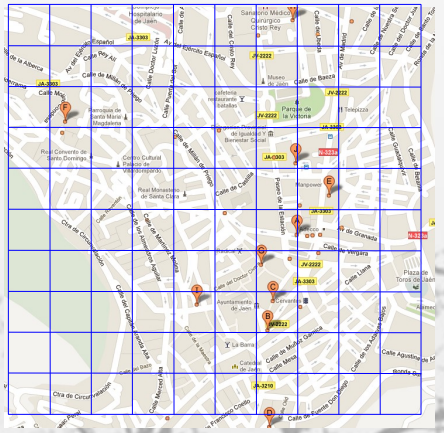
Lección 18: Quadtrees y range trees

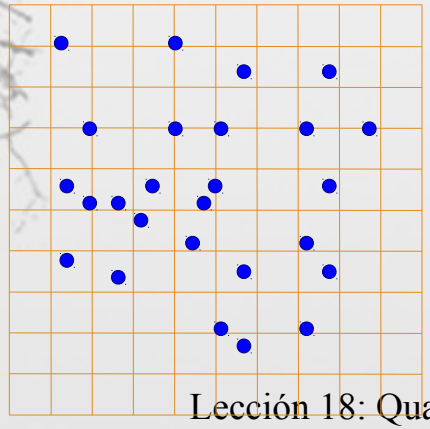
- Motivación
- Quadtrees
  - creación, inserción, localización
  - octrees
  - aplicaciones
- Range-trees
  - creación, búsqueda por rango
  - eficiencia
- Consideraciones finales

Motivación

Para la implementación de *GPService* se ha optado por una estructura de datos adaptativa, que permita más divisiones donde más puntos de interés existan

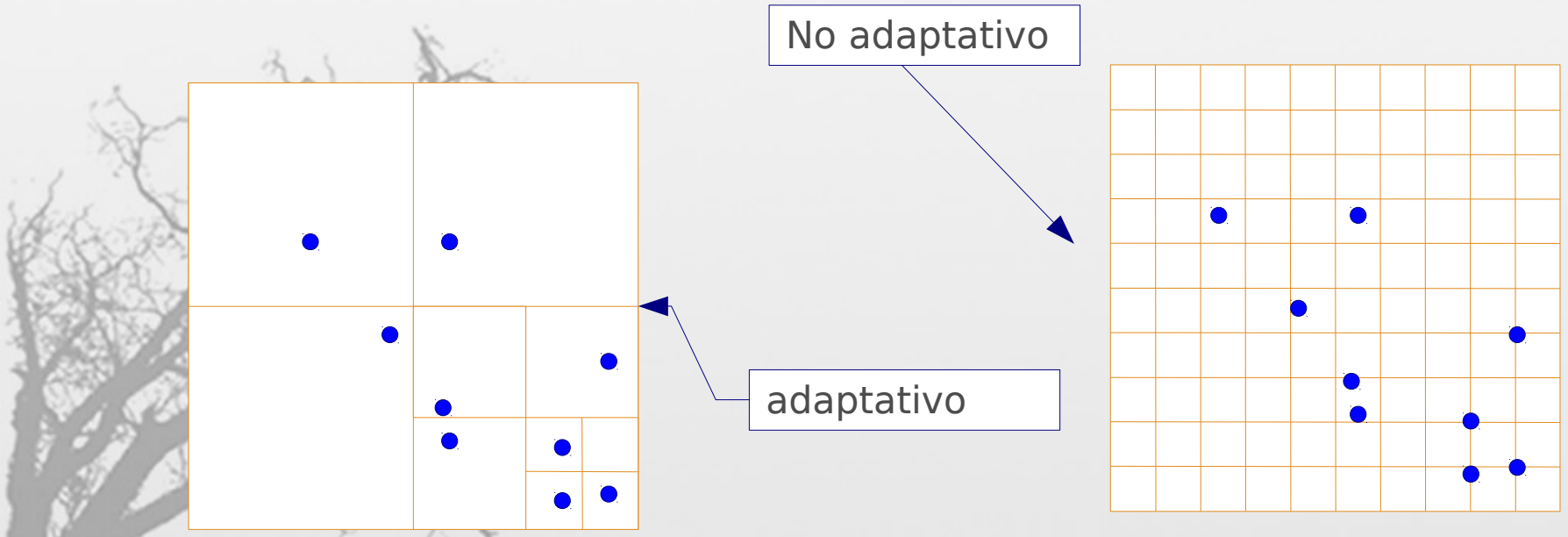


las mallas regulares
son EEDD no adaptativas



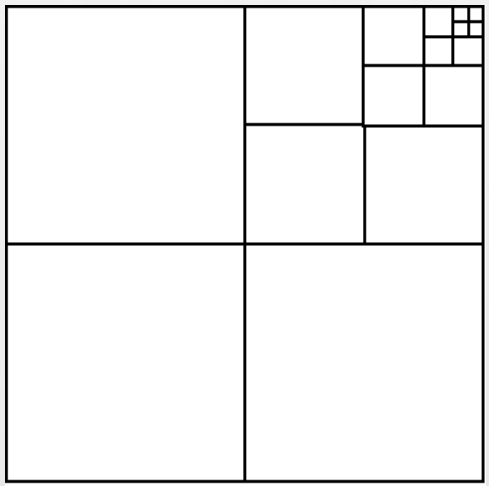
Motivación

Cuando los datos se distribuyen uniformemente por la malla regular ésta funciona de modo muy eficiente, pero la concentración de puntos en zonas específicas hace que se opte por otras EEDD adaptativas como los quadtrees.



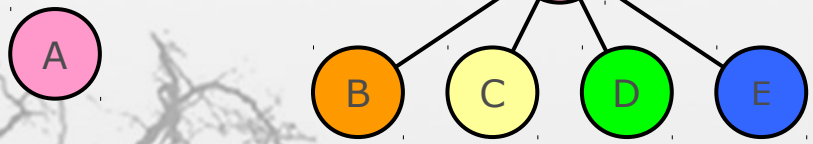
Quadtrees

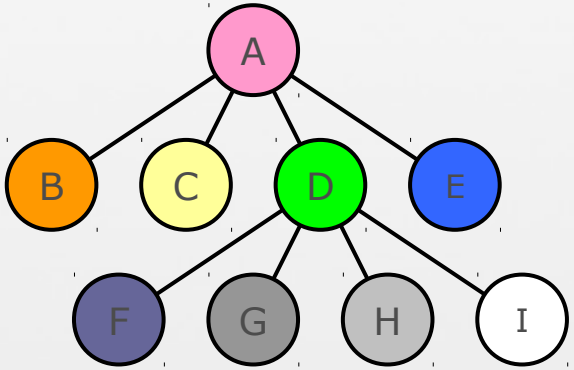
- Los **quadtrees** son EEDD jerárquicas que descomponen el plano de forma recursiva en cuatro cuadrantes disjuntos de igual tamaño
- Se implementan mediante árboles; el nodo raíz representa todo el plano objeto de estudio
- Cada nodo no hoja tiene 4 nodos hijos que representan una partición ortogonal de la misma forma que el nodo padre pero de menor tamaño
- Una región se vuelve a dividir si no cumple la condición de parada:
  - p.e: contiene muchos puntos

