Pontificia Universidad Javeriana
Departamento de Ingeniería de Sistemas Estructuras de Datos
Proyecto del curso, 2023-10

Christian Santiago Vera Rojas

Nicolás Alejandro Arciniegas Jaimes

Daniela Torres Gomez

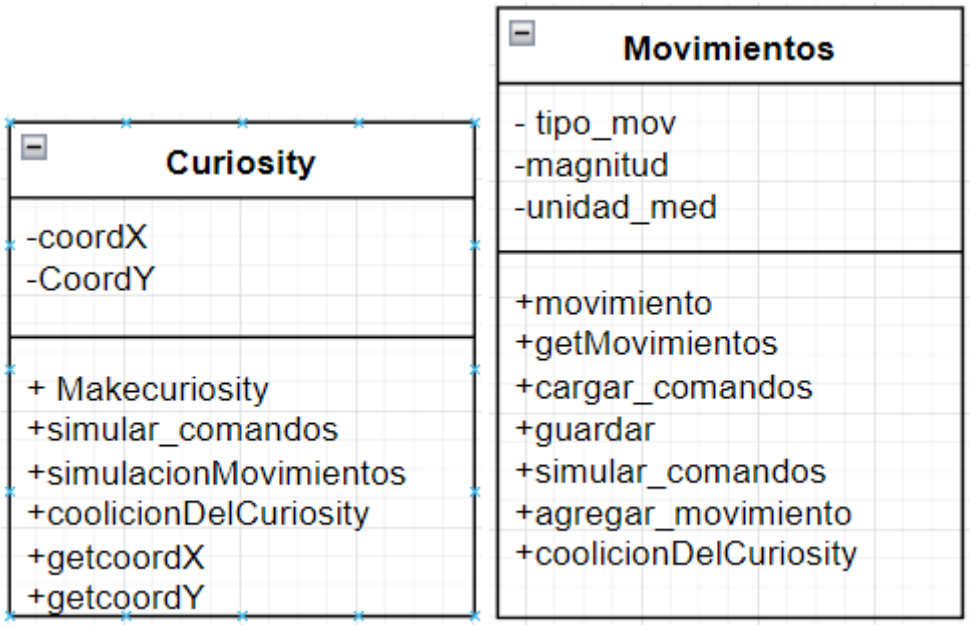
Propósito del sistema:

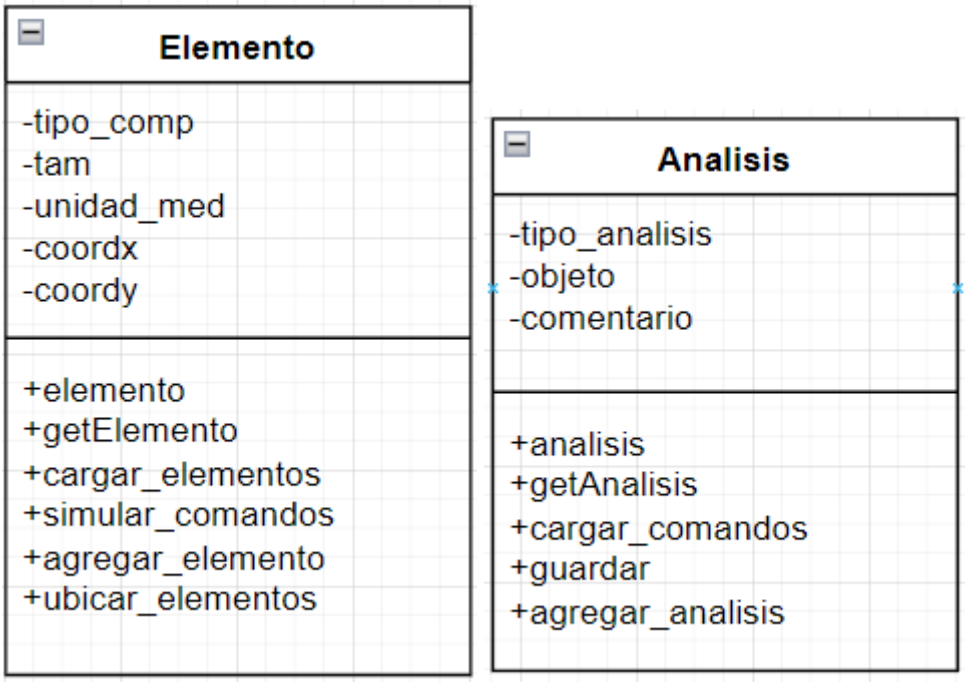
Simular el movimiento de un objeto en un plano bidimensional y mapear óptimamente elementos dentro de este, para la organización jerárquica de los elementos en un plano cartesiano:

