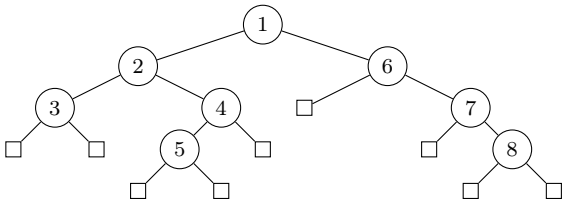
Feuille d'exercices n°4

Exercice I : Arbres binaires

Le type des arbres binaires est défini par

```
type 'a btree =
  Empty
| Node of ('a * 'a btree * 'a btree)
```

Exemple:




```
Node(1,
     Node(2, Node(3, Empty, Empty),
             Node(4,
                  Node(5, Empty, Empty),
                  Empty)),
     Node(6, Empty,
             Node(7, Empty,
                  Node(8, Empty, Empty))))
```

La *profondeur* d'u noeud est sa distance de la racine mesurée en nombre d'arêtes qui l'en sépare.

Dans notre exemple:

- la profondeur de (1) est 0;
- la profondeur de (2) est 1;
- la profondeur de (8) et de (5) est 3.

Q1 – Donner la définition de la fonction de signature

```
hauteur (bt:a btree) : int
```

qui calcule la hauteur d'un arbre (i.e. la plus grande profondeur).

La hauteur de l'arbre vide est 0; la hauteur de notre exemple est 4.

Q2 – Donnez la définition de la fonction de signature

```
list_by_depth (bt:'a btree) (n:int) : 'a list
```

qui donne la liste de toutes les étiquettes de `bt` de profondeur `n`.

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

Pour notre exemple, si `n`=2, on a la liste `[3;4;7]`.

Q3 – Donner la définition de la fonction de signature

```
to_list (bt:'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 de signature

```
taille (ubt:'a ubtree) : int
```

qui donne la taille de `bt`.

On considère que `(taille (Leaf x))`=1.

Q2 – Définir la fonction

```
hauteur (ubt:'a ubtree) : int
```

qui donne la hauteur de l'arbre `ubt`.

On considère que `(hauteur (Leaf x))`=1.

Q3 – Définir la fonction

```
leaves (ubt:'a ubtree) : 'a list
```

qui donne la liste des étiquette des feuilles de `ubt`.

Q4 – 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`.

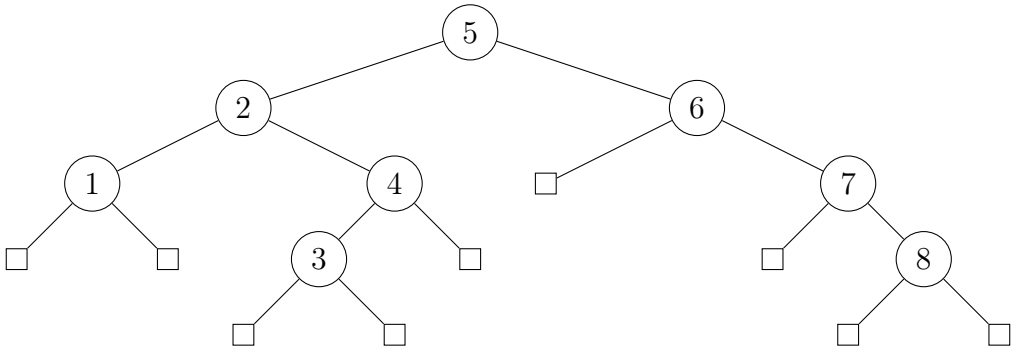
Dans `'a btree` une «feuille» est un nœud dont les deux sous arbres sont vides.

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 (bt:'a btree) (x:'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 (bt:'a btree) (x:'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 (bt:'a btree) : bool
```

qui teste si un arbre est un arbre binaire de recherche.

Q4 – Donner une définition de la fonction

```
mem (bt'a btree) (x:'a) : bool
```

telle que `(mem bt x)` vaut `true` si l'élément `x` est présent dans `bt` (et `false` sinon).

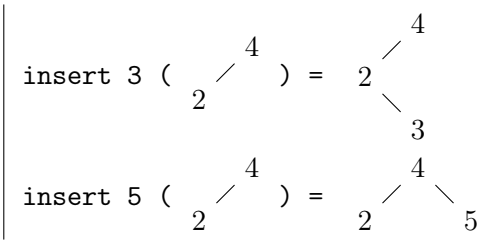
On fait l'hypothèse que `bt` est un arbre binaire de recherche et dans ce cas, il faut utiliser l'ordre des éléments pour optimiser la recherche.

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 (x:a) (bt:'a btree) : 'a btree` qui ajoute une nouvelle étiquette dans un 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 (xs:'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 (bt:'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))))
```

donne `[1;2;3;4]`

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