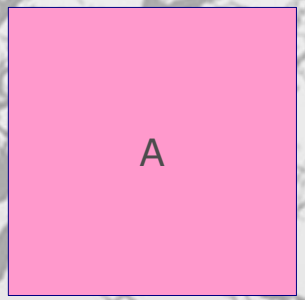


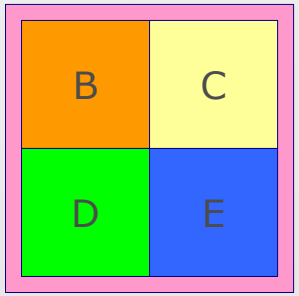
Quadtrees

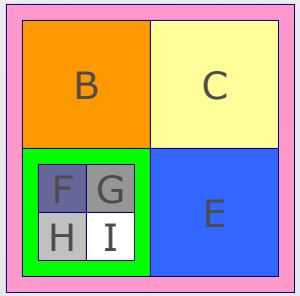
- El árbol se adapta a la distribución de puntos y no tiene por qué ser equilibrado
- El tiempo esperado de acceso es por tanto O(log_4n)





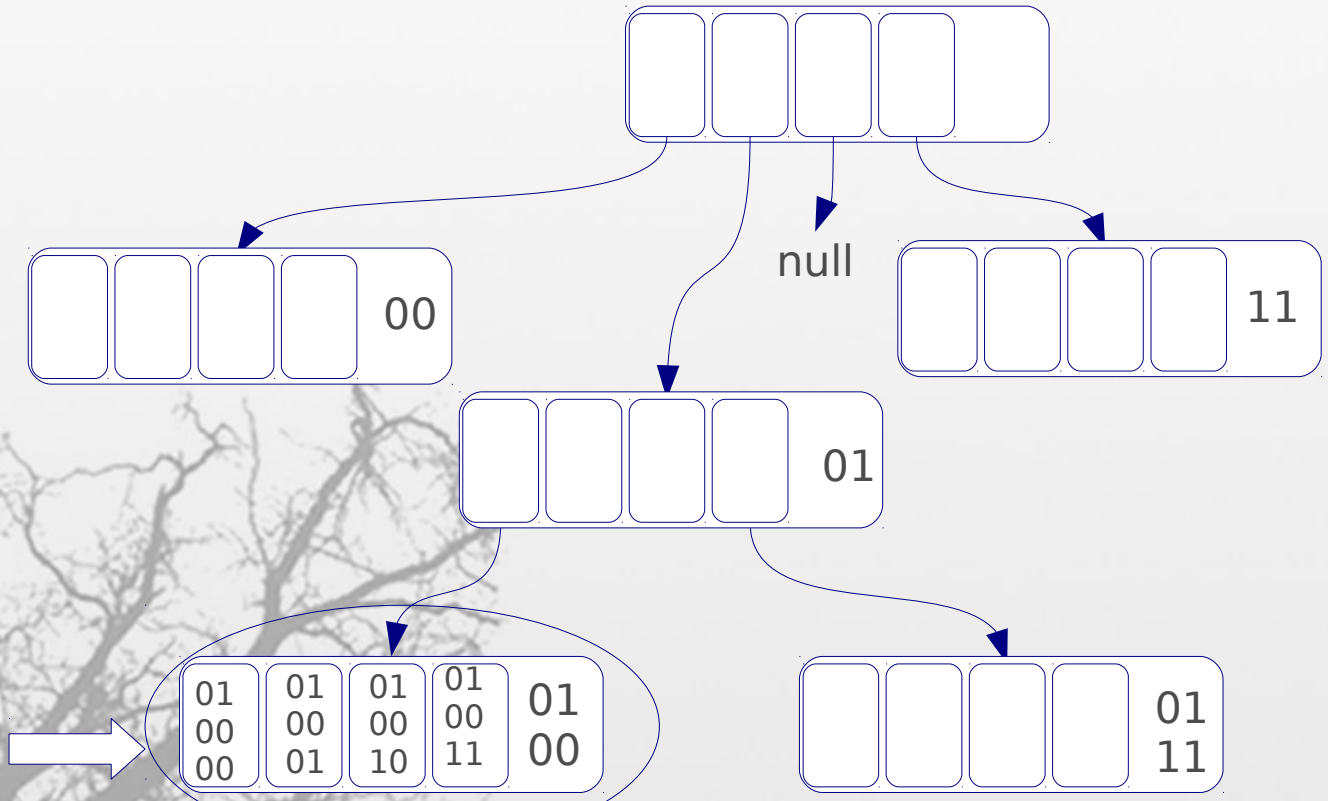


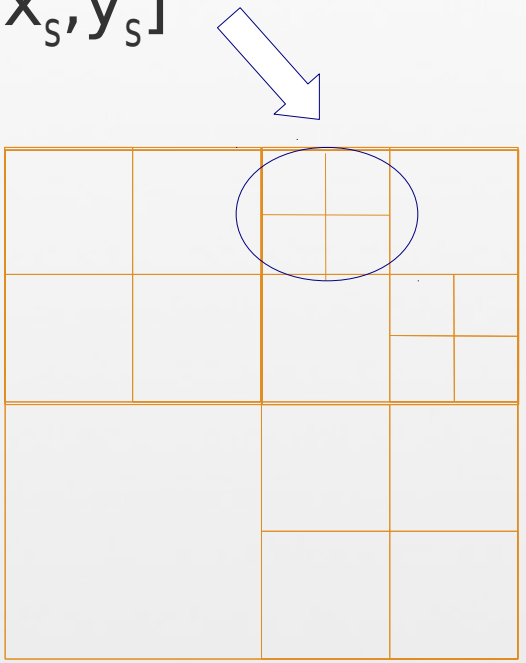




Quadtrees: creación

- Definición recursiva de los nodos del quadtree
- Cada nodo conoce su ubicación $[x_i,y_i][x_s,y_s]$





00	01
10	11

Quadtrees: creación

- Cada nodo almacena 4 punteros
- El puntero al padre es opcional, pero es interesantes para recorrer una región sin tener que comenzar el proceso desde la raíz
- El nodo raíz representa todo el escenario de trabajo [XMIN,YMIN][XMAX,YMAX]
- Los nodos hoja mantienen una lista de punteros a los puntos ubicados en la región que representa
- La clase Quadtree tiene un apuntador al nodo raíz y la EEDD que contiene la nube de puntos original
- Es necesario definir la condición de parada: por ejemplo que no haya más de un MAX_PUNTOS_CAJA

Quadtree: nodo

```
class Nodo {
       Nodo *hijos[4];  //ne, no, se, so
       Nodo *padre;
       double xsup, ysup, xinf, yinf;  //[0,1][0,1]
       list <Punto *> puntos;
public:
       friend class quadTree;
       Nodo(): puntos() {
          hijos[0] = hijos[1] = hijos[2] = hijos[3] =
          padre = 0;
          xsup = XMAX; ysup = YMAX; xinf = XMIN; yinf = YMIN;
       }

       Nodo (double xi,double yi,double xs,double ys,
                                    Nodo *papi=0): puntos() {
          hijos[0] = hijos[1] = hijos[2] = hijos[3] = 0;
          padre = papi;
          xsup = xs; ysup = ys; xinf = xi; yinf = yi;
       }

       bool esHoja (){
            return !(hijos[0] || hijos[1] || hijos[2] || hijos[3]);
       }
  };
```

añadimos el puntero al padre

puntos tiene la lista de puntos en las hojas, en el resto sirve para la construcción

