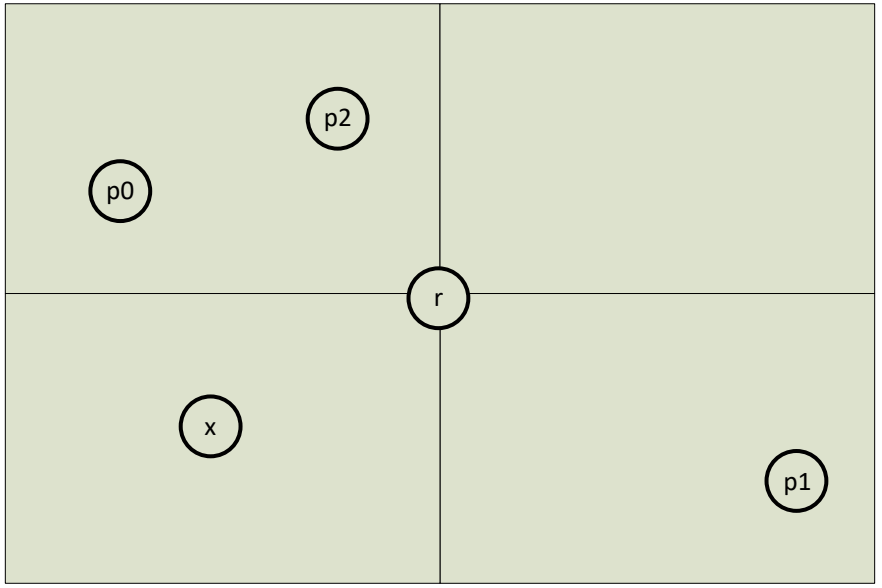
# Quad-Trees

# Quad-Trees

- Séparer l’espace en quatre sections autour d’un point central

# Quad-Trees

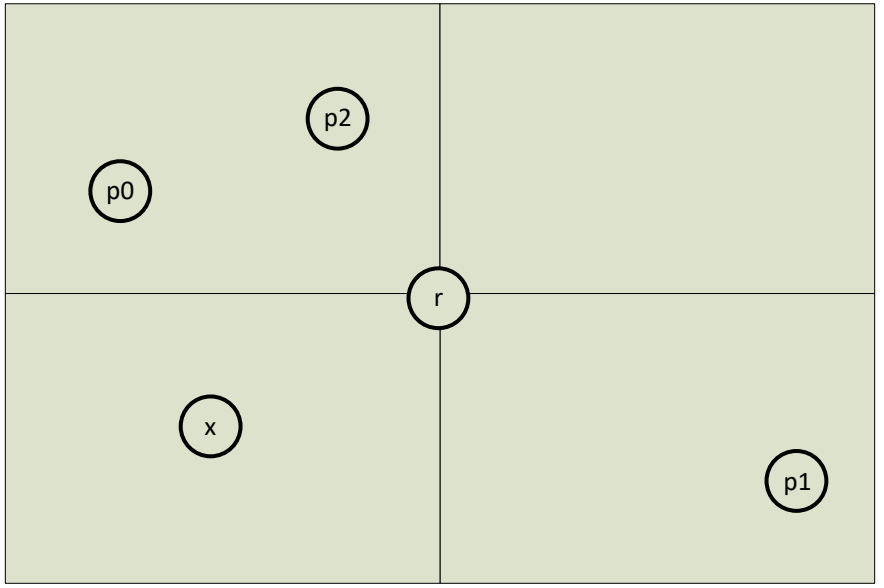
- Séparer l’espace en quatre sections autour d’un point central



# Quad-Trees

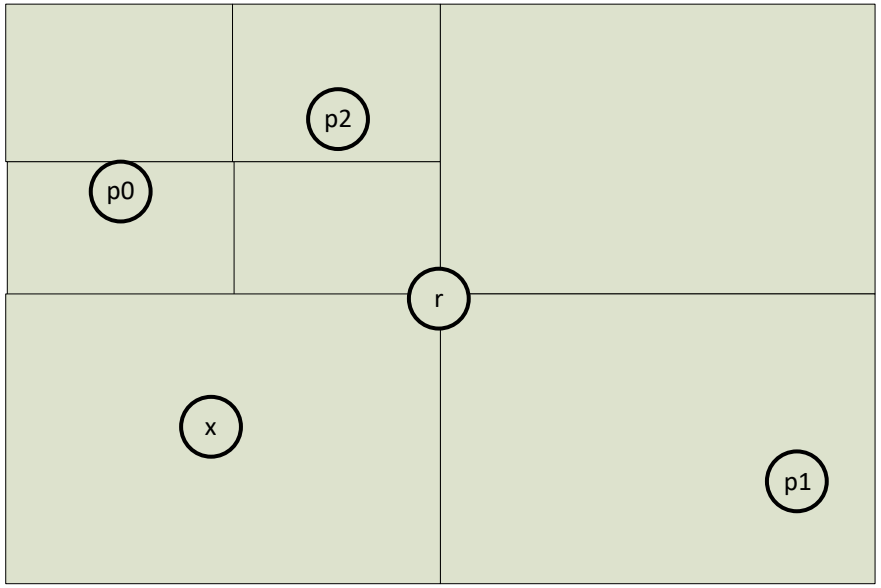
- Séparer l’espace en quatre sections autour d’un point central

Ici, r est la racine, et on veut savoir lequel des points p0, p1, p2, … est le plus près du point x



