M.TECH. (COMPUTER SCIENCE) DISSERTATION SERIES

ACCESS STRUCTURES FOR AN IMAGE DATA BASE

a dissertation submitted in partial fulfillments of the
requirements for the M.Tech. (Computer Science) degree of
the Indian Statistical Institute

By

SUDHANSU SEKHAR KUILA

under the supervision of

AMARNATH GUPTA

&

S. K. PARUI

INDIAN STATISTICAL INSTITUTE

203, Barrackpur Trunk Road

Calcutta - 700 035.

Contents

Abstract

A new indexing structure called **Spatial kd-Tree** is proposed in the book named, "Efficient Query processing in Geographic Infomation Systems" by Beng Chi Ooi. The new structure supports two types of search, namely the Containment search and Intersection search in order to facilitate the evaluation of queries involving spatial (geometric) objects. I have implemented the skd-tree structure for isothetic rectangular objects for retrieving all objects which are contained in or intersect with the isothetic query rectangle. I have extended the structure and have seen its implementation for convex polygonal objects keeping the query rectangle as before (i.e, isothetic rectangle).

Further extension is made for finding the intersection between convex polygonal object and isothetic rectangle and also for finding the area of the intersecting region.

1 Introduction

In Geographic Information Systems, the database represents a collection of geographic objects in a two dimensional map. Each geographic object can be classified as belonging to a particular entity class such as city, road, lake etc. Objects are described by their aspatial attributes (e.g, population, name etc) as well as their spatial attributes (e.g, location). Furthermore, they may be grouped into 3 generic spatial object classes namely, point, line, and region. Our project deals with region type objects.

On a given map, the geographic objects may intersect with or may contain other objects. In GIS, the entities have a physical location and extent in some spatial region of interest and queries involve identification of these entities based on their spatial or aspatial attributes and spatial relationships. Spatial relationships are intersection, containment, adjacency etc. Intersection can be used to answer queries of the form "which roads intersect with B. T. Road ".

Conventional database management systems are not suited to the task of efficient evaluation of spatial relationships. Some sort of spatial indexing machanism must be supported. Without a spatial index, a query such as "Find all objects (e.g, lake) that are within a radius of 10 kms from I.S.I. " may require a search of the whole database. This will be grossly inefficient compared to the retrieving only objects in the neighbourhood of I.S.I.

One structure that has been proposed for spatial indexing structure, for efficient storage manipulation and the ease of information retrieval for non-zero sized objects is called spatial kd-tree (**skd-tree**) . The skd-tree structure avoids storing the object duplication

and supports two types of search, namely intersection and containment search.

Input : A global map , objects (represented by a region in 2D-dimensional space) on the map and a query region.

Output : Retrieving of all objects which intersect with or contained in the query region that satiesfies some geo-predicate.

To get output, first step is to store the geogragphic objects using skd-tree structure and later depending on search type retrieve all the objects within the query region.

Note that objects such as lake, road etc do not conform to any fixed shape. The methods of region decomposition and minimum bounding rectangles (MBR) are used to approximate irregularly shaped spatial objects. First we consider the objects as represented by isothetic minimum bounding rectangle. Later, we shall consider objects as represented by convex polygons.

2 DESCRIPTION OF THE SPATIAL SKD-TREE STRUCTURE

At each node of a skd-tree, a value (the discriminator value) is chosen in **one** of the dimensions to partition a k-dimensional space into two subspaces. The two resultant subspaces, HISON and LOSON, normally have almost the same number of data objects. Since a space is always divided into two, we require only one extra value for each subspace: the maximum of the objects in the LOSON subspace, and the minimum of the objects in the HISON subspace; along the dimension defined by the discriminator. Thus the structure of an internal node of the skd-tree consists of

1. Two child pointers;
2. A discriminator;
3. A discriminator-value;
4. The maximum value of upper-ordinate of objects in LOSON along the dimension specified by discriminator;
5. The minimum value of lower-ordinate of objects in HISON along the dimension specified by discriminator.

Hence, internal nodes are of the form

(disc, disc-val, loson-ptr, hison-ptr, max-loson, min-hison)

Where **disc** indicate the dimension that is being partitioned and **disc-val is the value** that partitions the space. The **max-loson** is the maximum range value of the LOSON subspace and the **min-hison** is the minimum range value of the HISON subspace along the dimension specified by the **disc. loson-ptr** and **hison-ptr** points to left and right child node respectively.

Leaf nodes are of the same form as the internal node. But in leaf nodes **hison-ptr** is the address of the datapage in which objects identifier and its minimum bounding rectangle stored. **min-hison** and **max-loson** are respectively the minimun and maximun range values of the objects in the datapage along the dimension specified by **disc**

3 DESCRIPTION OF THE DIFFERENT FUNCTIONS

Functions and Descriptions

ORDINATE (br,ty,dim) :

Arguments - br: bounding rectangle; ty, dim: int;
ty = 1 for upper and ty = 0 for lower ordinate of
the bounding rectangle along dimension dim (dim = 0
indicate X dimension, dim = 1 indicate Y dimension).

Value returned - br[2.dim + ty].