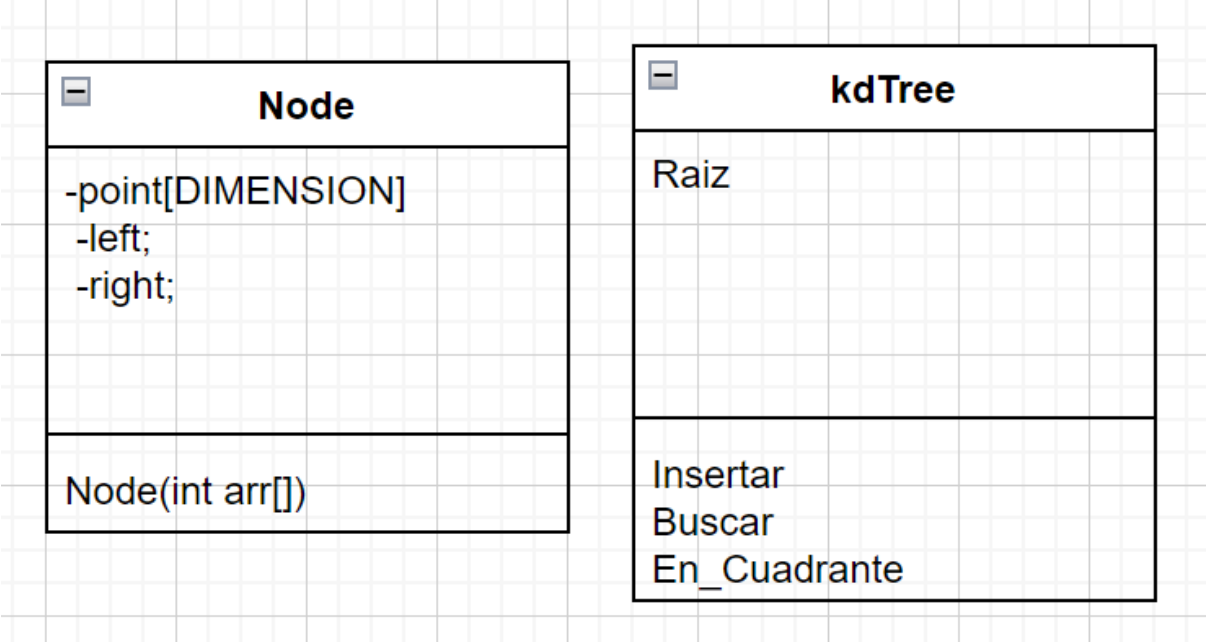
Objetivos del diseño:

Entender cómo funciona un plano bidimensional con temas propios de las estructuras de datos dentro de la programación en C++:

Estructuras de datos:







TAD CURIOSITY

Datos mínimos:

coordX: dato de tipo float, representa la coordenada en el eje x del vehículo curiosity
coordY: dato de tipo float, representa la coordenada en el eje y del vehículo curiosity

Operaciones:

Makecuriosity(ncoordX, ncoordY): retorna un objeto de la clase curiosity, funciona como el constructor de la clase.
simular_comandos: simular_comandos(coordX, coordY, *listaMovimientos, *ListaElemento, *curiosity): es de tipo void, lo que hace es de acuerdo a los movimientos y elementos que se puede encontrar el vehículo al ejecutar los comandos respectivos, y esto lo guardan los atributos del objeto curiosity.
simulacionMovimiento(magnitud, radianes, Curiosity): retorna un objeto de la estructura Curiosity, modificando sus atributos originales, con unas nuevas las cuales se calculan a partir de las funciones CoordenadasEnX y CoordenadasEnY.
coolicionDelCuriosity(nMagnitud, nRadianes, Curiosity, ListaElemento): retorna un void e imprime en pantalla si el robot se ha chocado con algún elemento o si llegó a su punto objetivo.
getcoordX(Curiosity): retorna la coordenada X del objeto de la estructura Curiosity que se le ingresó.
getcoordY(Curiosity): retorna la coordenada Y del objeto de la estructura curiosity que se le ingresó.

