12 Exercices sur les arbres

Question 1

En utilisant la représentation ①, dessiner les arbres suivants :
arbre1 : `'(1 (2 3 (4 5 6)))`,
arbre2 : `'(1 (2 3 (4 5 6)) (7 (8 9 10)))`,
arbre3 : `'(1 (2 3))`,
arbre4 : `'(1 (2 (3 4) (4 5)))`,
arbre5 : `'(1 (2 3 (4 5 6)) (7 8))`.

Question 2

En utilisant l'interface `bt1`, définir une fonction `nb-nodes` permettant de connaitre le nombre de nœuds d'un arbre.
Appeler `(nb-nodes arbre1)` retourne 6.
Appeler `(nb-nodes arbre2)` retourne 10.
(*fonctions utiles* : `leaf?`, `has-R-subtree?`, `L-subtree`, `R-subtree`)

Question 3

En utilisant l'interface `bt1`, définir une fonction `h` permettant de connaitre la hauteur d'un arbre.
Appeler `(h arbre1)` retourne 3.
Appeler `(h arbre3)` retourne 2.
(*fonctions utiles* : `leaf?`, `max`, `has-R-subtree`, `L-subtree`, `R-subtree`)

Question 4

En utilisant l'interface `bt1`, définir une fonction `nb-leaves` permettant de connaitre le nombre de feuilles d'un arbre.
Appeler `(nb-leaves arbre1)` retourne 3.
Appeler `(nb-leaves arbre2)` retourne 5.
(*fonctions utiles* : `leaf?`, `has-R-subtree`, `L-subtree`, `R-subtree`)

Question 5

En utilisant l'interface `bt1`, définir une fonction `nbn-at-d` qui retourne la largeur d'un arbre à la profondeur d.
Appeler `(nbn-at-d arbre1 2)` retourne 2.
Appeler `(nbn-at-d arbre1 3)` retourne 2.
(*fonctions utiles* : `leaf?`, `has-R-subtree?`, `L-subtree`, `R-subtree`)

Question 6

En utilisant l'interface `bt1`, définir une fonction `nbn-per-d` qui retourne la largeur max d'un arbre (on appelle largeur max, le plus grand nombre de nœuds à la même profondeur).
Appeler `(nbn-per-d arbre1)` retourne `'(1 1 2 2)`.
Appeler `(nbn-per-d arbre2)` retourne `'(1 2 3 4)`.
(*fonctions utiles* : `cons`, `nbn-at-d`)

Question 7

En utilisant l'interface `bt1`, définir une fonction `width` qui retourne la largeur max d'un arbre (on appelle largeur max, le plus grand nombre de nœuds à la même profondeur).
Appeler `(width arbre1)` retourne 2.
Appeler `(width arbre2)` retourne 4.
(*fonctions utiles* : `nbn-per-d`, `length`, `max`, `first`, `rest`)

Question 8

En utilisant l'interface `bt1`, définir un prédicat `isst-root?` qui retourne vrai si la racine de l'arbre est supérieure à tous les éléments du sous-arbre gauche et inférieure à tous les éléments du sous-arbre droit.
Appeler `(isst-root? '(2 1 4))` retourne `#t`.
Appeler `(isst-root? '(1 2 3))` retourne `#f`.
Appeler `(isst-root? '(4 (2 1 3) (6 5)))` retourne `#t`.
(*fonctions utiles* : `leaf?`, `root`, `chk?`, `flatten`, `has-R-subtree?`, `L-subtree`, `R-subtree`)

Question 9

En utilisant l'interface `bt1`, définir un prédicat `isst?` qui retourne vrai si l'arbre est un arbre de recherche.
Appeler `(isst? '(4 (2 1 3) (6 5)))` retourne `#t`.
Appeler `(isst? '(4 (1 2 3) (6 5)))` retourne `#f`.
(*fonctions utiles* : `leaf?`, `isst-root?`, `has-R-subtree?`, `L-subtree`, `R-subtree`)

Question 10

Pour compléter l'interface `bt6`, définir le prédicat `leaf?`.
Appeler `(leaf? (leaf 1))` retourne `#t`.
Appeler `(leaf? (tree 1 (leaf 2) (leaf 3)))` retourne `#f`.
(*fonctions utiles* : `empty?`, `L-subtree`, `R-subtree`)

Question 11

Pour compléter l'interface `bt6`, définir le prédicat `has-LR-subtree?` qui retourne vrai si la racine courante possède un sous-arbre gauche et un sous-arbre droit.
Appeler `(has-LR-subtree? (leaf 1))` retourne `#f`.
Appeler `(has-LR-subtree? (tree 1 (leaf 2) (leaf 3)))` retourne `#t`.
(*fonctions utiles* : `has-L-subtree?`, `has-R-subtree?`)

Question 12

En utilisant l'interface `bt6`, définir une fonction `nb-nodes` permettant de connaitre le nombre de nœuds d'un arbre de recherche.
Appeler `(nb-nodes (tree 1 empty (leaf 2)))` retourne 2.
Appeler `(nb-nodes (tree 2 (leaf 1) empty))` retourne 2.
Appeler `(nb-nodes (addl empty '(7 6 8 3 2 5)))` retourne 6.
(*fonctions utiles* : `leaf?`, `has-LR-subtree?`, `has-L-subtree?`, `has-R-subtree?`, `L-subtree`, `R-subtree`)