MAX (x,y) :

Arguments - x,y: float;

Value returned - the maximum value of x and y .

MIN (x,y) :

Arguments - x,y: float;

Value returned - the minimum value of x and y .

FIND_BOUND (ptr, count) :

Arguments - ptr : page_pointer; count : integer (no. of objects in the datapage);

purpose - to determine dimension that has least coverage; process is as below :

for all dimensions (i = 0 to 1) do

--- calculate the covering rectangle;

return that the dimension that have least coverage;

Value returned - 0 for X_dimension, 1 for Y_dimension.

CONTAINMENT (br1, br2) :

Arguments - br1, br2 : bounding rectangle;

purpose - to check if br1 contained in br2.

Value returned - 1 if rectangle br2 contains br1 otherwise 0.

The condition can be formulated as below:

ORDINATE (br1, 1, 0) <= ORDINATE (br2, 1, 0) and

ORDINATE (br1, 0, 0) >= ORDINATE (br2, 0, 0) and

ORDINATE (br1, 0, 1) >= ORDINATE (br2, 0, 1) and

ORDINATE (br1, 1, 1) <= ORDINATE (br2, 1, 1).

INTERSECT (br1, br2) :

```
Arguments - br1, br2 : bounding rectangle;
purpose - to check if two bounding rectangles intersect
          or not.
Value returned - 1 if rectangle br1 intersect with  br2
                 otherwise 0.
                 The condition can be formulated as below:
                 ORDINATE (br1, 0, 0) <= ORDINATE (br2, 1, 0) and
                 ORDINATE (br1, 0, 1) <= ORDINATE (br2, 1, 1) and
                 ORDINATE (br1, 1, 0) >= ORDINATE (br2, 0, 0) and
                 ORDINATE (br1, 1, 1) >= ORDINATE (br2, 0, 1).
```

MIN_RANGE (ptr, count, dim) :

```
Arguments - ptr : page-ptr;  dim : dimension;
            count : int (no. of objects in the datapage);
Value returned - the minimum range value along dimension,
                 dim of objects in the datapage to which ptr
                 is pointing.
```

MAX_RANGE (ptr, count, dim) :

```
Arguments - ptr : page-ptr;  dim : dimension;
            count : int (no. of objects in the datapage);
Value returned - the maximum range value along dimension
                 dim of objects in the datapage to which ptr
                 is pointing.
```

---

longest_dim_subspace (ptr) :

```
Argument - ptr : page_pointer;
purpose - to determine the longest dimension of the space
          which contains all the objects in the page ptr.
Value returned - 0 if X is the longest dimension of the
          space, otherwise 1 (i.e, Y is the longest
          dimension of the space).
```

---

bound_rec (poly, rect) :

```
Arguments - poly : convex_polygon;
            rect : isothetic_rectangle.
purpose - given the vertices of a convex polygon it deter-
          mines the isothetic minimum bounding rectangle
          of the polygon.
```

Value returned - isothetic minimum bounding rectangle thr-

ough argument rect.

CENTROID (mbr, dim) :

Arguments - mbr : isothetic minimum bounding rectangle

dim : int (0 for X_dimension, 1 for Y_dimension);

Value returned - centroid of the mbr along dimension dim.

---

distance (pt1, pt2) :

Arguments - pt1, pt2 : point;

Value returned - distance between two points.

---

area_triangle (pt1, pt2, p3) :

Arguments - pt1, pt2, pt3 : point;

Purpose - to find the area of a triangle formed by three points.

This is done by finding the distance of three sides of

the triangle and using the result,

area = s(s - a)(s -b)(s - c).

Value returned -area of the triangle.

seg_intersect (pseg, qseg, pt_intersect) :

Input : pseg - array of two terminal points of the edge
of a pentagon;
: qseg - array of two terminal points of the edge
of a query rectangle.

Value returned - return 0 if segments does not intersect,
otherwise return 1 and the point of intersection
through the argument pt_intersect.

Purpose - Function for checking whether edge of a pentagon
and edge of a query rectangle intersect or not.
if they intersect, obtain the point of inters-
ection. The procedure is as follows :
if the segments are parallel then return 0;
else find the point of intersection;
if the point of intersection cuts internally
to both the segments then set pt_intersect
and return 1;
otherwise Return 0;

---

4 ALGORITHMS AND RESULTS FOR QUERY SOLVING WHEN OBJECTS ARE ISOTHETIC RECTANGLE

Here we represent object and query region each as an isothetic rectangle. An input file which contains record of each object consisting of object identifier and two corner vertices. A query retrieves either all objects which intersect with or all objects contained in the query rectangle. First we store the objects using skd-tree. Leaf node of the tree points to the datapage. Each datapage can hold a fixed number of objects. The structure of each entity in the datapage is of the following form

(obj_id, lo_x, up_x, lo_y, up_y)

where **obj_id** is the object identifier. **lo_x** and **up_x** are the lower and upper bounds of X_ordinate. **lo_y** and **up_y** are the lower and upper bounds of Y_ordinate.