TAD MOVIMIENTOS

Datos mínimos;

tipo_mov: es un bool, el cual representa si el vehículo se desplaza o gira.
magnitud: float, representa la magnitud del movimiento de acuerdo al tipo.
unidad_med: es un string, el cual representa la unidad de medida en la cual se da el movimiento de acuerdo a su tipo.

Operaciones:

movimiento(ntipo_mov. magnitud. nunidad_med): retorna un objeto del tipo estructura Movimientos, funciona como el constructor de la clase.
getMovimientos(Movimientos): es del tipo void.
cargar_comandos(nombre_archivo, *listaAnalisis, *listaMovimientos): no retorna nada, es del tipo void, lo que hace es tomar los datos guardados en el archivo de comandos y los separa de acuerdo al tipo de comando.
guardar(tipo_archivo, nombre_archivo, *ListaElemento, *listaAnalisis, *listaMovimientos): no retorna nada, es del tipo void, guardar los comandos de las diferentes listas en un archivo, con un formato igual al que necesita el programa para luego poder leer el archivo nuevamente.
simular_comandos(coordX, coordY, *listaMovimientos, *ListaElemento, *curiosity): es de tipo void, lo que hace es de acuerdo a los movimientos y elementos que se puede encontrar el vehículo al ejecutar los comandos respectivos, y esto lo guardan los atributos del objeto curiosity.
agregar_movimiento(tipo_mov, magnitud, unidad_med, *listaMovimientos): es del tipo void, toma los datos que entran a la funcion, los unen en un objeto de la estructura movimientos, para luego agregarlo a la lista de movimientos.
coolicionDelCuriosity(nMagnitud, nRadianes, Curiosity, ListaElemento): retorna un void e imprime en pantalla si el robot se ha chocado con algún elemento o si llegó a su punto objetivo.

TAD ELEMENTO:

Datos mínimos:
tipo_comp: string, representa el material del que esta compuesto el elemento
tam: float, representa el tamaño del elemento
unidad_med:string, representa la unidad de medida en la cual esta escrito el tamaño
coordx: dato de tipo float, representa la coordenada en el eje x del elemento
coordy: dato de tipo float, representa la coordenada en el eje y del elemento

operaciones:
elemento(nTipo_comp, tam, unidad_med, ncoordx, ncoordy): retorna un objeto del tipo estructura elemento, funciona como constructor de la clase
getElemento(*elemento): es del tipo void
cargar_elementos(nombre_archivo, *ListaElemento): es de tipo void, toma la informacion de archivo elementos y la transforma en un objeto del tipo estructura Elementos, que luego guarda en la lista de elementos
simular_comandos(coordX, coordY, *listaMovimientos, *ListaElemento, *curiosity): es de tipo void, lo que hace es de acuerdo a los movimientos y elementos que se puede encontrar el vehiculo al ejecutar los comandos respectivos, y esto lo guardan los atributos del objeto curiosity.
agregar_elemento(tipo_comp, tamaño, unidad_med, coordX, coordY, *listaElemento): es del tipo void, toma los valores de entrada y los convierte en el tipo de estructura de elemento, para luego ponerlo en la lista de elementos
ubicar_elementos(*ListaElemento, *kdtree): es del tipo void, toma la lista de elementos y los coloca en el mapa.

TAD ANÁLISIS:

Datos mínimos:
tipo_analisis: es del tipo char, representa con su inicial el tipo de análisis que debe ejecutar el vehículo
objeto: es de tipo string. es el nombre del objeto
comentario:es de tipo string, representa un valor opcional que permite adicionar algún tipo de información acerca del análisis o del elemento.