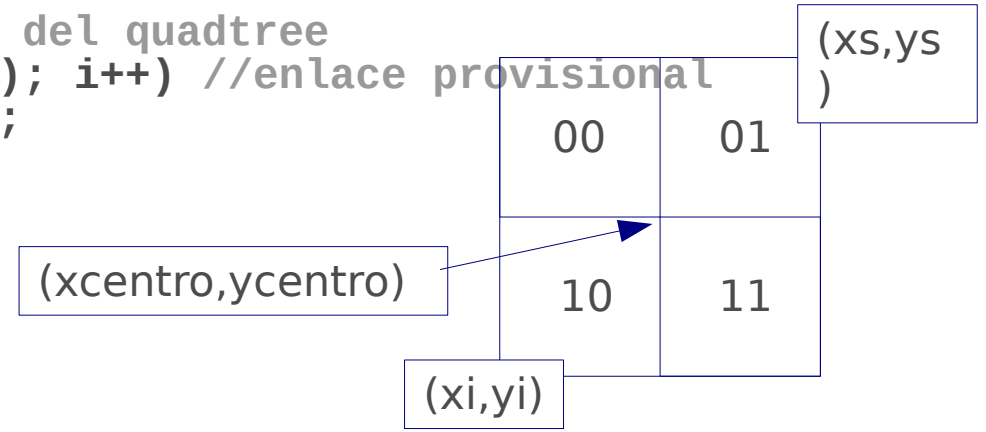
Quadtree: quadtree

```
class QuadTree {
    Nodo *raiz;
    vector<Punto> nube; //puntos originales

    Nodo* localiza(Nodo *raiz, const Punto &p);
    void inserta_nodo (int nivel, Nodo **h, double xin, double yin,
double xsu, double ysu, Nodo *padre, unsigned nnodo);
    void destruye(Nodo *nodo);
    bool esHoja(Nodo *nodo);

public:
    QuadTree(){ raiz = 0; }
    QuadTree(vector<Punto> &puntos);
    bool localiza(const Punto &p);
};
```

Quadtree: creación

Al menos el árbol tendrá un nodo

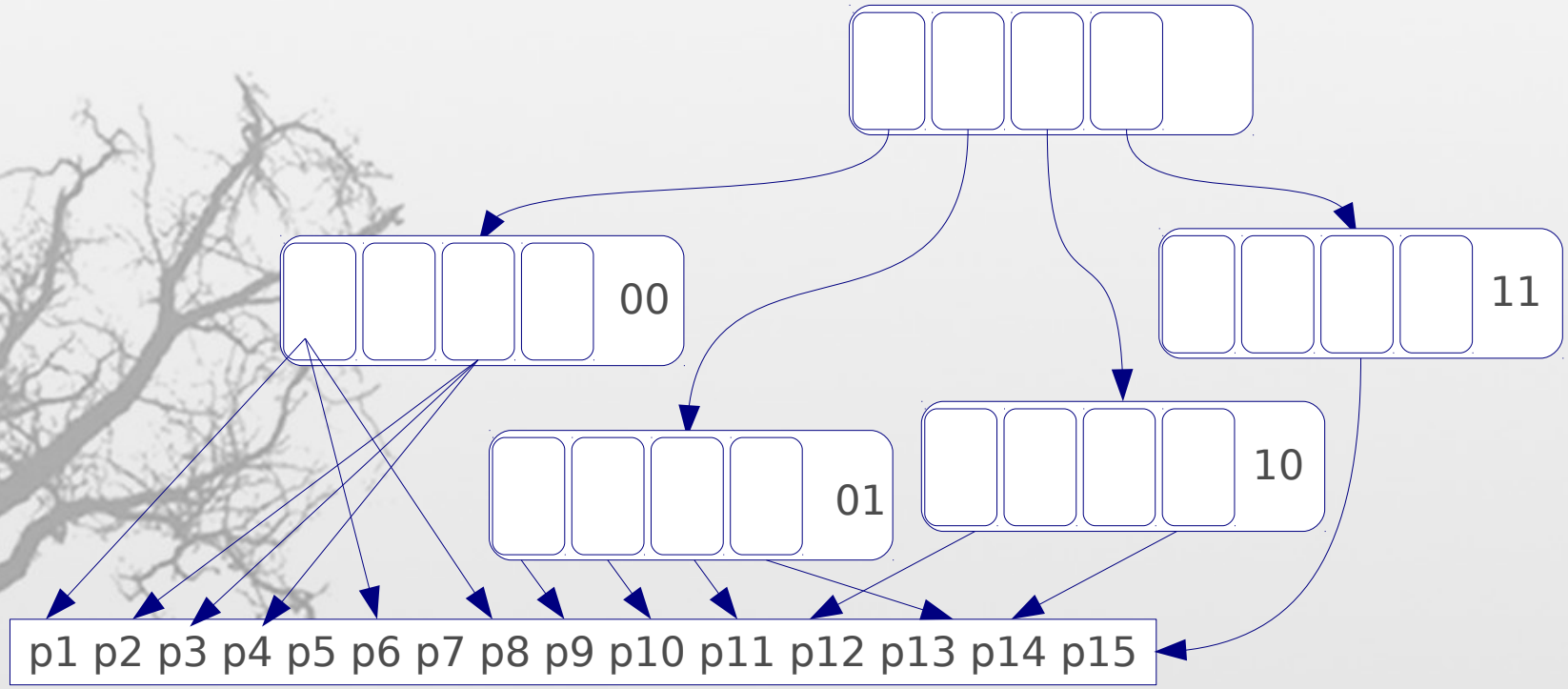
```
QuadTree::QuadTree(vector<Punto> &puntos){
   nube = puntos;
   raiz = new Nodo(); //inicializacion del quadtree
   for (unsigned i = 0; i < nube.size(); i++) //enlace provisional
      raiz->puntos.push_back(&nube[i]);

   double xs = raiz->xsup;
   double xi = raiz->xinf;
   double ys = raiz->ysup;
   double yi = raiz->yinf;

   double xcentro = (xs+xi)/2.0;
   double ycentro = (ys+yi)/2.0;
   insertaNodo(0, raiz->hijos[2],xi,yi,xcentro,ycentro,raiz,2);//10
   insertaNodo(0, raiz->hijos[0],xi,ycentro,xcentro,ys,raiz,0);//00
   insertaNodo(0, raiz->hijos[3],xcentro,yi,xs,ycentro,raiz,3);//11
   insertaNodo(0, raiz->hijos[1],xcentro,ycentro,xs,ys,raiz,1);//01
   raiz->puntos.clear();
}
```



todos los puntos desaparecen menos los de los nodo hojas

Quadtree: inserción

- Un nuevo nodo es siempre una hoja, y quita al padre los puntos que quedan dentro de su área
- Si no se cumple la condición de parada, se crean nuevos nodos hijo. Si finalmente es hoja se queda con esa lista de punteros