Search starts from the root of the tree. The algorithm search and some other required algorithms described below :

4.1 Main_program

```
Let fptr is the input file pointer.
read first record from input file.
create a node say, root;
```

```
create a datapage;
put the record into datapage;
/* initially root is the leaf node */
root.hison_ptr points to datapage;
root.loson_ptr = NULL;
root.disc_val = 1;
Calculate dim, is the shortest side of the object;
set root.disc to dim;
set root.max_loson and root.min_hison to maximum
and minimum  value of the the object along dim
respectively.
while not EOF do
{
Read the next record from input file, Store it in obje;
INSERT (obje, root);
}
Read the mapspace say, br (i,e, the rectangle
in which all objects lies);
Read the query rectangle say  query_region;
read search type say , search_ty;
SEARCH (root, br);
close input file;
end_main;
```

4.2 ALGORITHM Search

```
Input  :  node - an intermediate or a leaf node;
          initially it is the root.
       :  br - the subspace of the current node;
          initially the map space.
Output :  A list of objects.
Comment : Traverse the tree from the root. The process
          of searching is done recursively.
          search_ty - search type (containment or
          intersection).
          query_region - the query region which is
          isothetic rectangle.
          lo_br, hi_br - bounding rectangles used to
          define LOSON and HISON subspaces.
```

SEARCH (node, br)

```
if node is a leaf node then
  {
    ORDINATE (br, 0, node.disc) = node.min-hison;
    ORDINATE (br, 1, node.disc) = node.max-loson;
    if ( INTERSECT (br, query_region)) then
         CHECK_LEAF (node.hison_ptr, node.disc_val);
/* here node.disc-val contains no. of objects in the datapage */
```

```
    return;
   }
hi_br = br;
lo_br = br;
/* rectangles for HISON and LOSON subspaces */
if search_ty is containment search then
   {
    ORDINATE (lo_br, 1, node.disc) =
         MIN (node.disc_val, node.max_loson);
    ORDINATE (hi_br, 0, node.disc) =
         MAX (node.disc_val, node.min_hison);
   }
else /* an intersection search */
   {
   ORDINATE (lo_br, 1, node.disc) = node.max_loson;
   ORDINATE (hi_br, 0, node.disc) = node.min_hison;
   }
if ( INTERSECT (lo_br,query_region)) then
         SEARCH (node.loson_ptr, lo_br);
if ( INTERSECT (hi_br,query_region)) then
         SEARCH (node.hison_ptr, hi_br);
end_SEARCH.
```

4.3 ALGORITHM Insert

New records are added to the data page, and the page is split if it overflows. As the tree is traversed, the algorithm determines the branching direction of each node and updates the node if the minimum bounding rectangle(MBR) of the object extends over the node boundary. On reaching to a leaf node, the data page is fetched and insertion may be performed.

```
Input   : node - an intermediate node or a leaf node;
          initially the root.
        : obj - an isothetic rectangular object to be
          inserted.
Output  : the updated skd-tree.
Comment : Use the centroid to determine the place of an
          object. A split occurs if the page overflows.
```

INSERT (obj, node)

```
if node is a leaf node then
   {
   fetch the data page address by node.hison_ptr;
if node.disc_val is < page_size then
    {
    insert the record into the data page;
    add 1 to node.disc_val;
```

```
dim = FIND_BOUND (node.hison_ptr, node.disc_val);
if (dim == node.disc) then
   {
   node.min_hison =
    MIN(ORDINATE(obj, 0, dim), node.min_hison);
   node.max_loson =
    MAX(ORDINATE(obj, 1, dim), node.max_loson);
   }
  else  /* new boundaries in the leaf node  */
   {
   node.disc = dim ;
   node.min_hison =
   MIN_RANGE(node.hison_ptr, node.disc_val, dim);
   node.max_loson =
   MAX_RANGE(node.hison_ptr, node.disc_val, dim);
   }
else
   {
   create a new internal node say, in_root;
   split (node, in_root);
   INSERT (obj, in_root);
   } return;
}
```

```
/* for internal node  */
if ( CENTROID ( obj, node) < node.disc_val ) then
   {
   /* the centroid is in the LOSON subspace  */
   node.max_loson =
   MAX(ORDINATE(obj, 1, node.disc), node.max_loson);
   INSERT (obj, node.loson_ptr);
   }
else  /* the centroid is in the HISON subspace  */
   {
   node.min_hison =
   MIN(ORDINATE(obj, 0, node.disc), node.min_hison);
   INSERT (obj, node.hison_ptr);
   }
end_INSERT.
```

4.4 ALGORITHM Split

Input : node - the leaf node whose data page is to be split.

Output : an internal node which is pointing to two leaf nodes.

