Ben KABONGO BUZANGU – *21911598* **2020-2021**

L2 INFORMATIQUE

DM de ALGORITHMIQUE

On définit les structures et les fonctions suivantes, que l'on utilisera tout au long de ce travail.

Le type ***Point*** représente un point géométrique de coordonnées (x, y).

```
structure _Point {
        x : entier
        y : entier
}
type Point = pointeur sur _Point ;
```

Afin d'alléger le code, on pourra souvent considérer les objets de type *Point* comme des tableaux d'entiers à deux éléments dont l'indice 0 référencera l'abscisse x du point et l'indice 1 l'ordonnée y du point.

On pourrait considérer qu'il existe des fonctions qui permettent de passer d'une représentation à l'autre.

Ainsi, pour un point p, les valeurs *p->x* et *p[0]* sont considérées comme égales. Idem pour les valeurs *p->y* et *p[1]*.

Dans les lignes qui vont suivre, on emploiera souvent une représentation, souvent l'autre.

La fonction ***point*** crée un objet de type Point.

```
point(x : entier, y : entier) : Point
        tmp : Point ; tmp = Nouveau(_Point)
        tmp->x = x ; tmp-> y = y
        retourner tmp
```

La fonction ***dist*** calcule la distance entre deux points.

```
dist(p : Point, q : Point) : réel
        retourner ((q->y – p->y) ^ 2 + (q->x -p->x) ^ 2) ^ 0.5)
```

Le type ***Zone*** représente une zone rectangulaire du plan dont on donne le point inférieur gauche et le point supérieur droit.

```
structure _Zone {
        ig : Point
        sd : Point
}
type Zone = pointeur sur _Zone ;
```

La fonction ***zone*** crée une zone.

```
zone(ig : Point, sd : Point) : Zone
        tmp : Zone ; tmp = Nouveau(_Zone)
        tmp->ig = ig
        tmp->sd = sd
        retourner tmp
```

La fonction ***estUnPointDeLaZone*** détermine si un point donné est à l'intérieur d'une zone rectangulaire.

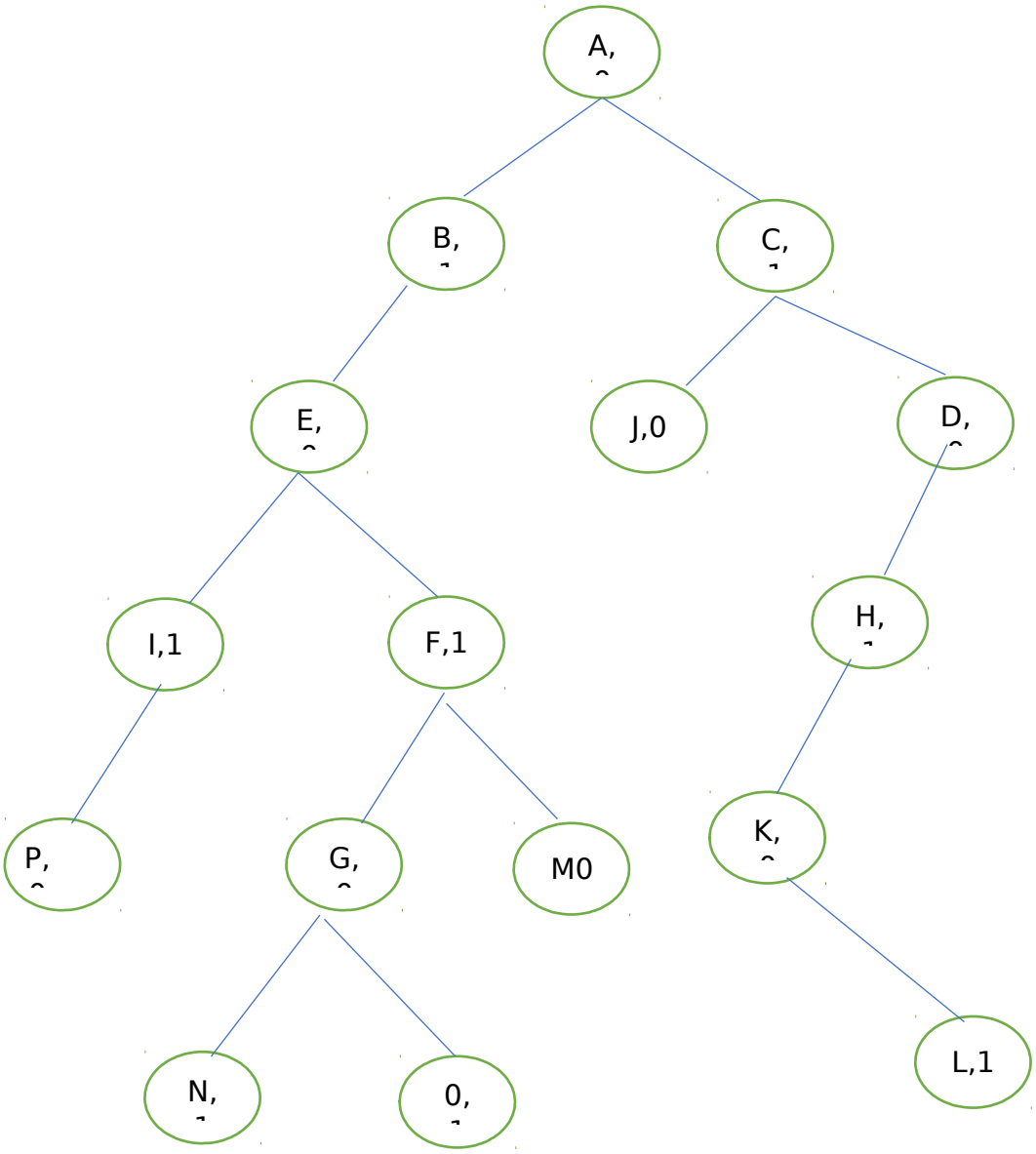
```
estUnPointDeLaZone(point: Point, zone: Zone) : booléen
   retourner ((zone->ig->x < point->x et point->x <= zone->sd->x) et
         (zone->ig->y < point->y et point->y <= zone->sd->y))
```