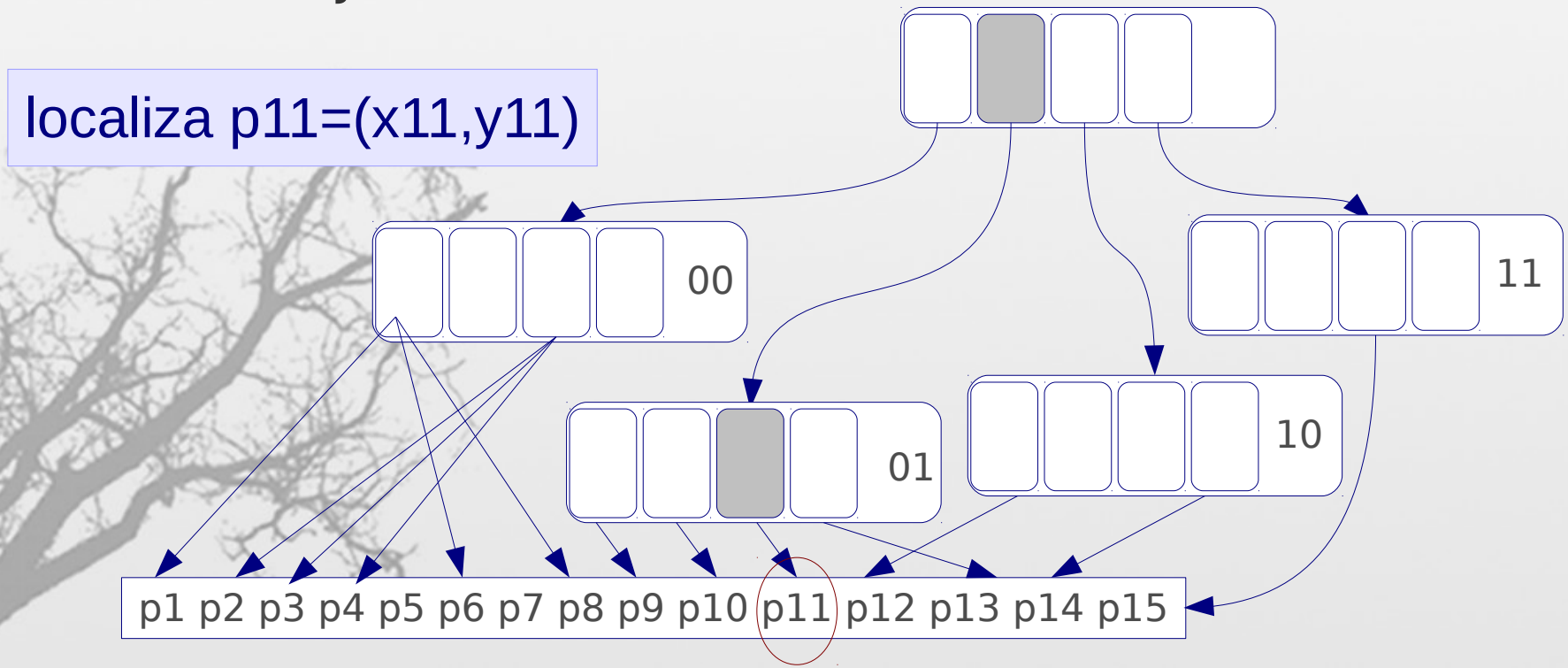


Quadtree: inserción

```
void QuadTree::insertaNodo (int nivel, Nodo& *nodo, double xin, double yin,
double xsu, double ysu, Nodo *padre, unsigned nnodo){

  list<Punto *>::iterator it = padre->puntos.begin();
  nodo = new Nodo(xin, yin, xsu, ysu, padre);

  while (it != padre->puntos.end()){
    Punto *t = *it;
    if (t->puntoEnCaja(xin, yin, xsu, ysu))
       nodo->puntos.push_back(*it);
    it++;
  }

  if ((*h)->puntos.size() > MAX_PUNTOS_CAJA){
     double xcentro = (xsu+xin)/2.0;
     double ycentro = (ysu+yin)/2.0;

     insertaNodo (nivel+1, nodo->hijos[2], xin,yin,xcentro,ycentro,nodo, 2);

     insertaNodo (nivel+1, nodo->hijos[0], xin,ycentro,xcentro,ysu,nodo, 0);

     insertaNodo (nivel+1, nodo->hijos[3], xcentro,yin,xsu,ycentro,nodo, 3);

     insertaNodo (nivel+1, nodo->hijos[1], xcentro,ycentro,xsu,ysu,nodo, 1);

     nodo->puntos.clear();
   }
}
```

Quadtree: localización

- Las coordenadas del punto a localizar llevan a una región única
- En esta región debe estar el punto
- Se recorre secuencialmente la lista de puntos del nodo hoja localizado



Quadtree: implementación

```
bool QuadTree::esHoja (Nodo *nodo){
   return !(nodo->hijos[0] || nodo->hijos[1]
          || nodo->hijos[2] || nodo->hijos[3]);
}
Nodo* QuadTree::localiza(Nodo* nodo, const Punto &p){
     if  (nodo->esHoja()){
         list<Punto *>::iterator it = nodo->puntos.begin();
         for (; it!= nodo->puntos.end(); it++){
             if (**it == p) return nodo;
         }
         return 0;
     }
     double xcentro = (ptraiz->xsup+ptraiz->xinf)/2.0;
     double ycentro = (ptraiz->ysup+ptraiz->yinf)/2.0;
     if (p.getX() < xcentro && p.getY() < ycentro)  //2
        return localiza(ptraiz->hijos[0],p);
     if (p.getX() < xcentro && p.getY() >= ycentro) //0
        return localiza(ptraiz->hijos[1],p);
     if (p.getX() >= xcentro && p.getY() < ycentro) //3
        return localiza(ptraiz->hijos[2],p);
     if (p.getX() >= xcentro && p.getY() >= ycentro) //1
        return localiza(ptraiz->hijos[3],p);
}
bool QuadTree::localiza(const Punto &p){
    return (localiza(raiz, p) != 0);
}
```

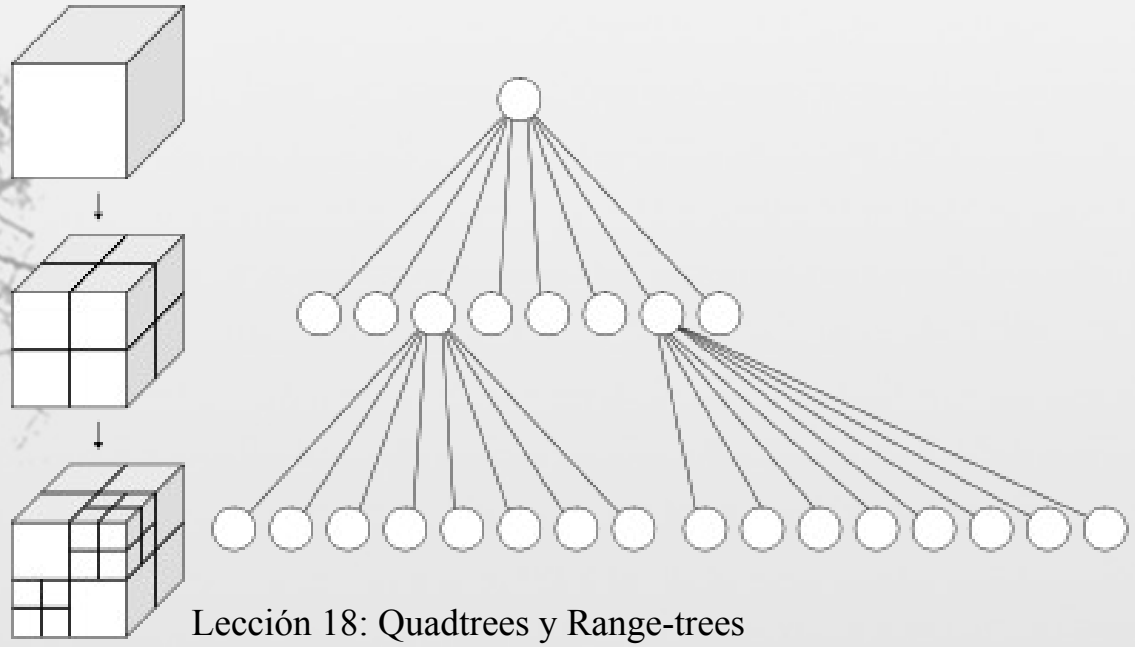
versión recursiva privada

operator== sobrecargado, cuidado con la comparación de datos en coma flotante!!!!