Comment : Data page to be split if there is no room for inserting another object.

```
                In_root is a internal node which will be created.

split (node, in_root)

Get the dimension dim =
                longest_dim_subspace (node.hison_ptr);
for all objects of the page node.hison_ptr
                                (i = 0 to page_size - 1) do
calculate centroid[i] =CENTROID(object[i], dim);
Order the objects in the data page node.hison_ptr
in the ascending order of their centroid along
dimension dim;
Calculate disc_value be the Halfway between object
m and m + 1 along dimension  dim;
Partition the objects comparing their centroid
along dim with disc_value;
set in_root.disc_val to disc_value;
Create two new datapages and redistribute objects
of the old datapage into new datapages by comparing
centroid of the objects with in_root.disc_val;
Create two new leaf nodes;
Father of the old leaf node will point to the in_root;
delete old leaf node and also old datapage;
Link in_root.loson_ptr and in_root.hison_ptr to
left_leaf node and right_leaf node respectively;
```

```
Set all other fields of internal node and leaf
nodes as done in insertion algorithm;
return;
end_split.
```

4.5 ALGORITHM Check_leaf

```
Input  : node - page_ptr;
         count - integer(no. of objects in the datapage);
         search_ty : char ( search_ty = 'i' for intersection
         search, and  = 'c' for containment search );

Output : get all objects in the data page which contained
         in / intersect with the query rectangle.
Comment : Query rectangle passed globally to the function;
          each object in the datapage checked whether it
          intersects or is contained in the query region.
```

CHECK_LEAF (node, count)

```
temp : array of object;
/* temp assigned to the datapage to which node
   is pointing. */
if (search_ty == 'i') then {
```

```
for all entities in the datapage(j = 0 to count) do
 if (INTERSECT (temp[j], query_region)) then display
    temp[j].object_name;
 return;
}
if (search_ty == 'c') then
   {
   for all entities in the datapage(j = 0 to count) do
   if ( CONTAINMENT (temp[j], query_region)) then
   display temp[j].object_name;
     return;
   }
end_CHECK_LEAF.
```

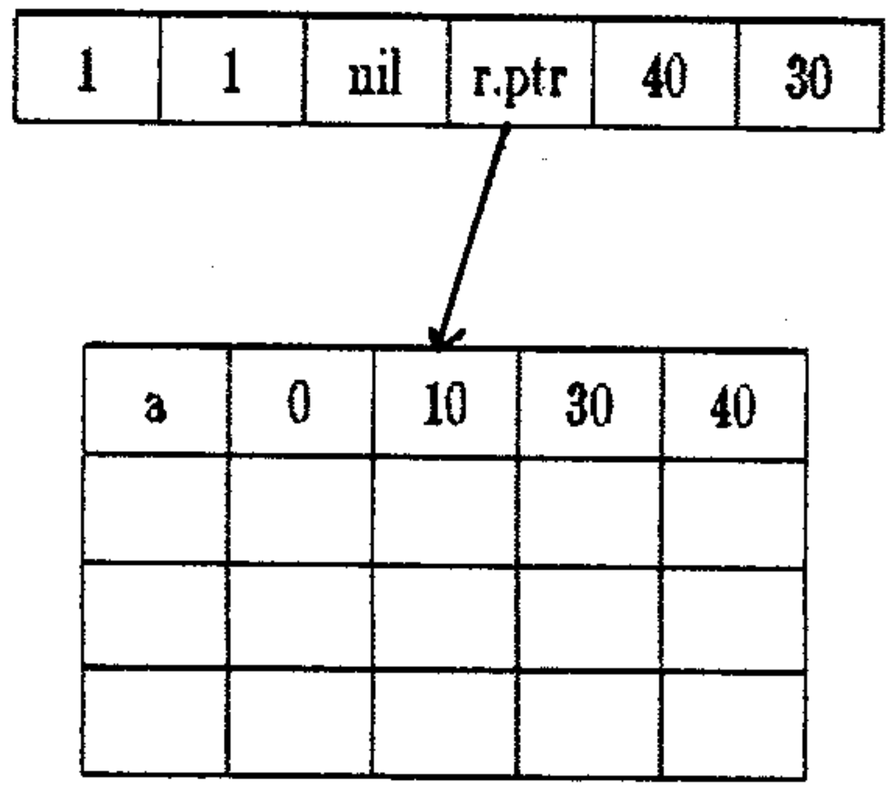
4.6 RESULTS :

Table - 1.1 and Fig - 1.1 in the last section shows the record of objects and geographical representation of the objects. With the the records in the Table - 1.1, the steps of tree construction during insertion are shown below :