# Quad-Trees

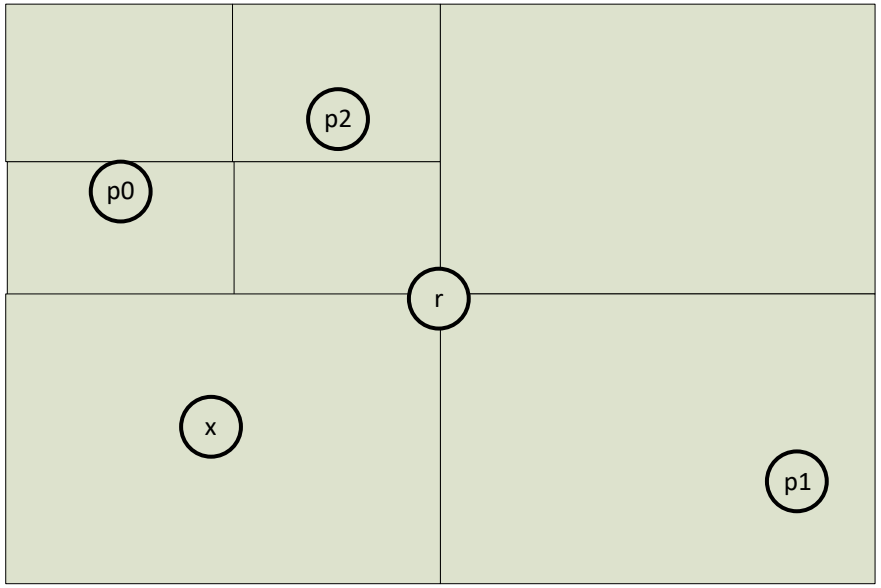
- Séparer l’espace en quatre sections autour d’un point central



# Quad-Trees

- Séparer l’espace en quatre sections autour d’un point central

Chaque quadrant peut être redécoupé en quadrants pour raffiner récursivement la recherche



# Quad-Trees

- Séparer l'espace en quatre sections autour d'un point central

```
struct Point {
    float x {}, y {};
    Point() = default;
    Point(float x, float y) : x{x}, y{y} {
    }
};
double distance(const Point &p0, const Point &p1) {
    return sqrt(pow(p0.x-p1.x,2) + pow(p0.y-p1.y, 2));
}
```

# Quad-Trees

- Séparer l'espace en quatre sections autour d'un point central

```
enum class quadrant { NE, NO, SO, SE };
quadrant ou_chercher(const Point &pt, const Point &x) {
    // 1 seulement si pt est à droite de x
    int horizontal = static_cast<int>(pt.x >= x.x);
    // 1 seulement si pt est à au-dessus de x
    int vertical = static_cast<int>(pt.y >= x.y);
    // les quadrants sont de 0 à 3
    return static_cast<quadrant>(
        2 * horizontal + vertical
    );
}
```

# Quad-Trees

```
struct quad_node {
   Point pt;
   size_t voisin[4]; // quatre quadrants voisins
};
using quad_tree = vector<quad_node>;
// ...
quad_tree zone = {
   { { 41.792f, -102.429f }, { 6, 20, 29, 11 } },
   { { 36.4843f, -78.9795f } , { 0, 0, 26, 28 } },
   { { 47.1305f, -99.3343f }, { 0, 0, 0, 0 } },
   // ... pages 132-133 des notes de cours
   { { 42.0087f, -70.8559f },  { 0, 0, 0, 0 } }
};
```

# Quad-Trees

- L’indice zéro est la racine
- On cherche le point *x* le plus près de { 41.8, -84 }

# Quad-Trees

- L’indice zéro est la racine
- On cherche le point *x* le plus près de { 41.8, -84 }
- Première approximation : quadrant 0 (NE) de l’indice 0
  - Il est à droite et en bas de *x*

# Quad-Trees

- L’indice zéro est la racine
- On cherche le point *x* le plus près de { 41.8, -84 }
- Première approximation : quadrant 0 (NE) de l’indice 0
  - Il est à droite et en bas de *x*
- De là (voir pages 132-133 des notes de cours), on trouve le point à l’indice 6 qui est { 42.7919, -77.6058 }, plus proche de *x*

# Quad-Trees

- L’indice zéro est la racine
- On cherche le point *x* le plus près de { 41.8, -84 }
- Première approximation : quadrant 0 (NE) de l’indice 0
  - Il est à droite et en bas de *x*
- De là (voir pages 132-133 des notes de cours), on trouve le point à l’indice 6 qui est { 42.7919, -77.6058 }, plus proche de *x*
- Prochaine approximation : quadrant 3 (SE) de l’indice 6 qui nous mène à l’indice 9
  - Encore mieux : { 42.5961, -83.1161 }

# Quad-Trees

- Au point à l’indice 9, trois quadrants sont à zéro
  - Rien à voir
- On explore la zone restante (indice 5)
  - Un peu mieux!
- Quand on remonte, on explore les autres quadrants au cas où
  - On n’explore pas tout; dès qu’on constate qu’on n’a pas mieux, on arrête d’explorer une branche

# Variantes

# Variantes

- Si on cherche en trois dimensions, un Oct-tree permet une représentation similaire

# Variantes

- Si on cherche en trois dimensions, un Oct-tree permet une représentation similaire
- Vos notes de cours (pages 133-134) parlent aussi de KD-Tree, ou arbre en K dimensions
  - L’exemple donné utilise deux zones plutôt que quatre
  - Les indices pairs permettent de trancher sur la base des abcisses
  - Les indices impairs permettent de trancher sur la base des ordonnées