TP noté -

Les question notées ★ sont des questions bonus.

Arbre quaternaire

L'*arbre quaternaire* (*quadtree* en anglais) est une structure de données très simple pour l'indexation de points dans le plan. Il s'agit d'une structure de données hiérarchique construite par divisions récursives du plan en quatre rectangles disjoints. Plus précisemment, dans un arbre quaternaire, chaque sommet représente une partie du plan à indexer, la racine représentant le plan à indexer dans son intégralité. Nous représenterons les parties du plan par des *rectangles*. Chaque sommet est soit une *feuille* contenant zéro, un ou plusieurs points, soit un *sommet interne* ayant exactement 4 fils représentant les 4 rectangles obtenus en divisant la partie du plan associée à ce sommet en 2 le long de chaque axe. La Figure 1 donne un exemple de division récursive du plan.

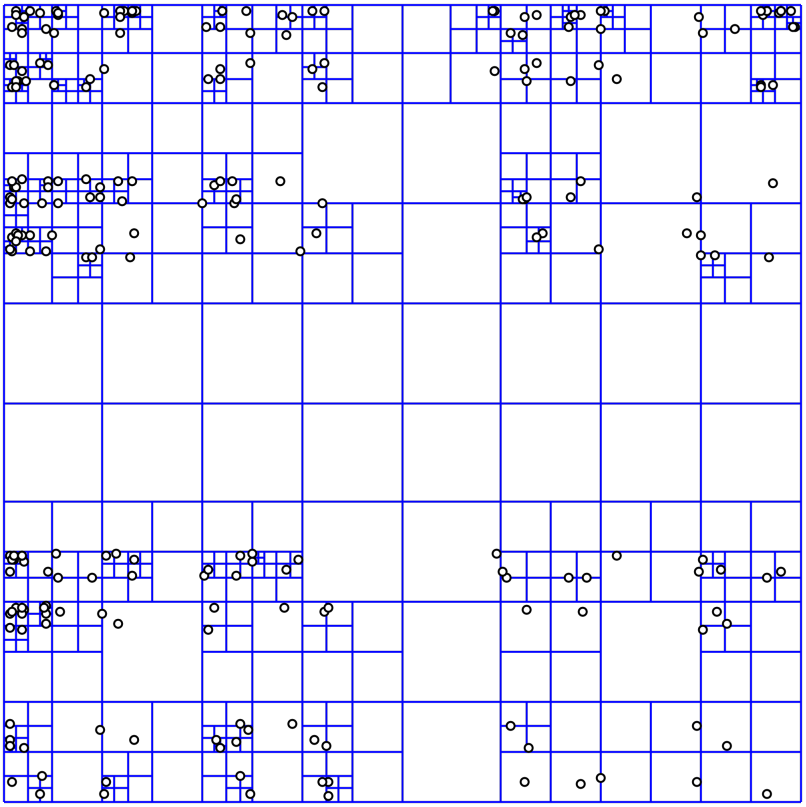

FIGURE 1 –

Nous avons évoqué le fait qu'une feuille contient zéro, un ou plusieurs points. Un arbre quaternaire est en effet paramétré par un entier positif indiquant le nombre maximum de points pouvant être stockés dans une feuille.

Pour faciliter la lecture, nous adopterons les conventions suivantes : pour un sommet interne, les 4 fils représenterons dans l'ordre (1) le rectangle supérieur gauche, (2) le rectangle supérieur

droit, (3) le rectangle inférieur droit, et (4) le rectangle inférieur gauche. Un exemple est donné Figure 2.

Dans ce TP nous allons progressivement développer une implémentation des arbres quaternaires en ocaml.