Operaciones:
analisis(ntipo_analisis, nobjeto, comentario): retorna un objeto del tipo estructura analisis, funciona como constructor de la clase
getAnalisis: es de tipo void
cargar_comandos(nombre_archivo, *listaAnalisis, *listaMovimientos): no retorna nada, es del tipo void, lo que hace es tomar los datos guardados en el archivo de comandos y los separa de acuerdo al tipo de comando
guardar(tipo_archivo, nombre_archivo, *ListaElemento, *listaAnalisis, *listaMovimientos): no retorna nada, es del tipo void, guardar los comandos de las diferentes listas en un archivo, con un formato igual al que necesita el programa para luego poder leer el archivo nuevamente.
agregar_analisis(tipo_analisis, objeto, comentario, *listaAnalisis): es del tipo void, toma los datos que entran a la función, los unen en un objeto de la estructura análisis, para luego agregarlo a la lista de análisis.

coolicionDelCuriosity(nMagnitud, nRadianes, Curiosity, ListaElemento): retorna un void que representa si el vehículo se chocó con algún elemento durante su movimiento

TAD NODO

Datos mínimos:

Punto: Coordenadas x y y en el plano bidimensional.
Nodo izquierdo: Elementos a la izquierda de un nodo raíz
Nodo derecho: Elementos a la derecha de un nodo raíz.

TAD KD TREE

Datos mínimos:

Raíz: Nodo que será predecesor de todos los demás nodos

Operaciones

newNode((int arr[]): Constructor del nodo raíz:
Insertar(Raiz, Punto Profundidad: Pasar por las dimensiones de cada punto, ver por coordenada x si está más a la derecha o a la izquierda, luego si está por arriba o por abajo, insertar en el árbol según esto, profundidad permite tomar la coordenada x, comparar y luego tomar la coordenada y
Buscar(Raiz, Punto Profundidad): Recorrer en el árbol y buscar unas coordenadas en el plano bidimensional
Puntos Iguales(Primer Punto, Segundo Punto): Verificar si hay unas coordenadas en X y en Y que se repitan
En_Cuadrante(Raiz, Punto min, punto max, profundidad): Mostrar todos los puntos que se encuentran dentro de un cuadrante
kdTree: Constructor para la raiz del kd tree
kdTree(Punto): Constructor, que tome un punto en el plano bidimensional

FUNCIONES AUXILIARES QUE NO SON USADAS DIRECTAMENTE POR LOS TAD

CoordenadasEnX(magnitud, radianes): Es un float que obtiene la magnitud y radianes de la coordenada x
CoordenadasEnY(magnitud, radianes): Es un float que obtiene la magnitud y radianes de la coordenada y
simulacionCoolicionConObjetos(tamaño, coordX, coordY, posicionActualX, posicionActualY):retorna un bool, el cual representa si el vehículo en cierta posición se chocará con un objeto de prueba de cierto tamaño en cierta posición.
addTxtExtension(filename) retorna un string el cual es la información guardada en filename seguido de ".txt", si ya tiene .txt no hará nada
help():es un void, retorna la información guardada en el archivo "help.txt" el cual es la explicación para el usuario de cómo usar los diferentes comandos

conversionUnidadesCuadradas(unidad_med, valorAConvertir): retorna un float, el cual es valorAConvertir, expresado en metros cuadrados
conversionUnidades(unidad_med, valorAConvertir): retorna un float, el cual es valorAConvertir, expresado en metros
conversionRadianes(unidad_med, valorAConvertir) retorna un float, el cual es valorAConvertir, expresado en radianes