versión pública

Quadtrees/Octrees

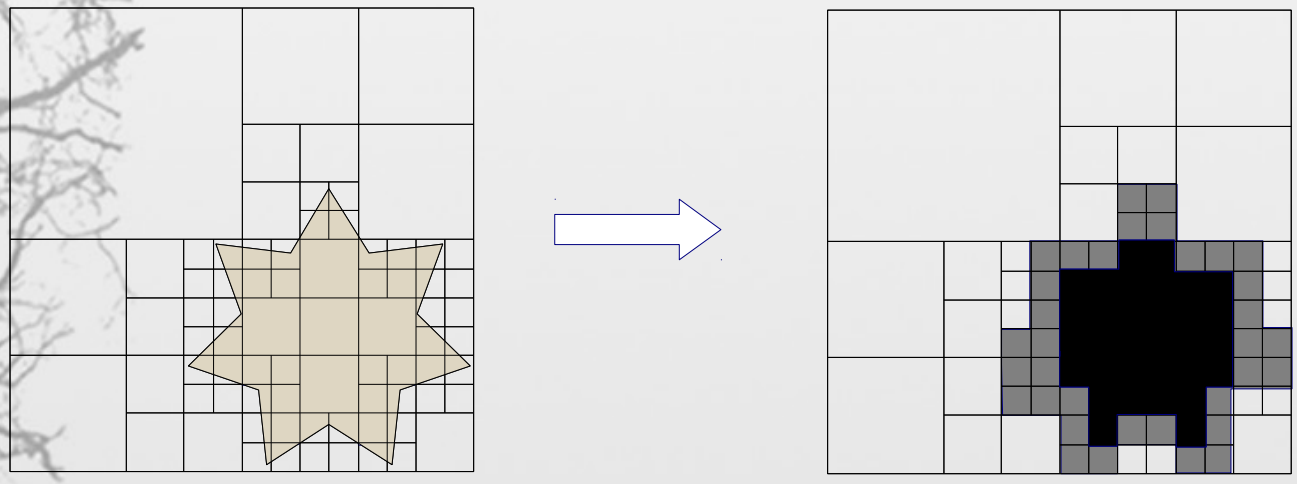
- Los quatrees particionan el plano, este concepto es fácilmente extensible al espacio con los **octrees**
- Un octree divide recursivamente la zona del espacio que representa en 8 subdivisiones
- La definición y los métodos vistos son extensibles a 3D



Aplicaciones

- Los quatrees/octrees tienen numerosas aplicaciones en Sistemas de Información Geográficos o Informática Gráfica

Representación de una fugura 2D para ser tratada en detección de colisiones:
- blanco: no hay colisión
- gris/negro sí hay colisión

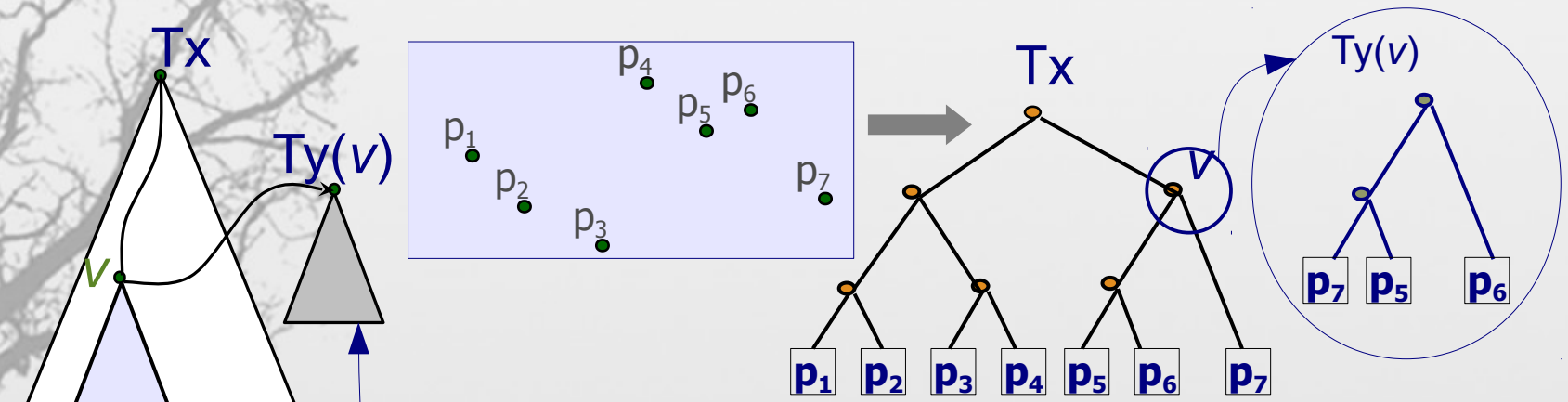


Range-tree

Telescopios virtuales:
Dado un fichero con información relativa a miles de estrellas, cuales son las que serían visibles en un rectángulo cualquiera?
Este es un ejemplo de búsqueda por rangos

Range-tree

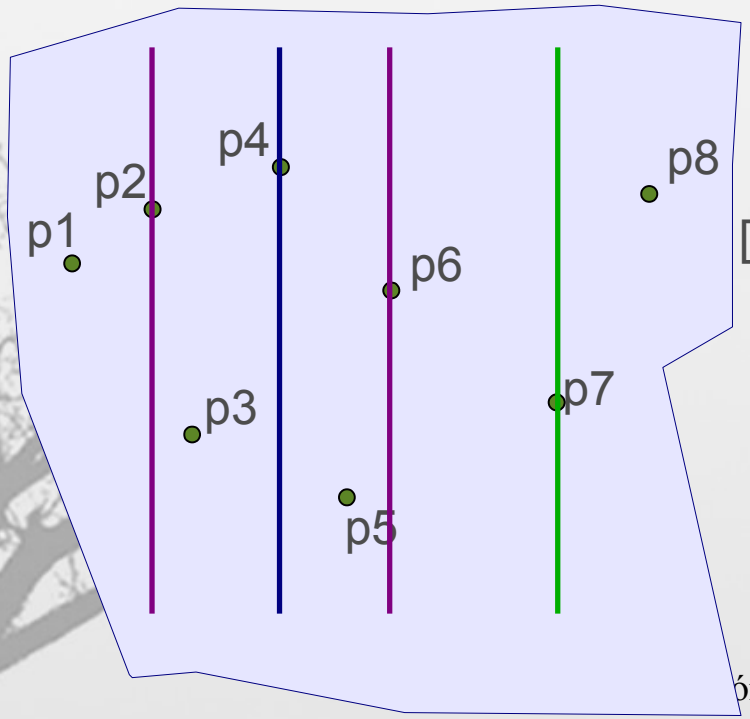
- Un **range-tree** es un árbol balanceado que almacena en cada nodo otro árbol balanceado
- El árbol principal es un **kd-tree** con **k=1** que ordena los puntos por la coordenada **X**
- El árbol interno a cada nodo mantiene los puntos del subárbol que representa ordenados por **Y**
- Este árbol interno se denomina **árbol canónigo**