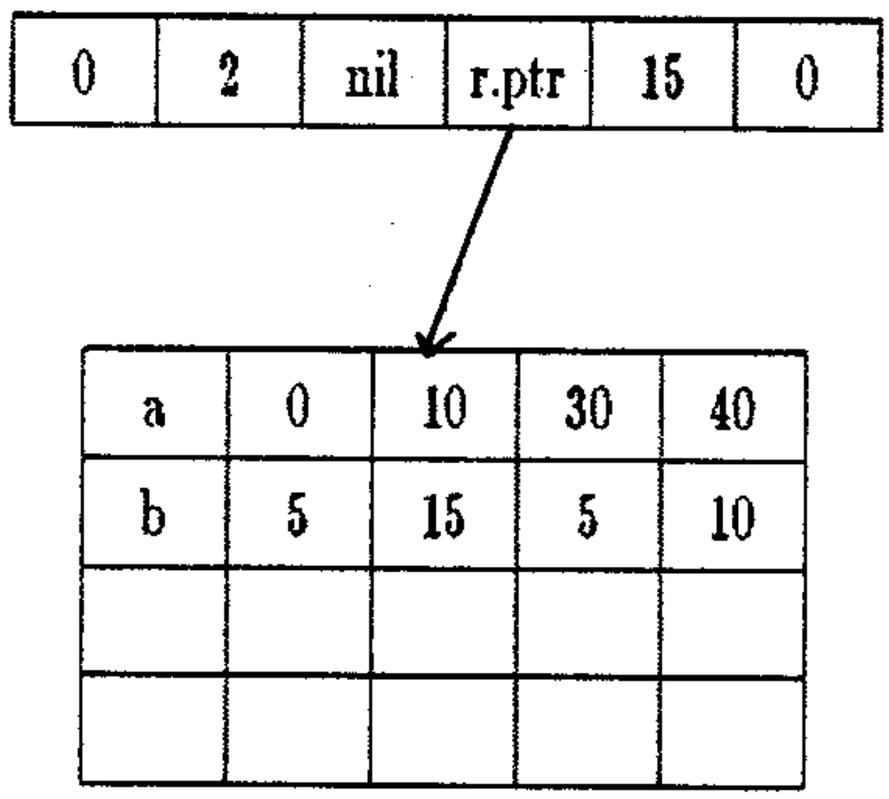
Each node has following structure

disc	*disc-val*	*l.ptr*	*r.ptr*	*l.max*	r.min

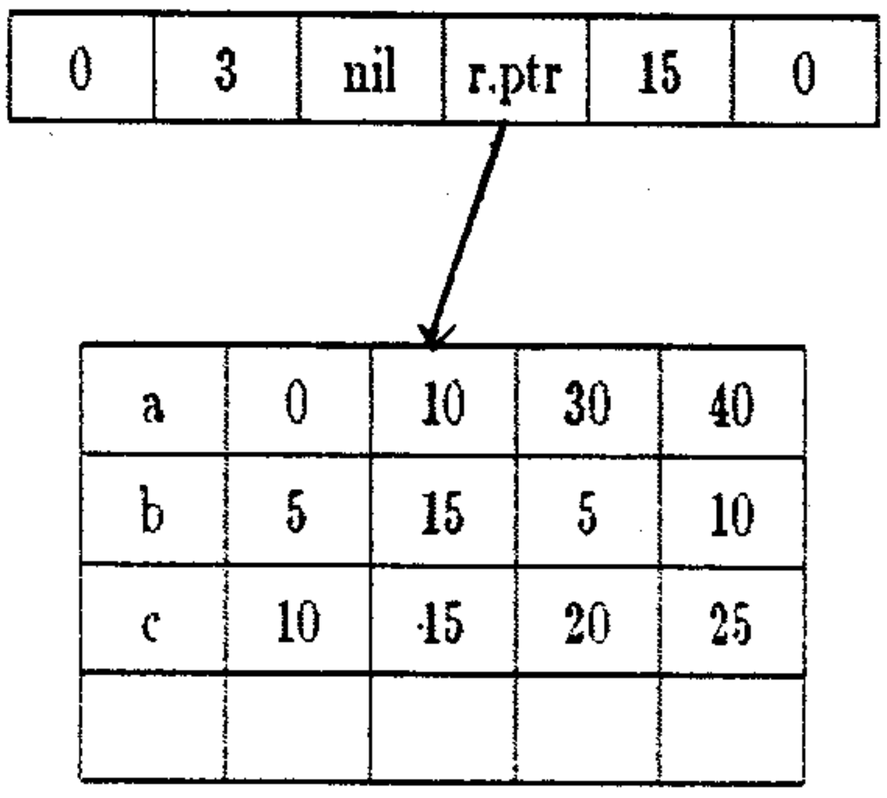
Step1 :



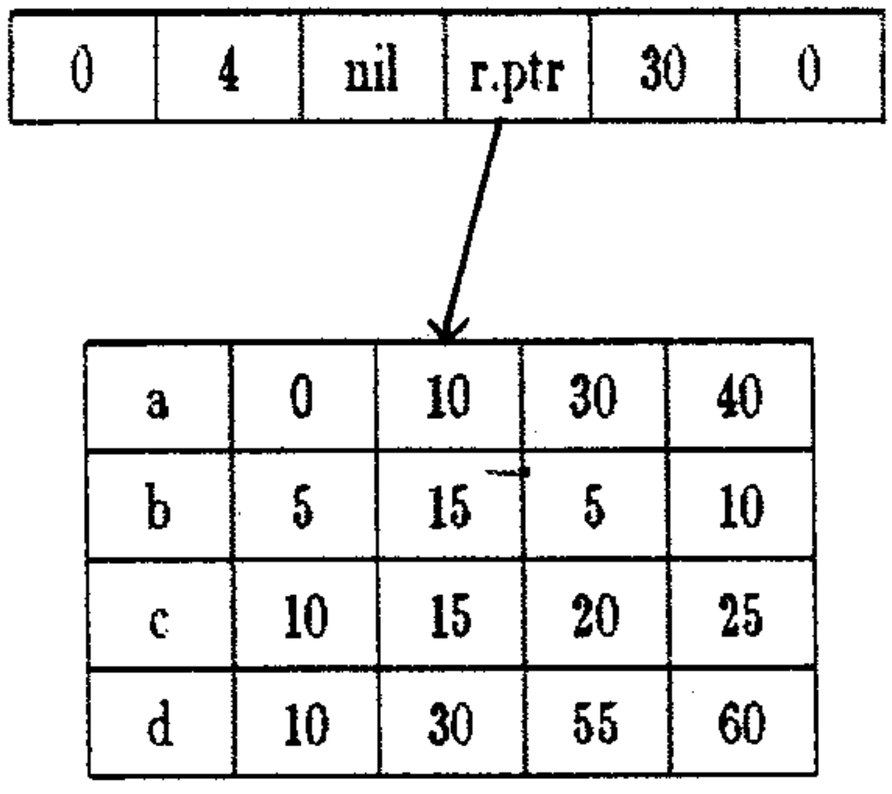
Step2 :