1.Tokenización

Se divide el prompt en cada usuario para guardar cada palabra puesta por el usuario en un arreglo user Input[] y el valor argc que cuenta la cantidad de palabras puestas por el usuario, luego se crea un std::map para relacionar el string de la primera palabra que se guardará en userInput[1] con una función lambda con funciones void que ejecutarán la validación de lo puesto por el usuario con el llamado a la función, la interfaz de estas funciones están en funcionalidades.y su implementación en funcionalidades.cpp

```
 bool running = true;
std::list<Movimientos> listaDeMovimientos;
std::list<Analisis> listaDeAnalisis;
std::list<Elemento> listaDeElementos;
kdTree<2> KDTree;
Curiosity curiosity;
  while(running){
//TOKENIZACION, SEPARACION DEL COMANDO POR ESPACIOS
    std::string comand;
    printf("$");
    getline(std::cin,comand);
    char comando[MAX_LENGHT];
    char* token;
    strcpy(comando,comand.substr(0,MAX_LENGHT-1).c_str());
    token = strtok(comando," ");
    std::string last; //ULTIMO COMANDO CADENA
    std::string userInput[MAX_LENGHT];
    int argc = 0;
    while(token != NULL) {
        argc++;
        userInput[argc] = token;
        last = token; //GUARDA ULTIMO COMANDO CADENA
        token = strtok(NULL," ");
    }
    argc++;
    userInput[argc] = last; //AGREGAR LAST A ARRAY DE USER INPUT
    if((argc-1) > 6){
        printf("opcion no valida\n");
        return 0;
    }
```

En esta parte del código se divide el prompt en el array userInput, ahí un bucle while(running) para poder poner varias veces el prompt para el usuario. En un std::map tenemos el nombre por el que el usuario conoce la función, la validación de datos, que incluye el número de argumentos que debe tener la función y el llamado a la función:

2. Funcionalidades:

El proyecto debe tener las siguientes funcionalidades:

cargar comandos:

El programa debe ser capaz de cargar de un archivo de texto, los movimientos y análisis propuestos por un usuario.

La interfaz de esta función es:

```
void cargar_comandos(std::string nombre_archivo,std::list<Analisis>
*listaAnalisis,std::list<Movimientos> *listaMovimientos);
```

donde el primer parámetro es el nombre que tendrá el archivo .txt a cargar y un puntero a una lista de análisis y movimientos

cargar elementos:

El programa debe ser capaz de cargar de un archivo de texto, los elementos propuestos por un usuario.

La interfaz de esta función es:

```
void cargar_elementos(std::string nombre_archivo,std::list<Elemento>
*ListaElemento);
```

donde el primer parámetro es el nombre que tendrá el archivo .txt a cargar y un puntero a una lista de elementos
guardar:

El usuario debe ser capaz de seleccionar si va a guardar comandos o elementos y ponerlos en un archivo de texto con un nombre personalizado

```
void guardar(std::string tipo_archivo,std::string
nombre_archivo,std::list<Elemento> *ListaElemento,std::list<Analisis>
*listaAnalisis,std::list<Movimientos>
```

donde el primer parámetro es si va a guardar análisis y movimientos o elementos, el nombre propuesto por el usuario y la lista de elemento, movimiento y análisis.

agregar elemento, movimiento y análisis:

Así mismo, se debe hacer las funciones correspondientes para llenar las listas de elemento, análisis y

movimientos, cada una toma los parámetros como prompts puestas por el usuario y tokens guardados en user Input según los elementos de o elemento, movimiento o análisis. Aquí se presentan las interfaces que va a utilizar para la función de agregar.

```
void agregar_analisis(std::string tipo_analisis,std::string
objeto,std::string comentario,std::list<Analisis> *listaAnalisis);
//agregar_movimiento userInput[1], tipo_mov userInput[2], magnitud
userInput[3], unidad_med userInput[4]
void agregar_movimiento(std::string tipo_mov,std::string
magnitud,std::string unidad_med,std::list<Movimientos>
*listaMovimientos);
//agregar_elemento userInput[1], tipo_comp userInput[2], tamaño
userInput[3], unidad_med[4], coordX[5], coordY[6]
void agregar_elemento(std::string tipo_comp,std::string
tamaño,std::string unidad_med,std::string coordX,std::string coordY,
std::list<Elemento> *ListaElemento);
```

Conversión de unidades:

Para utilizar el Sistema Internacional de Unidades, un prompt válido para las unidades del curiosity es mm, cm y km, se ha creado en el archivo conversionUnidades.cpp el manejo de estas unidades de medida ,así como, una validación para no aceptar valores puesto por el usuario que no son números o números negativos.

simular comando:

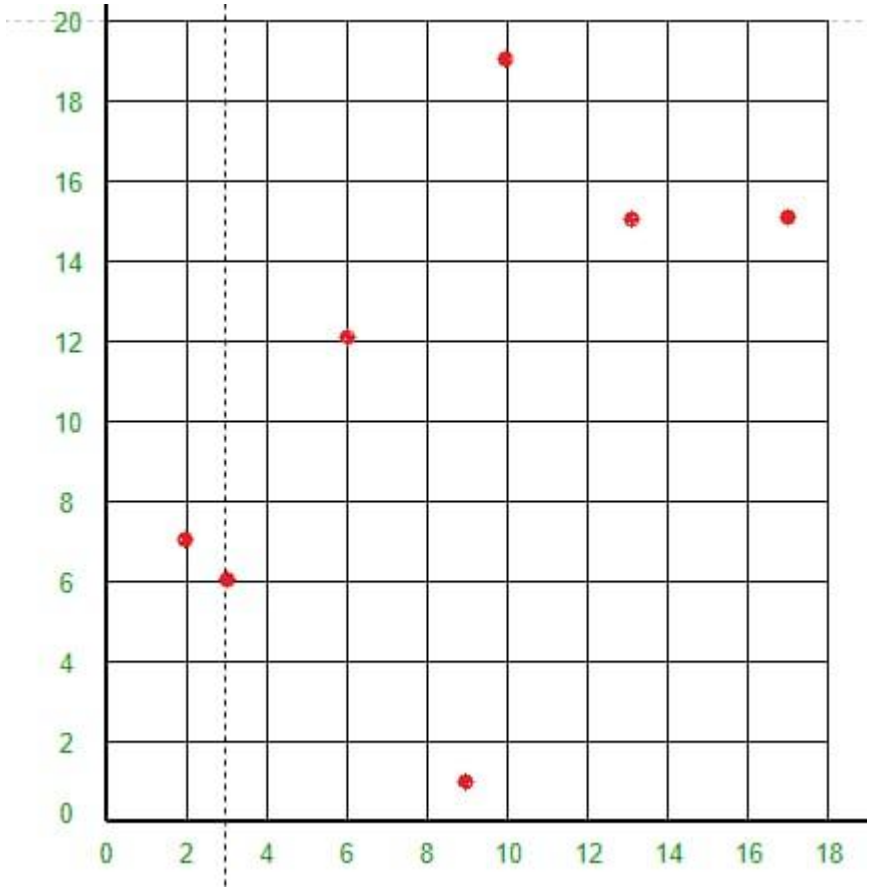
El programa debe ser capaz de simular como sería el movimiento del curiosity dado una orden de movimientos ordenados para el movimiento paso por paso, así como detectar colisiones con los elementos puestos por el usuario.

El programa pasará por cada movimiento y ejecutará una función que puede encontrar los componentes en ejes x y en eje y del curiosity, para simular el movimiento en el plano cartesiano, por cada movimiento, vera si una línea, que va del punto al que llegó al centro de cada elemento es más pequeño que la mitad del lado de un elemento que se representa como un cuadrado, si la línea del punto en donde esta el curiosity al centro del elemento es más pequeña que la mitad del lado del elemento habrá una colisión.