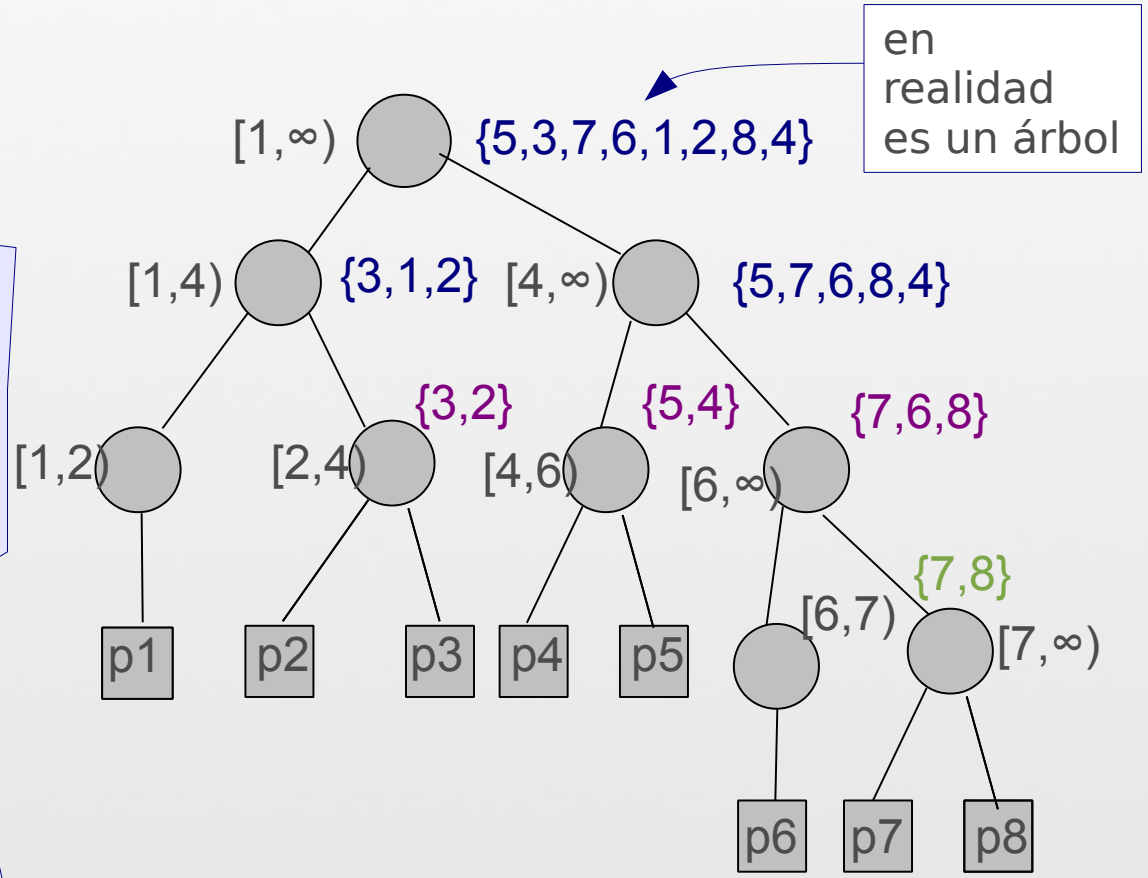


ambos subárboles mantienen los mismos puntos
ordenados por X
ordenados por Y

Range-tree

- El árbol canónigo de la raíz tiene todos los puntos ordenados por ordenadas (coordenada Y)
- El resto sólo tiene los nodos del subárbol que representa



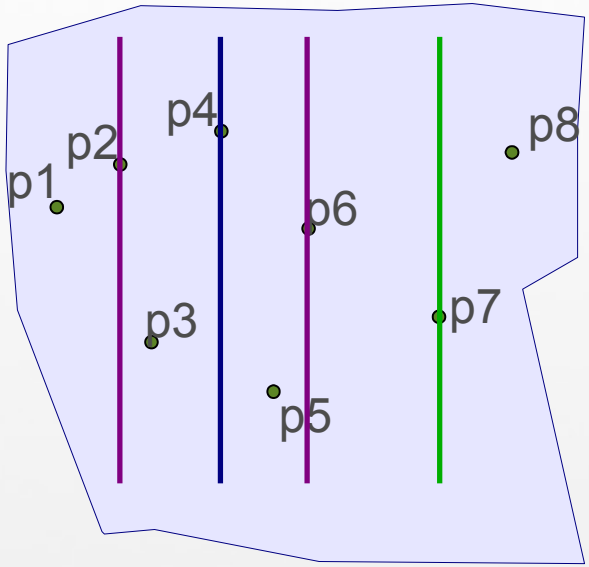


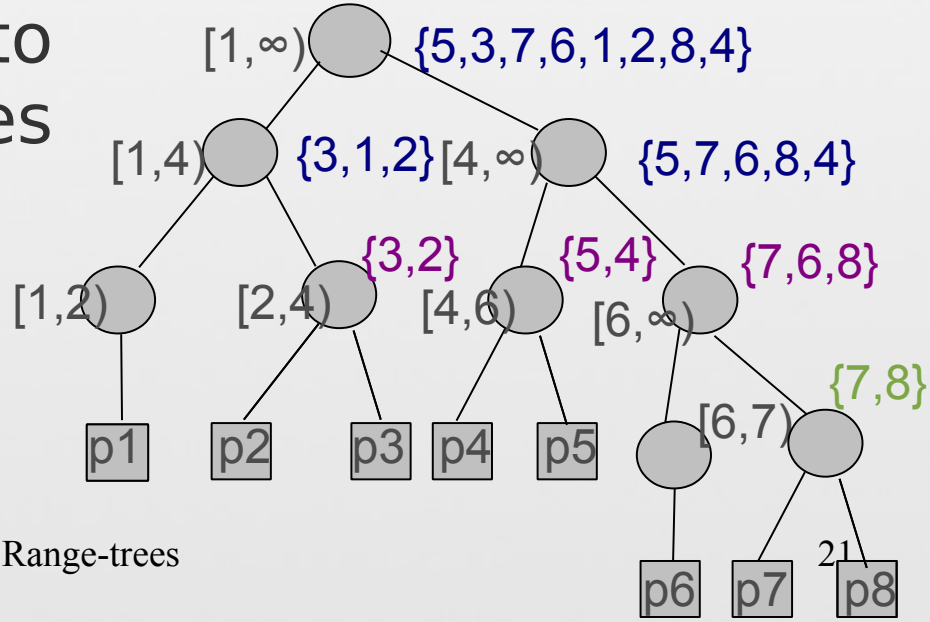
Range-tree: creación

```
Algoritmo Construir2DRangeTree(P)
Entrada: P de tamaño n
Salida: El árbol v resultado (su raíz)

INICIO
Construir el árbol binario Tcanon utilizando la coordenada Y del
conjunto P
SI Tamaño(P) = 1
ENTONCES
    Crear una hoja v cargando este punto y Tcanon como árbol canónigo
SINO
    Obtener Xmid, la coordenada x media
    Partir P en Pizda (conteniendo puntos con x<=Xmid)
    Partir P en Pdecha(conteniendo puntos con x>Xmid)
    v_izda     Construir2DRangeTree(Pizda)
    v_dech     Construir2DRangeTree(Pdecha)
    Crear v con hijos v_izda y v_dech y Tcanon como árbol canónigo
    Devolver (v)
FIN
```

Range-tree: creación

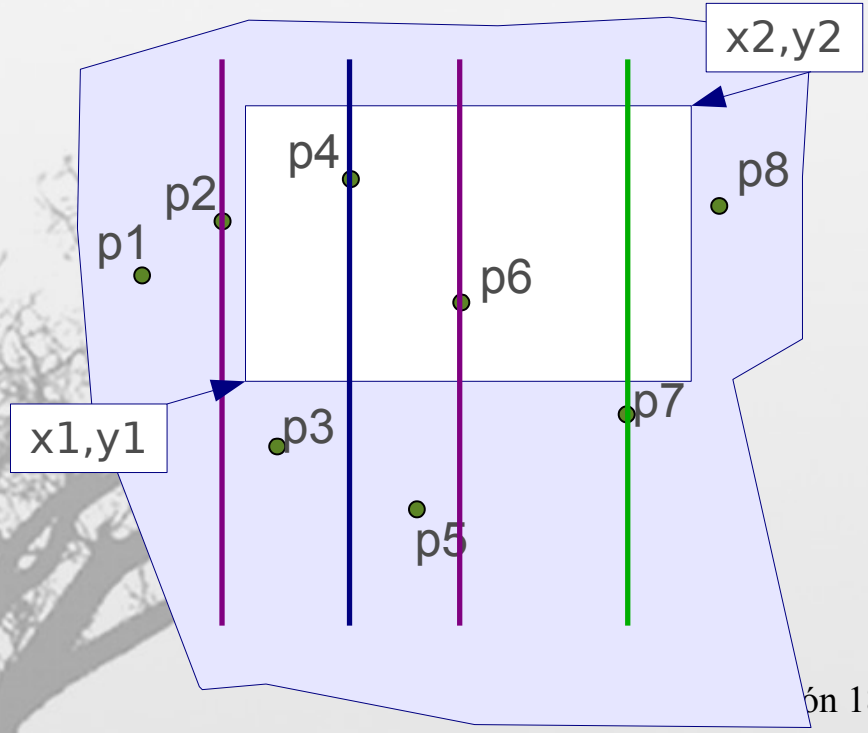
- Por el modo de construcción, el árbol queda equilibrado
- El tiempo de construcción es $O(n log^2 n)$
  - $T(1) = O(1)$
  - $T(n) = 2\,T(n/2) + O(n log n)$
- Partiendo de un conjunto ordenado el tiempo es $O(n log n)$
  - $T(1) = O(1)$
  - $T(n) = 2\,T(n/2) + O(n)$