La structure ***NoeudZ*** représente un nœud de l'arbre binaire de recherche que nous allons construire.

```
structure NoeudZ {
        valeur : Point
        cs : entier
        gauche : pointeur sur NoeudZ
        droit : pointeur sur NoeudZ
}
type ABRZ = pointeur sur NoeudZ;
```

Question 1

Représentation de l'arbre



Représentation du plan

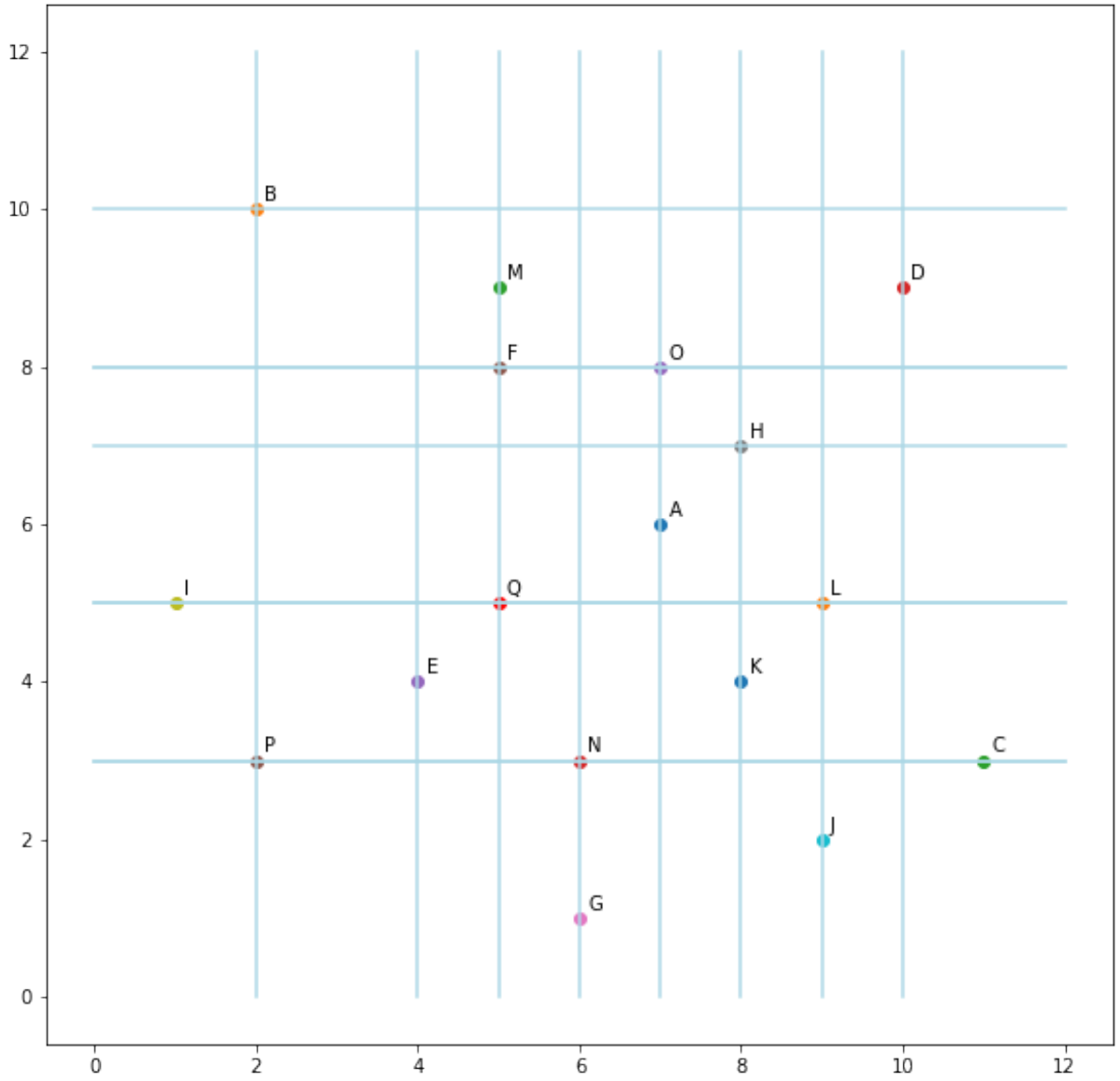
Nous avons commencé par représenter les points A, B, C, D, E, F, G, H, I, J, K, L, M, O, P et Q dans le repère *P* orthonormé.