Question 13

En utilisant l'interface `bt6`, définir une fonction `nb-leaves` permettant de connaitre le nombre de feuilles d'un arbre de recherche.
Appeler `(nb-leaves (tree 1 empty (leaf 2))` retourne 1.
Appeler `(nb-leaves (tree 2 (leaf 1) empty)` retourne 1.
Appeler `(nb-leaves (addl empty '(7 6 8 3 2 5)))` retourne 3.
(*fonctions utiles* : `leaf?`, `has-LR-subtree?`, `has-L-subtree?`, `has-R-subtree?`, `L-subtree`, `R-subtree`)

Question 14

En utilisant l'interface `bt6`, définir une fonction `h` permettant de connaitre la hauteur d'un arbre de recherche.
Appeler `(h (addl empty '(6 3 7 8 5 2)))` retourne 2.
Appeler `(h (addl empty '(7 6 8 3 2 5)))` retourne 3.
(*fonctions utiles* : `leaf?`, `has-LR-subtree?`, `has-L-subtree?`, `has-R-subtree`, `L-subtree`, `R-subtree`)

Question 15

Conformément à la représentation ❻, en utilisant la fonction `addl`, définir les arbres suivants :

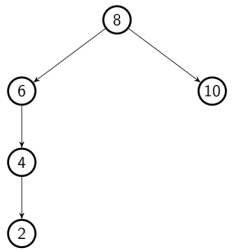

Fig. 15 – `arbre6`.

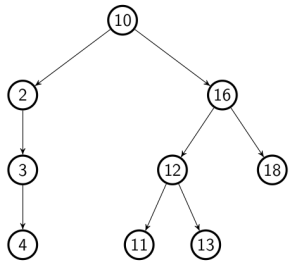

Fig. 16 – `arbre7`.

Question 16

Conformément à la représentation ⑥, dessiner les arbres suivants :
arbre8 : `(addl empty '(7 1 13 17 9))`
arbre9 : `(addl empty '(5 8 7 6 13 3 9 1 15))`
arbre10 : `(addl empty '(5 4 7 6 2 11 12 9 3 10 1 8 13))`

Question 17

En utilisant l'interface `bt6`, définir le prédicat `same?` permettant de savoir si deux arbres sont identiques dans leur construction.
Appeler `(same? (addl empty '(7 1 13 17 9)) (addl empty '(2 1 4 3 5)))` retourne `#t`.
Appeler `(same? (addl empty '(7 2 4 6)) (addl empty '(5 2 4 6)))` retourne `#f`.
(*fonctions utiles* : `leaf?`, `has-L-subtree?`, `has-R-subtree`, `L-subtree`, `R-subtree`)

Question 18

En utilisant l'interface `bt6`, écrire un prédicat `isin?` qui vérifie si une valeur v est dans un arbre.
Appeler `(isin? (addl empty '(6 3 7 8 5 2)) 2)` retourne `#t`.
Appeler `(isin? (addl empty '(6 3 7 8 5 2)) 1)` retourne `#f`.
(*fonctions utiles* : `root`, `leaf?`, `has-L-subtree?`, `has-R-subtree?`, `L-subtree`, `R-subtree`)

Question 19

En utilisant l'interface `bt6`, écrire une fonction `getl` qui récupère les éléments d'un arbre et les met dans une liste.
Il n'est pas nécessaire de garantir `L = (getl (add-l empty L))`.
Il suffit de garantir `T = (add-l empty (getl T))`.
Appeler `(addl empty '(6 3 7 8 5 2))` retourne
`'(6 (3 (2 () ()) (5 () ())) (7 () (8 () ())))`.
Appeler `(addl empty (getl (addl empty '(6 3 7 8 5 2))))` retourne
`'(6 (3 (2 () ()) (5 () ())) (7 () (8 () ())))`.
(*fonctions utiles* : `flatten`, `leaf?`, `has-L-subtree?`, `has-R-subtree?`, `L-subtree`, `R-subtree`)

Question 20

En utilisant l'interface `bt6`, écrire une fonction `max-tree` qui retourne le maximum des valeurs d'un arbre.
Appeler `(max-tree (addl empty '(6 3 7 8 5 2)))` retourne `8`.
Appeler `(max-tree (addl empty '(7 6 8 3 2 5)))` retourne `8`.
(*fonctions utiles* : `root`, `leaf?`, `has-R-subtree?`, `R-subtree`)

Question 21

En utilisant l'interface `bt6`, écrire une fonction `min-tree` qui retourne le minimum des valeurs d'un arbre.
Appeler `(min-tree (addl empty '(6 3 7 8 5 2)))` retourne 2.
Appeler `(min-tree (addl empty '(7 6 8 3 2 5)))` retourne 2.
(*fonctions utiles* : `root`, `leaf?`, `has-L-subtree?`, `L-subtree`)

Question 22

En utilisant l'interface `avltree`, écrire la fonction `rotate-L` qui réalise la rotation gauche d'un arbre.
Appeler `(rotate-L (addl empty '(4 3 11 8 12 13)))`.
(*fonctions utiles* : `root`, `depth`, `L-subtree`, `R-subtree`)

Question 23

En utilisant l'interface `avltree`, écrire la fonction `rotate-LR` qui réalise la rotation gauche droite d'un arbre.
Appeler `(rotate-LR (addl empty '(6 7 2 4 1 3 5)))`.
(*fonctions utiles* : `root`, `depth`, `L-subtree`, `R-subtree`)

Question 24

En utilisant l'interface `avltree`, écrire la fonction `rotate-RL` qui réalise la rotation droite gauche d'un arbre.
Appeler `(rotate-RL (addl empty '(2 1 6 7 4 3 5)))`.
(*fonctions utiles* : `root`, `depth`, `L-subtree`, `R-subtree`)

Question 25

En utilisant l'interface `avltree`, écrire la fonction `reequi` qui réalise les rotations nécessaires à l'équilibrage d'un arbre.