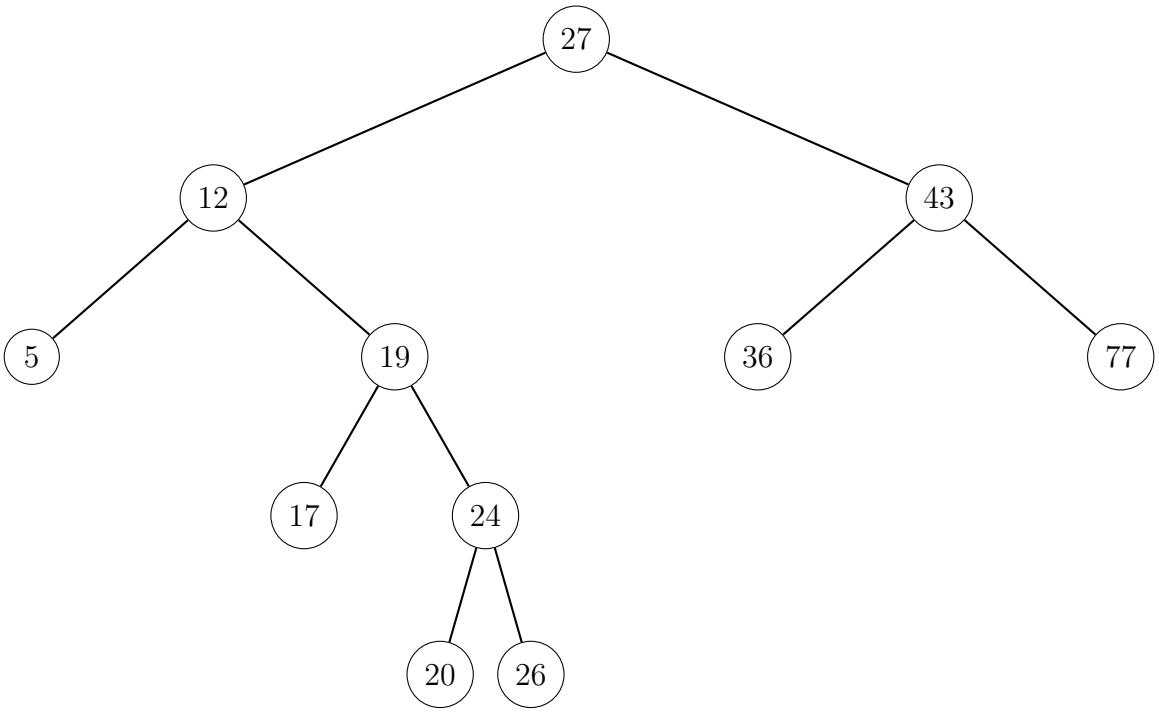
Un arbre binaire non vide étiqueté est appelé arbre binaire de recherche si les étiquettes appartiennent à un ensemble totalement ordonné et s'il vérifie l'une des deux conditions équivalentes suivantes :

- 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
```

3. Écrire une fonction qui retourne l'étiquette du nœud le plus à gauche, ainsi que l'étiquette du nœud le plus à droite.

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

   La fonction renvoie un message d'erreur si l'arbre est vide.
   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
   ```

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
   ```

   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
   ```

   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
   ```

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
   ```

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
   ```

   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
   ```

   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];;
   ```