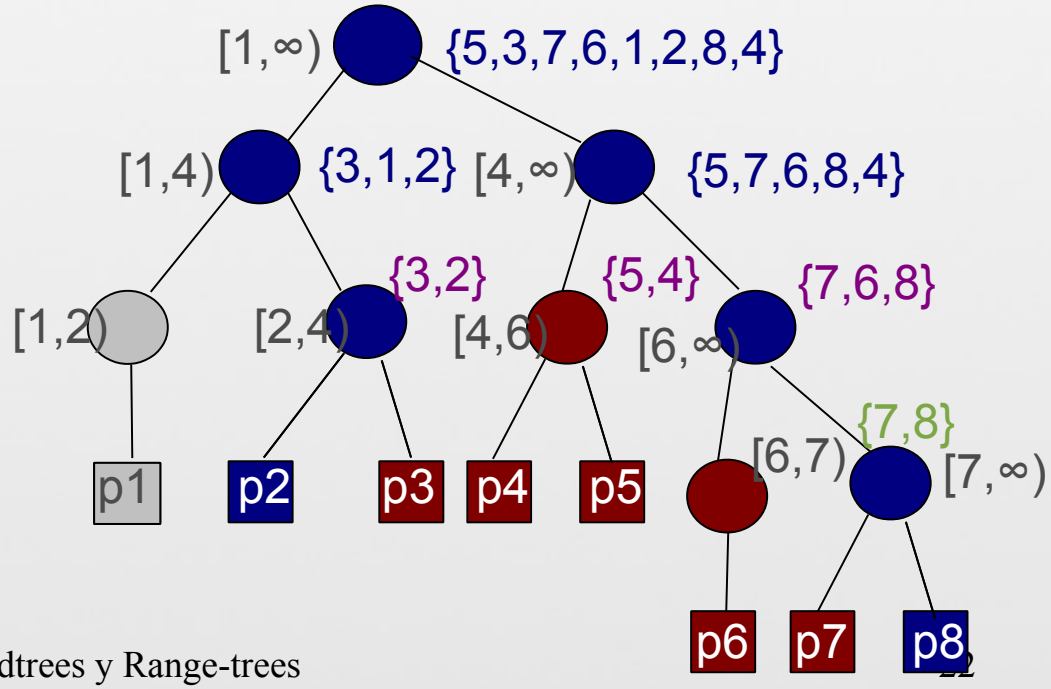




Range-tree: búsqueda por rango

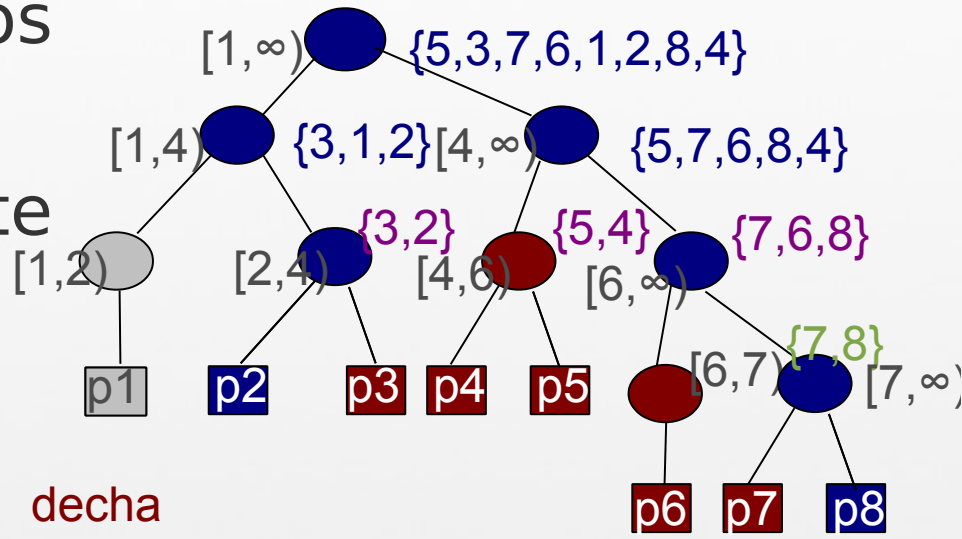
- La búsqueda por rango consiste en encontrar todos los puntos dentro del rectángulo [x1,y1][x2,y2]
- Para ello primero se hace una búsqueda por rango en el árbol principal en el intervalo [x1,x2]