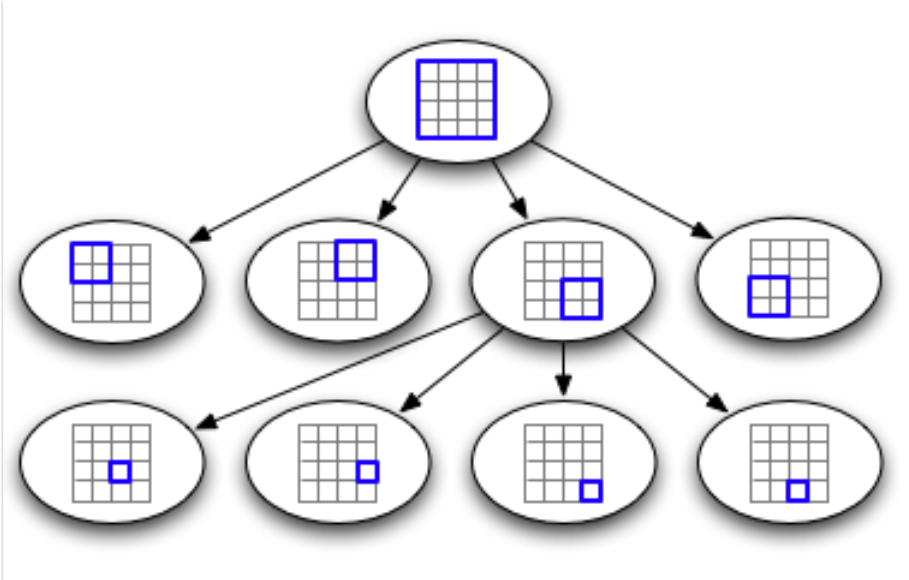

FIGURE 2 –

Rappels

Le synthétiseur de type ocaml produit le type le plus général. Par exemple

```
# type int_pair = int * int;;
type int_pair = int * int
# let p = 1, 2;;
val p : int * int = (1, 2)
# let f = fst;;
val f : 'a * 'b -> 'a = <fun>
```

On peut néanmoins utiliser la contrainte de type pour voir apparaître le nom désiré :

```
# let p : int_pair = 1, 2;;
val p : int_pair = (1, 2)
# let f : int_pair -> int = fst;;
val f : int_pair -> int = <fun>
```

Des points

Les points du plan seront représentés par le type `point` :

```
(* point type definition *)
type point = float * float;;
```

Question 1 Écrire la fonction `make_point` de type `float -> float -> point` qui prend en arguments les coordonnées x et y du point à construire.

```
# make_point 1.5 2.;;
- : point = (1.5, 2.)
# make_point (-1.0) 2.5;;
- : point = (-1., 2.5)
```

Question 2 Écrire les fonctions `point_x` et `point_y`, toutes deux de type `point -> float`, qui retournent la coordonnée x ou y du point passé en argument.

```
# let p = make_point 1. 2.;;
val p : point = (1., 2.)
# point_x p;;
- : float = 1.
# point_y p;;
- : float = 2.
```

Question 3 Un point $p_1 = (x_1, y_1)$ *domine* un point $p_2 = (x_2, y_2)$ si $x_1 \geq x_2$ et $y_1 \geq y_2$. Écrire la fonction `point_domination` de type `point -> point -> bool` qui retourne vrai si le premier point passé en argument domine le second.

```
# let p1 = make_point 0. 0. and p2 = make_point 0. 1. and p3 = make_point 1. 1.;;
val p1 : point = (0., 0.)
val p2 : point = (0., 1.)
val p3 : point = (1., 1.)
# point_domination p1 p1, point_domination p1 p2, point_domination p1 p3;;
- : bool * bool * bool = (true, false, false)
# point_domination p2 p1, point_domination p2 p2, point_domination p2 p3;;
- : bool * bool * bool = (true, true, false)
# point_domination p3 p1, point_domination p3 p2, point_domination p3 p3;;
- : bool * bool * bool = (true, true, true)
```

Des rectangles

Un rectangle, dont les cotés sont parallèles aux axes, est défini par 2 points du plan : son point inférieur gauche et son point supérieur droit. (Vous remarquerez que son point supérieur droit domine son point inférieur gauche !) Les rectangles seront donc représentés par le type `rectangle` :

```
(* rectangle type definition *)
type rectangle = point * point;;
```