Step3 :



Step4:



0	4	nil	r.ptr	30	0

a	0	10	30	40
b	5	15	5	10
c	10	15	20	25
d	10	30	55	60

Step5:

1	28.75	l.ptr	r.ptr	25	30

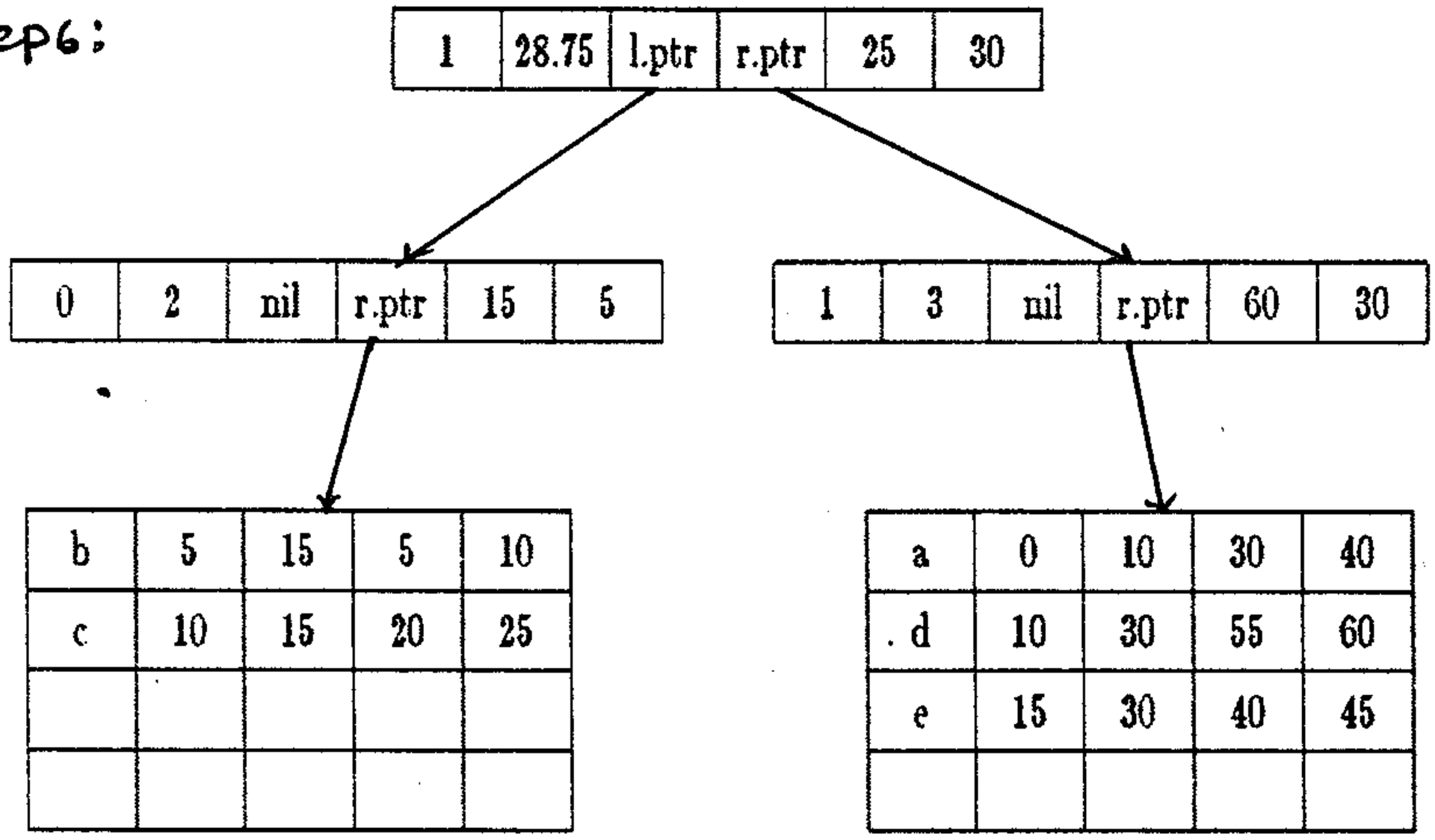
0	2	nil	r.ptr	15	5

0	2	nil	r.ptr	30	0

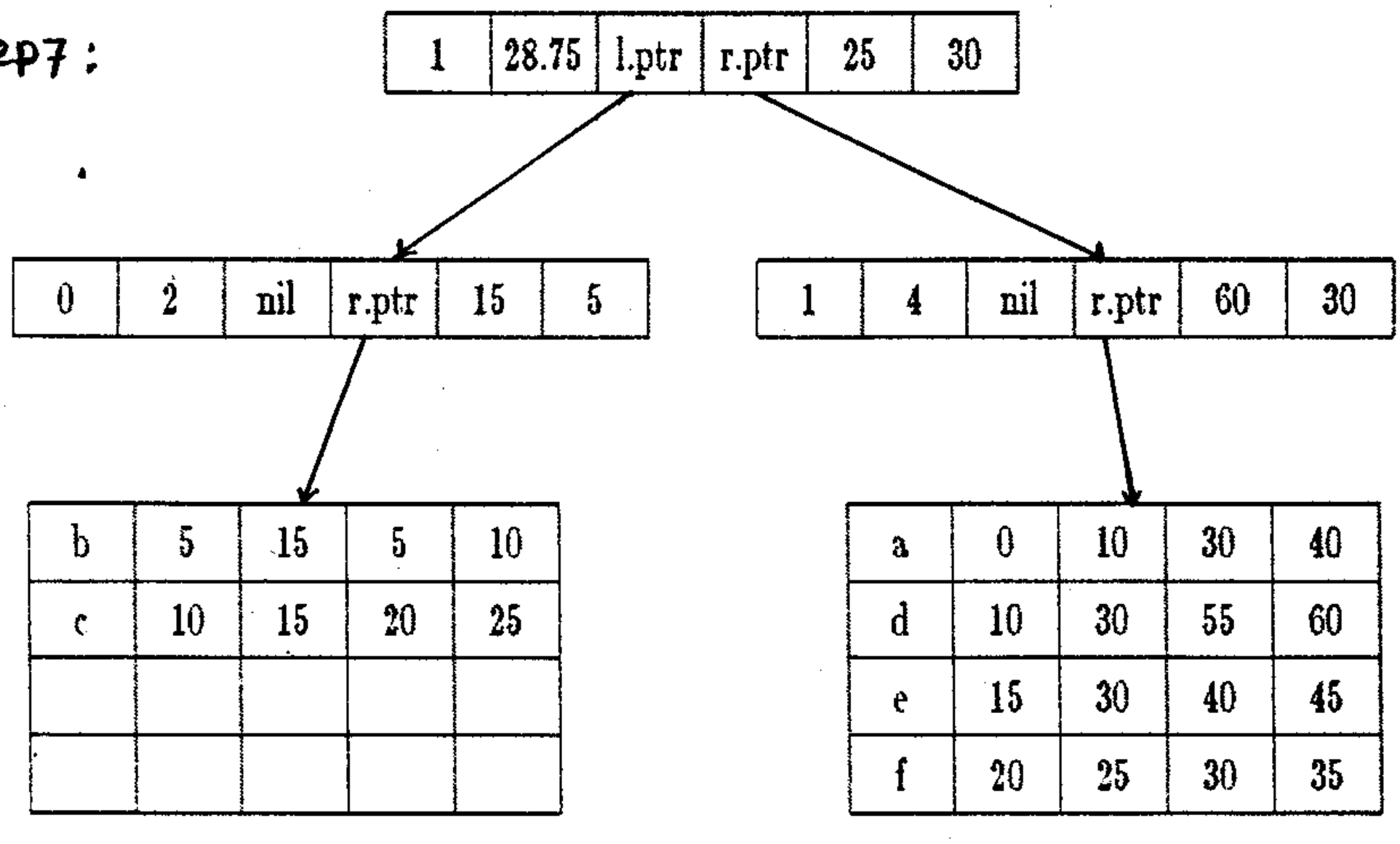
b	5	15	5	10
c	10	15	20	25

a	0	10	30	40
d	10	30	55	60

Step6:



Step7:



And So on.....

The result of the query search shown below :

```
Input lower bound of x for map space :   -1
Inputf upper bound of x for map space :   70
Input lower bound of y for map space :   -1
Input upper bound of y for map space :   70

RESULT OF INTERSECTION SEARCH :
-----------------------------------------

Input lower bound of x for query region :  8
Input upper bound of x for query region : 42
Input lower bound of y for query region :  8
Input upper bound of y for query region : 32

Input c for containment search  otherwise,
Input i for intersection search :  i

Query region intersects with objects whose
names are :     b, c, g, h, j, a, f, k
```

```
RESULT OF CONTAINMENT SEARCH :
-----------------------------------------

Input lower bound of x for query region : 12

Input upper bound of x for query region : 42

Input lower bound of y for query region :  8

Input upper bound of y for query region : 42


Input c for containment search  otherwise ,

input i for intersection search :  c


Query region contains objects whose names

are  :       g, f, k
```

5 ALGORITHMS AND RESULTS FOR QUERY SOLVING WHEN OBJECTS ARE CONVEX POLYGONAL SHAPED.

Here we represent each object as a convex pentagon and query region is an isothetic rectangle. An input file which contains record of each object consisting of object identifier and five vertices of the pentagon. Query is to retrieve all objects which intersect with or contained in the query rectangle and also to obtain the region and area of intersection. We shall store the objects using skd-tree. Leaf node of the tree points to the datapage. Each datapage can hold a fixed number of object. The structure of each entity in the datapage is of the following form

(poly (obj_id, v1, v2, v3, v4, v5); obou(lo_x, up_x; lo_y, up_y))

where **obj_id** is the object identifier. **vi**(x, y), is the ith vertex of a pentagon. **lo_x** and **up_x** are the lower and upper bounds of X_ordinate of its isothetic MBR. **lo_y** and **up_y** are the lower and upper bounds of Y_ordinate of its isothetic MBR.

