Feuille d'exercices n°4

Travaux dirigés

Exercice I : Arbres binaires

Le type des arbres binaires étiquetés est défini dans le cours avec deux constructeurs :

- `Empty`, l'arbre vide ;
- `Node`, opérateur ternaire pour désigner l'opération d'ajout d'une racine pour connecter deux arbres.

Q1 – Donner la définition du type `'a btree` représentant les arbres binaires.

Q2 – Donner la définition de la fonction `taille : 'a btree -> int` qui calcule la taille d'un arbre (i.e. son nombre de nœuds).

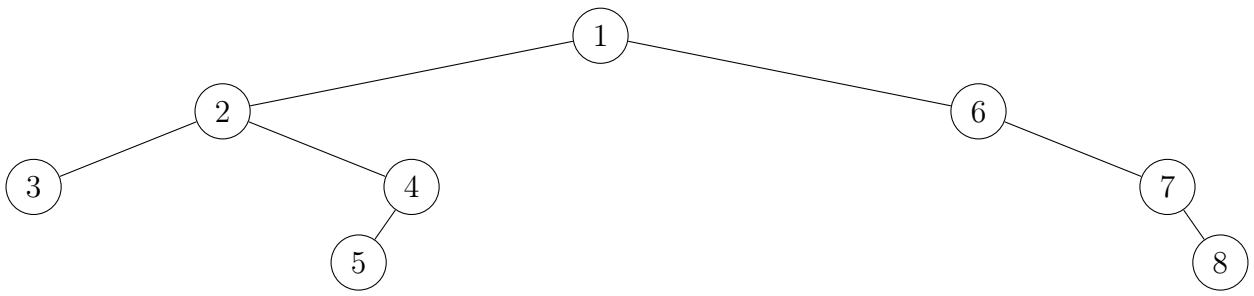
Q3 – Donner la définition de la fonction `hauteur : 'a btree -> int` qui calcule la hauteur d'un arbre (i.e. la distance entre la racine et la feuille la plus éloignée).

Q4 – En général, la *profondeur* d'un nœud dans un arbre est sa distance par rapport à racine.

Donnez la définition de la fonction `list_by_depth : 'a btree -> int -> 'a list` telle que `(list_by_depth bt n)` donne la liste de toutes les étiquettes de `bt` de profondeur `n`.

Pour tout `n`, on a que `(list_by_depth Empty n)` = `[]`.

Q5 – Donner la définition de la fonction `to_list : ('a btree) -> 'a list` qui calcule la liste préfixe des étiquettes présentes dans l'arbre: c'est-à-dire que l'étiquette en racine apparaîtra dans la liste avant les étiquettes du fils gauche, apparaissant elles-mêmes avant les étiquettes du fils droit.



Avec l'arbre ci-dessus, `to_list` donnera `[1;2;3;4;5;6;7;8]`

Exercice II : Variante

On peut choisir d'autres représentation pour les arbres binaires. par exemple, il lassant de devoir écrire `Node(Empty,x,Empty)` pour l'arbre qui ne contient que l'étiquette `x`. On appelle de tels arbres des *feuilles*.

On peut choisir de se donner un *constructeur* pour ce cas particulier et définir une variante du type `'a btree`:

```
type 'a ubtree =
    Empty2
  | Leaf of 'a
  | Node2 of 'a ubtree * 'a * 'a ubtree
```

Q1 – Définir la fonction `hauteur : 'a ubtree -> int` qui donne la hauteur de son argument.

Q2 – Définir la fonction `leaves : 'a ubtree -> 'a list` qui donne la liste des feuilles de son argument.

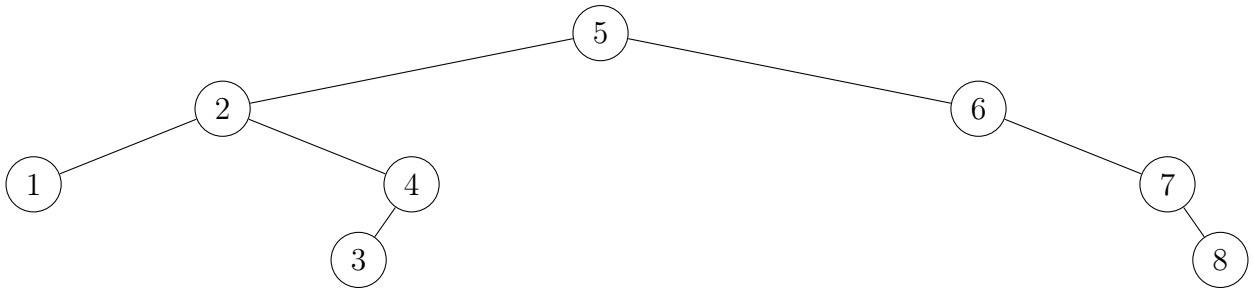
Q3 – Définir la fonction `bt_to_ubt : 'a btree -> 'a ubtree` qui transforme un arbre binaire de type `'a btree` en un arbre de type `'a ubtree`.

Exercice III : Arbres binaires de recherche

Un arbre binaire *de recherche* t est un arbre binaire dans lequel tout sous-arbre u de t est :

- soit vide
- soit de la forme `Node(`e`,`g`,`d`)` et alors : pour toute étiquette a de g, $a < e$ et pour toutes étiquettes b de d, $e \leq b$; et g et d sont des arbres binaires de recherche.

Exemple:



Q1 – Donnez une définition de la fonction `lt_btree : 'a btree -> 'a -> bool` telle que `(lt_btree bt x)` donne `true` si et seulement si toutes les étiquettes de `bt` sont inférieures (au sens strict) à `x`. On a `(lt_btree Empty x)=true`, pour tout `x`

Utiliser l'opérateur de comparaison polymorphe `<`.