Pour les 16 premiers, qui sont les points des nœuds de l'arbre que nous allons créer, lors de leur insertion dans l'arbre, on obtient la valeur de la coordonnée de séparation (cs), qui est donnée par la profondeur du nœud dans l'arbre. Si la profondeur est paire, cs=0 ; sinon cs=1.

Ainsi, on a pu calculer les valeurs des coordonnées de séparation pour tous les 16 points de l'arbre.

On obtient donc :

Point	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Cs	0	1	1	0	0	1	0	1	1	0	0	1	0	1	1	0



Si la coordonnée de séparation vaut 0 pour un point M = (a, b), la coordonnée à considérer est l'abscisse. On trace dans le plan une droite verticale passant par le point. L'équation de cette droite est donc **x = a**.

Si par contre, la coordonnée de séparation du point vaut 1, la coordonnée à considérer est l'ordonnée. On trace dans le plan une droite horizontale passant par le point. L'équation de cette droite est donc **y = b**.

Question 2

1. La fonction *creerArbre* :

```
creerArbre(p: Point, g: ABRZ, d: ABRZ, cs: entier) : ABRZ
        A : ABRZ ; A = Nouveau(NoeudZ)
        A->valeur = p
        A->cs = cs
        A->gauche = g
        A->droit = d
```

2. La fonction *inserer :*

```
inserer(A: ABRZ, p: Point): ABRZ
   si A = None alors
      retourner creerArbre(p, None, None, 0)
   si A->valeur->x = p->x et A->valeur->y = p->y alors
      afficher "Ce point existe déjà"
   sinon
      si p[A->cs] <= A->valeur[A->cs] alors
         si A->gauche = None alors
            A->gauche = creerArbre(p, None, None, (A->cs+1) mod 2)
         sinon
            A->gauche = inserer(A->gauche, p)
      sinon
         si A->droit = None alors
            A->droit = creerArbre(p, None, None, (A->cs+1) mod 2)
         sinon
            A->droit = inserer(A->droit, p)
   retourner A
```

Question 3

1. La fonction *recherche* :

```
recherche(A: ABRZ, p: Point): ABRZ
   si A != None et (A->valeur->x != p->x ou A->valeur->y != p->y) alors
      si p[A->cs] <= A->valeur[A->cs] alors          // recherche à gauche
         retourner recherche(A->gauche, p)
      sinon    // recherche à droite
         retourner recherche(A->droit, p)
   retourner A
```

2. Le nombre maximum de nœuds visités est égal à la hauteur de l'arbre.

Donc C(A) = h(A), avec h(A) : hauteur de A.

Question 4

1. La fonction *minX* :

```
minX(A: ABRZ): ABRZ
   si A->gauche = None et A->droit = None alors
      retourner A
   si A->cs = 0 alors
      si A->gauche = None alors retourner A
      sinon retourner minX(A->gauche)
   sinon
      res : ABRZ ; res = None
      si A->gauche = None alors res = minX(A->droit)
      sinon si A->droit = None alors res = minX(A->gauche)
      sinon
         gauche : ABRZ ; gauche = minX(A->gauche)
         droit : ABRZ ; droit = minX(A->droit)
       si gauche->valeur[0] < droit->valeur[0] alors res = gauche
       sinon res = droit
      si res->valeur[0] < A->valeur[0] alors retourner res
      sinon retourner A
```

2. Les nœuds visités par le parcours sont A, B, E, I, P.

3. La fonction *min* :

```
min(A: ABRZ, cs: entier) : ABRZ
   si A->gauche = None et A->droit = None alors
      retourner A
   si A->cs = cs alors
```

Question 5

1. La fonction *dansDroiteV* :

```
dansDroiteV(A: ABRZ, x: entier)
   si A != None alors
      si A->valeur->x = x alors
        afficher "La droite verticale x = " + x +" passe par le point (" + A-
>valeur->x + "," +
                 A->valeur->y +") de l'arbre"
      si A->cs = 0 alors
         si A->valeur->x >= x alors
            dansDroiteV(A->gauche, x)
         sinon
            dansDroiteV(A->droit, x)
      sinon
         dansDroiteV(A->gauche, x)
         dansDroiteV(A->droit, x)
```

2. La droite verticale x = 9 passe par les points J = (9, 2) et L = (9, 5) de l'arbre.

Les nœuds visités par la fonction sont : A, C, J, D, H, K, L.