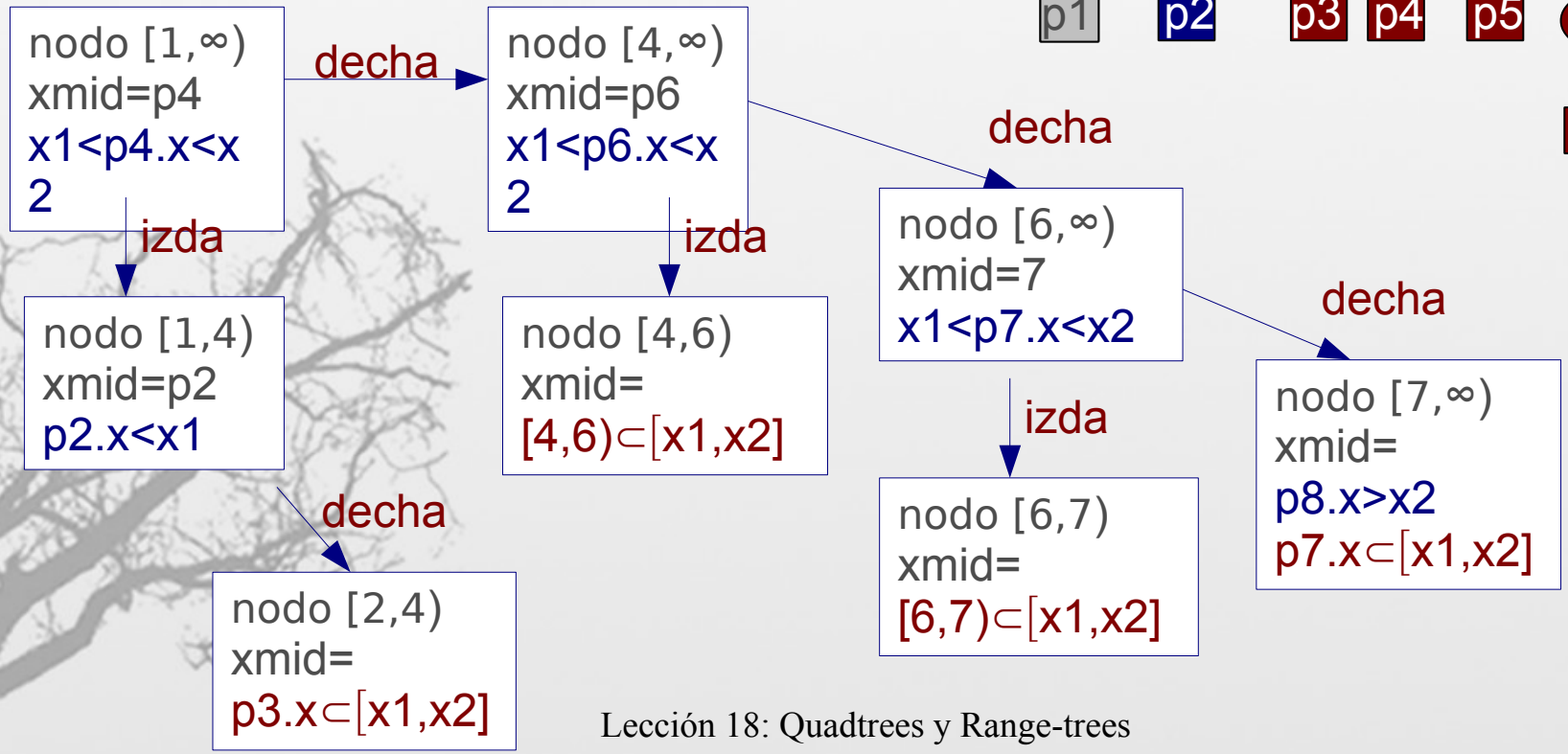




Range-tree: búsqueda

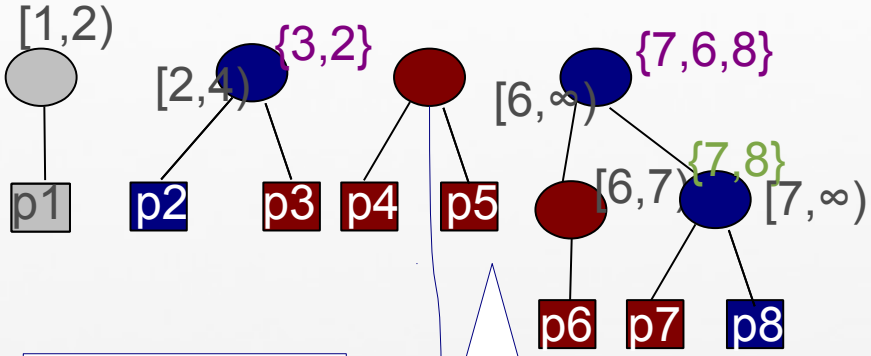
- Los nodos azules son nodos visitados
- los rojos están totalmente contenidos en [x1,x2]



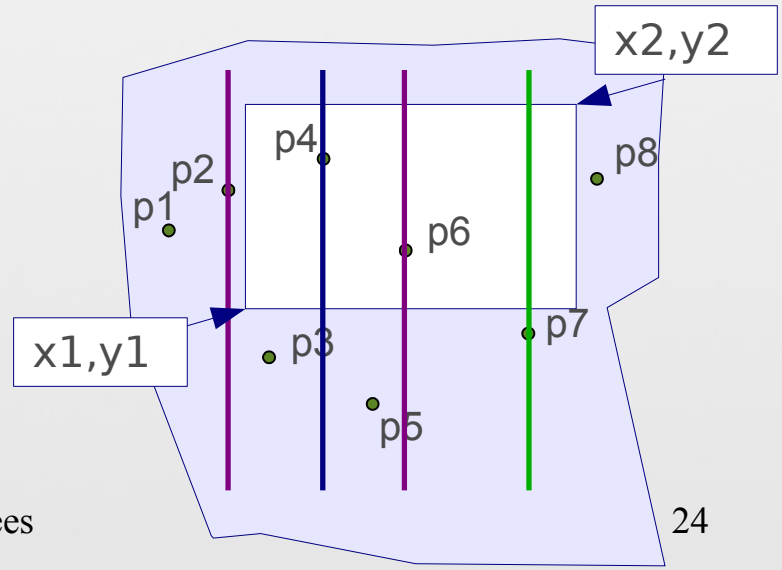


Range-tree: búsqueda

- No todos los nodos rojos están en **[y1,y2]**
- Si el nodo rojo es hoja, entonces si **y1<=p.y<y2**, entonces p∈[x1,y1][x2,y2]
- Si es nodo interior, entonces no es necesario descender, basta con buscar por rangos en su árbol canónigo

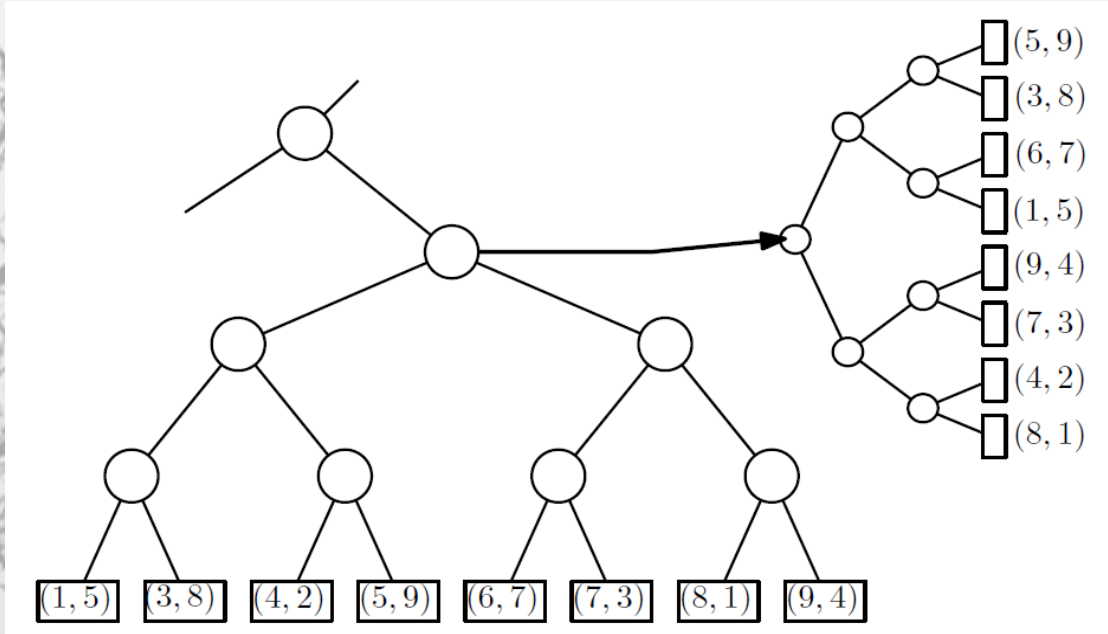


p4 y p5 se inspeccionan en el árbol canónigo



Range-tree: eficiencia

- El tiempo de búsqueda del árbol principal es O(*log*n)
- Dentro de cada árbol canónigo también se realizan búsquedas binarias
- Total: **O(log^2 n+k)** para k puntos en [x1,y1][x2,y2]



Consideraciones finales

- Las EEDD espaciales vistas en este capítulo permiten realizar búsquedas eficientes de un punto o de una región de puntos
- Los quadtrees son adaptativos y permiten accesos muy rápidos
- Los range-trees son especialmente interesantes para búsquedas por regiones
- Mejoran el tiempo de búsqueda de los kd-trees vistos en el tema anterior
- Ambos permiten extensiones a 3 dimensiones