Question 4 Écrire la fonction `make_rectangle` de type `point -> point -> rectangle` qui construit un rectangle à partir de son point inférieur gauche et de son point supérieur droit. Vous supposerez que le premier point est toujours dominé par le second point lors de l'appel de la fonction (il n'est donc pas nécessaire de faire le test dans votre fonction).

```
# let p1 = make_point 0. 0. and p2 = make_point 1. 2.;;
val p1 : point = (0., 0.)
val p2 : point = (1., 2.)
# make_rectangle p1 p2;;
- : rectangle = ((0., 0.), (1., 2.))
```

Question 5 Écrire les fonctions `rectangle_lower_left` et `rectangle_upper_right`, toutes deux de type `rectangle -> point`, qui retournent les points inférieur gauche et supérieur droit du rectangle passé en argument.

```
# let r = make_rectangle (make_point 0. 0.) (make_point 1. 2.);;
val r : rectangle = ((0., 0.), (1., 2.))
# rectangle_lower_left r;;
- : point = (0., 0.)
# rectangle_upper_right r;;
- : point = (1., 2.)
```

Question 6 Écrire les fonctions `rectangle_width` et `rectangle_height`, toutes deux de type `rectangle -> float`, qui retournent la largeur et la hauteur du rectangle passé en argument.

```
# let r = make_rectangle (make_point 0. 0.) (make_point 1. 2.);;
val r : rectangle = ((0., 0.), (1., 2.))
# rectangle_width r;;
- : float = 1.
# rectangle_height r;;
- : float = 2.
```

Question 7 Un point $p = (x, y)$ est contenu dans un rectangle de point inférieur gauche (x_ℓ, y_ℓ) et de point supérieur droit (x_r, y_r) si $x_\ell \leq x \leq x_r$ et $y_\ell \leq y \leq y_r$. Un point situé sur un coté du rectangle est donc contenu dans ce dernier. Écrire la fonction `rectangle_contains_point` de type `rectangle -> point -> bool` qui décide si un point est contenu dans un rectangle.

```
# let r = make_rectangle (make_point 0. 0.) (make_point 1. 1.);;
val r : rectangle = ((0., 0.), (1., 1.))
# let p1 = make_point 0. 0. and p2 = make_point 0.5 0.5 and p3 = make_point 0.5 1.5;;
val p1 : point = (0., 0.)
val p2 : point = (0.5, 0.5)
val p3 : point = (0.5, 1.5)
# rectangle_contains_point r p1;;
- : bool = true
# rectangle_contains_point r p2;;
- : bool = true
# rectangle_contains_point r p3;;
- : bool = false
```

Question 8 Écrire la fonction `rectangle_contained_points` de type `rectangle -> point list -> point list` qui, pour un rectangle et une liste de points donnés, retourne la liste des points contenus dand le rectangle (bonus : écrire la fonction sans récursion).

```
# let r = make_rectangle (make_point 0. 0.) (make_point 1. 1.);;
val r : rectangle = ((0., 0.), (1., 1.))
# let p1 = make_point 0. 0. and p2 = make_point 0.5 0.5 and p3 = make_point 0.5 1.5;;
val p1 : point = (0., 0.)
val p2 : point = (0.5, 0.5)
val p3 : point = (0.5, 1.5)
# rectangle_contained_points r [p1; p2; p3];;
- : point list = [(0., 0.); (0.5, 0.5)]
```

Des arbres quaternaires

Rappel : chaque sommet d'un arbre quaternaire est soit une *feuille* contenant zéro, un ou plusieurs points, soit un *sommet interne* ayant exactement 4 fils représentant les 4 rectangles obtenus en divisant la partie du plan associée à ce sommet en 2 le long de chaque axe. Pour faciliter l'écriture des fonctions nous associerons explicitement à chaque sommet de l'arbre quaternaire le rectangle associé. Les arbres quaternaires seront donc représentés par le type `quadtree` :

```
(* quadtree type definition *)
type quadtree = | Leaf of point list * rectangle
                | Node of quadtree * quadtree * quadtree * quadtree * rectangle;;
```