3. La fonction *dansDroite* :

```
dansDroite(A: ABRZ, p: entier, cs: entier):
   si A != None alors
      si A->valeur[cs] = p alors
        si cs = 0 alors
                afficher "La droite verticale x = " + p +" passe par le point (" + A-
      >valeur->x
                + "," + A->valeur->y +") de l'arbre"
        sinon afficher "La droite horizontale y=" + p +" passe par le point (" + A-
>valeur->x
                + "," + A->valeur->y +") de l'arbre"
      si A->cs = cs alors
         si A->valeur[cs] >= p alors dansDroite(A->gauche, p, cs)
         sinon dansDroite(A->droit, p, cs)
      sinon
         dansDroite(A->gauche, p, cs)
         dansDroite(A->droit, p, cs)
```

Question 6

La fonction *intersection* :

```
intersection(A: ABRZ, p: Point, q: Point)
   si A != None alors
      z : Zone ; z = zone(p, q)
      si estUnPointDeLaZone(A->valeur, z) alors
         afficher "Le point {" + A->valeur->x + "," + A->valeur->y "} est un
point de la zone"
        intersection(A->gauche, p, q)
        intersection(A->droit, p, q)
      sinon
        si A->valeur[A->cs] <= z->ig->x alors intersection(A->droit, p, q)
        sinon intersection(A->gauche, p, q)
```

Question 7

1. La fonction *plusProche* :

Afin de définir cette fonction, nous avons défini une seconde fonction *_plusProche* qui prend en paramètres, dans cet ordre, le sous-arbre couramment visité (A), le sous-arbre dont le point

est le plus proche du point à approcher (*currentProche*), le point à approcher (*p*), la distance minimale courante (*d*), les zones définies par le nœud par le nœud père du sous-arbre visité (*zonesParent*).

Par exemple, pour *A->cs = 0*, on sait que *A* découpe le plan en une zone gauche *zg* et une zone droite *zd*. Si on a *A->gauche = B*, alors dans l'appel de la fonction, pour *B*, on aura *_plusproche(B, p, currentProche, currentDistance, zg)*. Et, si *A->droit = C*, alors l'appel de la fonction pour *C* sera *_plusproche(C, p, currentProche, currentDistance, zd)*. La même logique est à appliquer si *A->cs = 1* et qu'il s'agit de zone basse et zone haute.

Ce paramètre de plus est nécessaire pour éviter de parcourir à nouveau l'arbre de départ afin de retrouver les zones associées à chaque nœud. On a donc juste besoin de la zone parente à laquelle est associée le nœud afin de retrouver les zones qu'il crée

```
_plusproche(A: ABRZ, currentProche: ABRZ, p: Point, d: int, zoneParent: Zone) :
ABRZ
   si dist(A->valeur, p) < d alors
      currentProche = A
      d = dist(A->valeur, p)
   zone1, zone2 : Zone
   zone1 = zone(point(zoneParent->ig->x, zoneParent->ig->y),
                point(zoneParent->sd->x, zoneParent->sd->y))
   zone2 = zone(point(zoneParent->ig->x, zoneParent->ig->y),
                point(zoneParent->sd->x, zoneParent->sd->y))
   zone1->sd[A->cs] = A->valeur[A->cs]
   zone2->ig[A->cs] = A->valeur[A->cs]
   si estUnPointDeLaZone(p, zone1) et A->gauche != None alors
      retourner _plusproche(A->gauche, currentProche, p, d, zone1)
   sinon si estUnPointDeLaZone(p, zone2) et A->droit != None alors
      retourner _plusproche(A->droit, currentProche, p, d, zone2)
   retourner currentProche

plusproche(A: ABRZ, p: Point) :ABRZ
   retourner _plusproche(A, A, p, dist(A.valeur, p),
               zone(point(-inf, -inf), point(inf, inf)))
```

2. Les nœuds visités par cette fonction dans l'ordre sont : A, B, E, F, G, N.

Donc C(A) = h(A) + 1, avec h(A) : hauteur de A.

Le point de l'arbre le plus proche du point Q = (5, 5) est le point E = (4, 4). Ils sont à une distance de $\sqrt{2}$.