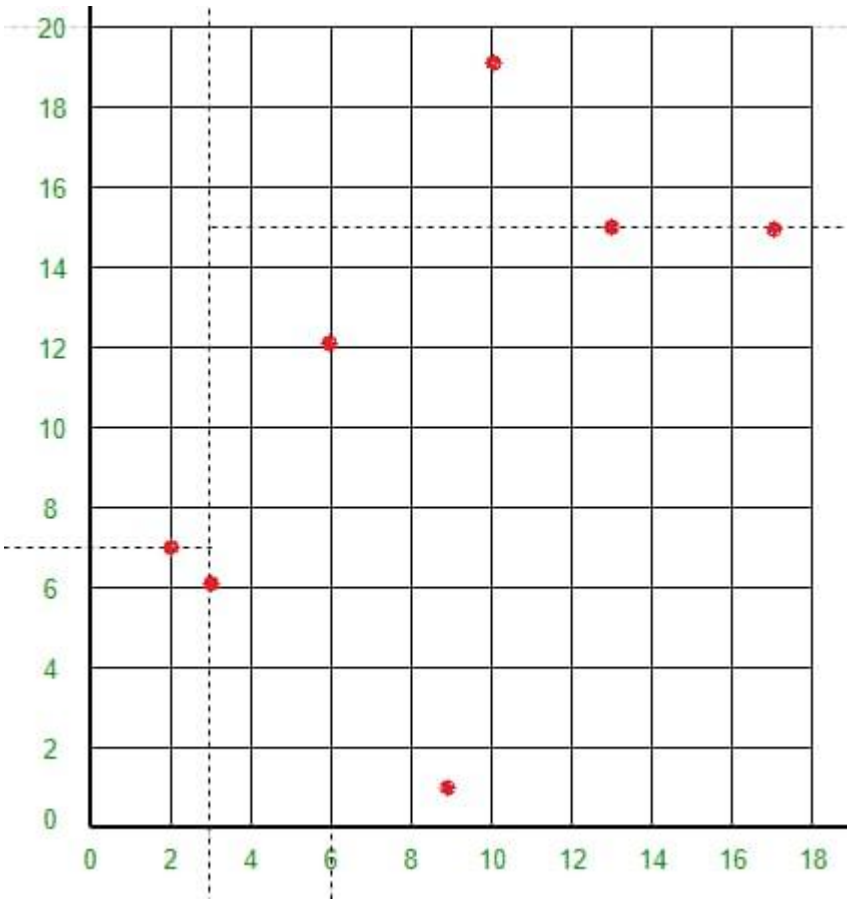
Para ello se ha creado las siguientes funciones:
CoordenadasEnY: Simulará el movimiento en el componente y
CoordenadasEnX: Simulará el movimiento en el componente x
simularCoolicionConObjetos: Un booleano que dirá si hubo coalición o no
simulacionMovimiento: Ejecutará los cambios hechos al curiosity en el componente y y en el componente x
coolicionDelCuriosity: Vera por cada movimiento, si colisionó para simular el movimiento del curiosity

ubicar elementos

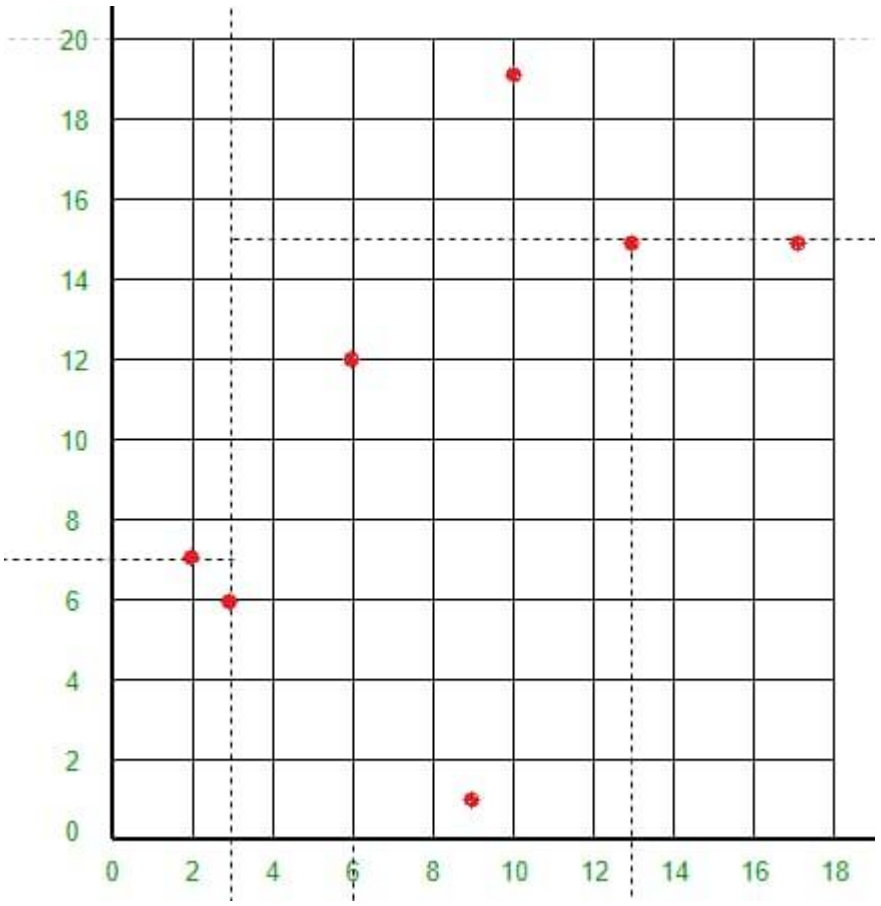
El programa deberá ser capaz de organizar jerárquicamente los puntos en el espacio de los objetos, para ello se hará la implementación de un kd Tree, el kd Tree separa el espacio según los valores en x y en y de cada elemento para ubicarlos dentro de un espacio bidimensional de la siguiente manera.