Question 9 Écrire la fonction `rectangle_split4` de type `rectangle -> rectangle * rectangle * rectangle * rectangle` qui prend en argument un rectangle et qui calcule un tuple représentant les 4 rectangles obtenus en divisant la partie du plan associée à ce rectangle en 2 le long de chaque axe.

```
# let r = make_rectangle (make_point 0. 0.) (make_point 1. 2.);;
val r : rectangle = ((0., 0.), (1., 2.))
# rectangle_split4 r;;
- : rectangle * rectangle * rectangle * rectangle =
(((0., 1.), (0.5, 2.)), ((0.5, 1.), (1., 2.)), ((0.5, 0.), (1., 1.)),
 ((0., 0.), (0.5, 1.)))
```

Question 10 Écrire les fonctions `smallest` et `largest`, toutes deux de type `'a list -> 'a`, qui retourne l'élement minimum et maximum d'une liste (bonus : écrire les fonctions sans récursion).

```
# let l = [2; 4; 6; 1; 8; 4; 3; 1];;
val l : int list = [2; 4; 6; 1; 8; 4; 3; 1]
# smallest l;;
- : int = 1
# largest l;;
- : int = 8
```

Question 11 Écrire la fonction `enclosing_rectangle` de type `point list -> rectangle` qui prend en argument une liste de points et qui calcule le plus petit rectangle contenant tous ces points.

```
# let p1 = make_point 0. 0. and p2 = make_point 0.5 0.5 and p3 = make_point 0.5 1.5;;
val p1 : point = (0., 0.)
val p2 : point = (0.5, 0.5)
val p3 : point = (0.5, 1.5)
# enclosing_rectangle [p1; p2; p3];;
- : rectangle = ((0., 0.), (1.5, 1.5))
```

Question 12 Écrire la fonction `quadtree_make` de type `point list -> int -> quadtree` qui prend en argument une liste de points et un entier n et qui calcule l'arbre quaternaire associé à ces points, chaque feuille de cet arbre contenant au plus n points.

```
# let p1 = make_point 0. 0. and p2 = make_point 0.5 0.5 and p3 = make_point 0.5 1.5;;
val p1 : point = (0., 0.)
val p2 : point = (0.5, 0.5)
val p3 : point = (0.5, 1.5)
# quadtree_make [p1; p2; p3] 1;;
- : quadtree =
Node (Leaf ([(0.5, 1.5)], ((0., 0.75), (0.75, 1.5))),
 Leaf ([], ((0.75, 0.75), (1.5, 1.5))), Leaf ([], ((0.75, 0.), (1.5, 0.75))),
 Node (Leaf ([], ((0., 0.375), (0.375, 0.75))),
  Leaf ([(0.5, 0.5)], ((0.375, 0.375), (0.75, 0.75))),
  Leaf ([], ((0.375, 0.), (0.75, 0.375))),
  Leaf ([(0., 0.)], ((0., 0.), (0.375, 0.375))), ((0., 0.), (0.75, 0.75))),
 ((0., 0.), (1.5, 1.5)))
```

Question 13 Écrire la fonction `quadtree_count` de type `quadtree -> int` qui retourne le nombre de points stockés dans un arbre quaternaire.

```
# let p1 = make_point 0. 0. and p2 = make_point 0.5 0.5 and p3 = make_point 0.5 1.5;;
val p1 : point = (0., 0.)
val p2 : point = (0.5, 0.5)
val p3 : point = (0.5, 1.5)
# quadtree_count (quadtree_make [p1; p2; p3] 1);;
- : int = 3
```

Question 14 Écrire la fonction `quadtree_signature` de type `quadtree -> int list` qui retourne la liste du nombre de points associé à chaque feuille. Les fils d'un sommet internet seront visités de gauche à droite.

```
# let p1 = make_point 0. 0. and p2 = make_point 0.5 0.5 and p3 = make_point 0.5 1.5;;
val p1 : point = (0., 0.)
val p2 : point = (0.5, 0.5)
val p3 : point = (0.5, 1.5)
# quadtree_signature (quadtree_make [p1; p2; p3] 1);;
- : int list = [1; 0; 0; 0; 1; 0; 1]
```