Q2 – Donnez une définition de la fonction `ge_btree : 'a btree -> 'a -> bool` telle que `(lt_btree bt x)` donne `true` si et seulement si toutes les étiquettes de `bt` sont supérieures (au sens large) à `x`. On a `(ge_btree Empty x)=true`, pour tout `x`

Utiliser l'opérateur de comparaison polymorphe `>=`.

Q3 – Donner une définition de la fonction `is_abr : 'a btree -> bool` qui teste si un arbre est un ABR (arbre binaire de recherche).

Q4 – Donner une définition de la fonction `mem : 'a btree -> 'a -> bool` telle que `(mem bt x)` calcule `true` si l'élément `x` est présent dans `bt` (et `false` sinon). On fait l'hypothèse que `bt` est un arbre binaire de recherche.

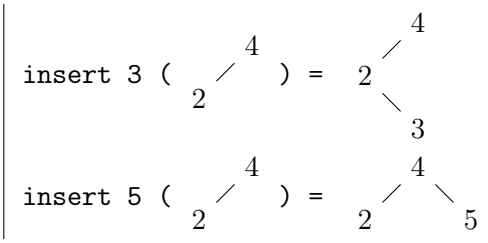
Travaux sur machines

Exercice IV : Tri par arbre binaire de recherche

On considère les arbres binaires de recherche définis à l'exercice 3, on va les utiliser pour réaliser une fonction de tri des éléments d'une liste.

Q1 – Donner une définition de la fonction `insert : 'a -> 'a btree -> 'a btree` qui ajoute une nouvelle étiquette dans un ABR (arbre binaire de recherche) en préservant la propriété *de recherche* de l'arbre.

Exemple



Remarque: l'ajout se fait *aux feuilles* de l'arbre.

Q2 – Donner 3 définitions de la fonction `from_list : 'a list -> 'a btree` prenant une liste en argument et construisant un arbre binaire de recherche contenant tous les éléments de la liste.

1. une définition récursive non terminale
2. une définition récursive terminale (avec fonction locale récursive terminale)
3. une définition utilisant l'itérateur `List.fold_left`

Q3 – Donner une définition de la fonction `to_list : 'a btree -> 'a list` qui donne la liste des élements d'un arbre binaire de recherche dans l'orde suivant: d'abord les étiquettes du fils gauche, celle de la racine et enfin, celles du fils droit.

Exemple

```
to_list
     Node(2,
      Node(1,Empty,Empty),
      Node(3,
       Empty)),
       Node(4,Empty,Empty) = [1;2;3;4]
```

Q4 – En déduire une fonction `tri : 'a list -> 'a list` qui trie la liste passée en argument.

Exercice V : Arbre de priorités

On imagine que l'on doit traiter une listes de *tâches* affectées chacune d'une priorité. Les priorités seront données par des entiers. Une *tâche* est un couple de type `(int*string)` où la deuxième composante est appelé le *nom de la tâche*.

Q1 – Définir la fonction
`insert_task : (int*string) -> (int*string) btree -> (int*string) btree`.

Elle est analogue à la fonction d'insertion de l'exercice précédent, mais utilise l'ordre suivant entre les couples: `(p1,t1) < (p2,t2)` si et seulement si `p1 < p2`.

Q2 – L'élément maximal dans un arbre binaire de recherche est celui qui se trouve à l'extrémité de la branche droite de l'arbre.

Définir la fonction `take_max : (int*string) btree -> string` qui renvoit le nom de la tâche de priorité maximale. La fonction déclenche l'exception `Invalid_argument` si l'arbre est vide.

Q3 – Définir la fonction `remove_max : (int*string) btree -> (int*string) btree` telle que `(remove_max bt)` renvoie l'arbre binaire de recherche obtenu en retirant de `bt` sa tâche de priorité maximale. La fonction déclenche l'exception `Invalid_argument` si l'arbre est vide.

Attention: la fonction `remove_max` ne retire que l'étiquette: `(remove_max (Node(bt1, (p,t), Empty)))` donne `bt1`.

Q4 – On veut ici faire les deux opérations précédentes «en même temps»; c'est-à-dire en ne parcourant qu'une seule fois la structure de l'arbre. Pour cela on calcule le couple formé de la tâche de priorité maximale et de l'arbre privé de cette tâche.

Définir la fonction `take_and_remove_max : (int*string) btree -> (string * (int*string) btree)` qui donne le nom de la tâche de priorité maximale et l'arbre privé de cette tâche.

Q5 – Maintenant, on veut utiliser la structure d'arbre de tâches (rangées selon leur priorité) pour exécuter une suite *d'actions*. Les actions sont de deux sortes:

1. Soit exécuter la tâche de priorité maximale stockée dans un arbre de tâche et la retirer de l'arbre.
2. Soit ajouter une nouvelle tâche dans l'arbre.

Les actions sont représentées par le type

```
type action =
  Pop
| Push of (int*string)
```

- l'action `Push (p,t)` consiste à mettre la tâche `(p,t)` en attente dans un arbre binaire de recherche de tâche;
- l'action `Pop` consiste à «exécuter» la tâche de priorité maximale de l'arbre et à la retirer de celui-ci.

On simule l'exécution d'une liste d'actions en définissant une fonction qui produit la liste des noms des tâches exécutées à partir d'une liste d'actions et d'un arbre de tâches supposées rangées selon leur priorité.

Définir la fonction `exec : action list -> (int*string) btree -> string list` telle que `(exec acts bt)` donne la liste des noms de tâches exécutées en suivant la liste d'actions donnée.

Pour les tests, on pourra démarrer avec un arbre de tâche vide et faire l'hypothèse que la première action de la liste n'est pas `Pop`.