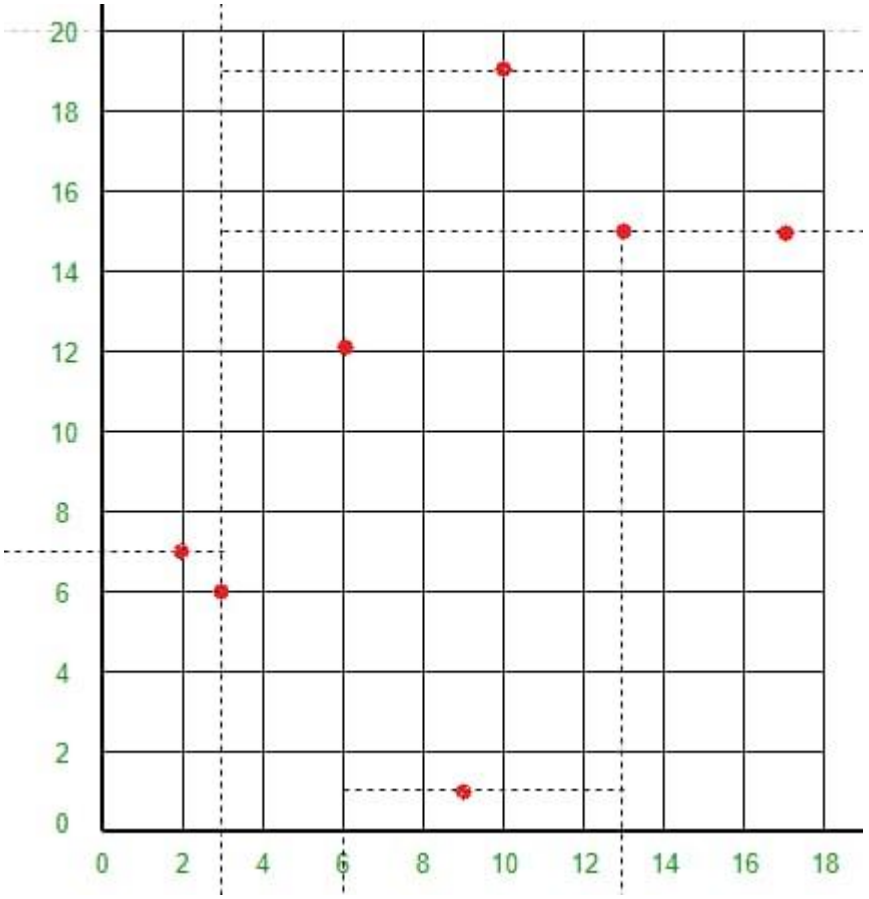
El programa encontrará un elemento y separa el espacio en el eje x



El programa encuentra un elemento en el subespacio a la derecha y a la izquierda en el eje x del elemento y hace cuadrantes en el eje y



El programa volverá a tomar valores en el eje x, y creará más subespacios



De esta manera se irá organizando jerárquicamente los elementos, cada uno con coordenada x y y

Decir las coordenadas de los elementos en un cuadrante:

Dentro de la implementación del kd tree, se agregara una función en_cuadrante que atravesará el kd-tree en postorden y verificará por cada elemento con coordenadas x y y. si este se encuentra dentro de los límites propuestos por el usuario