Rechercher des points à l'aide d'arbres quaternaires

Question 15 Écrire la fonction `quadtree_all_points` de type `quadtree -> point list` qui calcule la liste de tous les points contenus dans les feuilles d'un arbre quaternaire.

```
# let p1 = make_point 0. 0. and p2 = make_point 0.5 0.5 and p3 = make_point 0.5 1.5;;
val p1 : point = (0., 0.)
val p2 : point = (0.5, 0.5)
val p3 : point = (0.5, 1.5)
# quadtree_all_points (quadtree_make [p1; p2; p3] 1);;
- : point list = [(0.5, 1.5); (0.5, 0.5); (0., 0.)]
```

Question 16 On suppose données les fonctions

- `rectangle_contains_rectangle` de type `rectangle -> rectangle -> bool` qui décide si un rectangle est totalement inclu dans un autre,
- `rectangle_disjoint_rectangle` de type `rectangle -> rectangle -> bool` qui décide si deux rectangles donnés ont au moins un point d'intersection,
- `rectangle_intersection_rectangle` de type `rectangle -> rectangle -> rectangle` qui calcule le rectangle intersection de deux rectangles.

```
(* - : rectangle -> rectangle -> bool = <fun> *)
let rectangle_contains_rectangle r1 r2 =
  rectangle_contains_point r1 (rectangle_lower_left r2) &&
  rectangle_contains_point r1 (rectangle_upper_right r2);;

(* val rectangle_disjoint_rectangle : rectangle -> rectangle -> bool = <fun> *)
let rectangle_disjoint_rectangle r1 r2 =
  let r1_lower_left  = rectangle_lower_left r1  in
  let r1_upper_right = rectangle_upper_right r1 in
  let r2_lower_left  = rectangle_lower_left r2  in
  let r2_upper_right = rectangle_upper_right r2 in
  point_domination r2_lower_left r1_upper_right ||
  point_domination r1_lower_left r2_upper_right;;

(* - : rectangle -> rectangle -> rectangle = <fun> *)
let rectangle_intersection r1 r2 =
  let r1_ll = rectangle_lower_left r1  in
  let r1_ur = rectangle_upper_right r1 in
  let r2_ll = rectangle_lower_left r2  in
  let r2_ur = rectangle_upper_right r2 in
  let ll_x_max = largest [point_x r1_ll; point_x r2_ll] in
  let ll_y_max = largest [point_y r1_ll; point_y r2_ll] in
  let ur_x_min = smallest [point_x r1_ur; point_x r2_ur] in
  let ur_y_min = smallest [point_y r1_ur; point_y r2_ur] in
  let ll = make_point ll_x_max ll_y_max in
  let ur = make_point ur_x_min ur_y_min in
  make_rectangle ll ur;;
```

En utilisant ces fonctions, écrire la fonction `quadtree_rectangle_query` de type `rectangle -> quadtree -> point list` qui calcule – efficacement ! –la liste de points stockés dans un quadtree qui sont contenus dans un rectangle requête. Par efficace, nous entendons que :

- il n'est pas nécessaire de parcourir un arbre quaternaire si le rectangle associé à sa racine n'a aucun point commun avec le rectangle requête,
- tous les points contenus dans les feuilles d'un abre quaternaire sont nécessairement des solutions si le rectangle associé à la racine de cet arbre quaternaire est inclus dans le rectangle requête.

Des arbres quaternaires dynamiques ★

Question 17 ★ Écrire la fonction `quadtree_insert` de type `quadtree -> int -> point -> quadtree` qui, étant donné un arbre quaternaire contenant au plus n points dans chaque feuille, calcule le nouvel arbre quaternaire obtenu en insérant un point. (Ce nouvel arbre contient lui-aussi au plus n points dans chaque feuille.)

Question 18 ★ Écrire la fonction `quadtree_delete` de type `quadtree -> point -> quadtree` qui calcule le nouvel arbre quaternaire obtenu en supprimant un point donné. (Ce point peut exister ou non dans l'arbre quaternaire.)

Annexe

```
# let rec range i j = if i > j then [] else i::(range (i+1) j);;
val range : int -> int -> int list = <fun>
# let cartesian l l' =
  List.concat (List.map (fun e -> List.map (fun e' -> (e, e')) l') l);;
val cartesian : 'a list -> 'b list -> ('a * 'b) list = <fun>
# let points m n =
  let xs = List.map float_of_int (range 0 m) and
      ys = List.map float_of_int (range 0 n) in
    List.map (fun (x, y) -> make_point x y ) (cartesian xs ys);;
val points : int -> int -> point list = <fun>
# let ps = points 4 4 (* quelques points pour nos tests *)
val ps : point list =
  [(0., 0.); (0., 1.); (0., 2.); (0., 3.); (0., 4.); (1., 0.); (1., 1.);
   (1., 2.); (1., 3.); (1., 4.); (2., 0.); (2., 1.); (2., 2.); (2., 3.);
   (2., 4.); (3., 0.); (3., 1.); (3., 2.); (3., 3.); (3., 4.); (4., 0.);
   (4., 1.); (4., 2.); (4., 3.); (4., 4.)]
# let qt = quadtree_make ps 2 (* l'arbre quaternaire pour nos tests *);;
val qt : quadtree =
  Node
   (Node
     (Node (Leaf ([(0., 4.)], ((0., 3.5), (0.5, 4.))),
       Leaf ([(1., 4.)], ((0.5, 3.5), (1., 4.))),
       Leaf ([(1., 3.)], ((0.5, 3.), (1., 3.5))),
       Leaf ([(0., 3.)], ((0., 3.), (0.5, 3.5))), ((0., 3.), (1., 4.))),
     Node (Leaf ([(1., 4.)], ((1., 3.5), (1.5, 4.))),
      Leaf ([(2., 4.)], ((1.5, 3.5), (2., 4.))),
      Leaf ([(2., 3.)], ((1.5, 3.), (2., 3.5))),
      Leaf ([(1., 3.)], ((1., 3.), (1.5, 3.5))), ((1., 3.), (2., 4.))),
     Node (Leaf ([(1., 3.)], ((1., 2.5), (1.5, 3.))),
      Leaf ([(2., 3.)], ((1.5, 2.5), (2., 3.))),
      Leaf ([(2., 2.)], ((1.5, 2.), (2., 2.5))),
      Leaf ([(1., 2.)], ((1., 2.), (1.5, 2.5))), ((1., 2.), (2., 3.))),
     Node (Leaf ([(0., 3.)], ((0., 2.5), (0.5, 3.))),
      Leaf ([(1., 3.)], ((0.5, 2.5), (1., 3.))),
      Leaf ([(1., 2.)], ((0.5, 2.), (1., 2.5))),
      Leaf ([(0., 2.)], ((0., 2.), (0.5, 2.5))), ((0., 2.), (...))),
     ...),
   ...)
# let points_in = let p1 = make_point 0. 0. and p2 = make_point 2. 3. in
                  let query = make_rectangle p1 p2                    in
                  quadtree_rectangle_query query qt;;
val points_in : point list =
  [(1., 3.); (0., 3.); (2., 3.); (2., 2.); (1., 2.); (0., 2.); (2., 1.);
   (2., 0.); (1., 1.); (0., 1.); (1., 0.); (0., 0.)]
# let points_in = let p1 = make_point 3. 3. and p1 = make_point 5. 5. in
                  let query = make_rectangle p1 p2                    in
                  quadtree_rectangle_query query qt;;
val points_in : point list = [(3., 4.); (3., 3.); (4., 4.); (4., 3.)]
```

```
# let points_in = let p1 = make_point 2. (-1.) and p2 = make_point 3. 5. in
                  let query = make_rectangle p1 p2                        in
                  quadtree_rectangle_query query qt;;
val points_in : point list =
  [(2., 4.); (2., 3.); (2., 2.); (3., 4.); (3., 3.); (3., 2.); (3., 1.);
   (2., 1.); (3., 0.); (2., 0.)]
# let points_in = let p1 = make_point 3.75 3.75 and p2 = make_point 6. 6. in
                  let query = make_rectangle p1 p2                        in
                  quadtree_rectangle_query query qt;;
val points_in : point list = [(4., 4.)]
# let points_in = let p1 = make_point 4.75 4.75 and p2 = make_point 6. 6. in
                  let query = make_rectangle p1 p2                        in
                  quadtree_rectangle_query query qt;;
val points_in : point list = []
```