Here we consider objects which are represented by a convex pentagon. Necessary algorithms which are used for solving queries are described below:

5.1 M_Main_program

```
Let fptr,optr are the input and output file pointers
respectively;
```

```
read first record from input file say, poly;
create a node say, root;
create a datapage;
call bound_rec(polygon, bounding_rectangle);
put the  polygon and its bounding_rectangle into
datapage;    /* initially root is the leaf node */
root.hison_ptr points to datapage;
root.loson_ptr = NULL;
root.disc_val = 1;
Calculate dim, is the shortest side of the
bounding_rectangle;
set root.disc to dim;
set root.max_loson and root.min_hison to maximum and minimum
value of the the bounding_rectangle along dim respectively.
while not EOF reached do
{
Read the next record from input file, Store it in polygon;
call bound_rec(polygon, bounding_rectangle);
M_INSERT (polygon, bounding_rectangle, root);
}
Read the mapspace say, br (i,e, the rectangle in which
all objects lies);
Read the query rectangle say, query_region;
```

```
read search type say , search_ty;
M_SEARCH (root, br);
close input and output file;
end_main;
```

5.2 ALGORITHM M_Search

Algorithm M_Search is same as algorithm Search which is described previously. Only difference is the CHECK_LEAF function which is replaced by M_CHECK_LEAF function.

5.3 ALGORITHM m_insert

New records are added to the data page, and the page is split if it overflows. As the tree is traversed, the algorithm determines the branching direction of each node and updates the node if the minimum bounding rectangle(MBR) of the object extends over the node boundary. On reaching to a leaf node, the data page is fetched and insertion may be performed.

```
Input   : node - an intermediate node or a leaf node;
                 initially the root.
        : polygon - the object which is a convex polygon;
        : mbr - isothetic minimum bounding rectangle  of
          the polygon to be inserted.
Output  : the updated skd-tree.
```

Comment : Use the centroid of mbr to determine the place of a polygonal object.

A split occurs if the page overflows.

```
M_INSERT (polygon, mbr, node)
if node is a leaf node then
   {
   fetch the data page address by node.hison_ptr;
   if node.disc_val is < page_size then
       {
       insert the polygon and mbr into the datapage;
       add 1 to node.disc_val;
dim = FIND_BOUND (node.hison_ptr, node.disc_val);
/* FIND_BOUND uses only mbr of the polygon */
if (dim == node.disc) then  {
   node.min_hison =
       MIN (ORDINATE(mbr, 0, dim), node.min_hison);
   node.max_loson =
       MAX (ORDINATE(mbr, 1, dim), node.max_loson);
   }
else {      /* new boundaries in the leaf node  */
   node.disc = dim ;
```

```
            node.min_hison =
                MIN_RANGE(node.hison_ptr, node.disc_val, dim);
            node.max_loson =
                MAX_RANGE(node.hison_ptr, node.disc_val, dim);
        /* MAX_RANGE and MIN_RANGE uses only mbr of the polygon  */
            }
        else
            {
            create a new internal node say, in_root;
            m_split(node, in_root);
            M_INSERT(polygon, mbr, in_root);
            }
            return;
          }
        /* for internal node  */
        if ( CENTROID( mbr, node) < node.disc_val ) then
           {
           /* the centroid is in the LOSON subspace  */
           node.max_loson =
             MAX (ORDINATE(mbr, 1, node.disc), node.max_loson);
           M_INSERT (polygon, mbr, node.loson_ptr);
           }
```

```
else {    /* the centroid is in the HISON subspace  */
  node.min_hison =
     MIN (ORDINATE(mbr, 0, node.disc), node.min_hison);
 N_INSERT (polygon, mbr, node.hison_ptr);
  }
end N_INSERT.
```

5.4 ALGORITHM M_Split

```
Input  : node - the leaf node whose datapage is to
         be split.
Output : an internal node which is pointing to two
         leaf nodes.
Comment : Data page to be split if there is no room for
          inserting another object. In_root is a internal
          node which will be created.

m_split(node, in_root)
Get the dimension dim =
                  longest_dim_subspace(node.hison_ptr);
/* to determine longest dimension, use mbr of the objects */
for all objects of the page node.hison_ptr
                                (i = 0 to page_size - 1) do
calculate centroid[i] = CENTROID (object[i].mbr, dim);
```

```
Order the objects in the datapage node.hison_ptr in the
ascending order of their centroid along dimension dim;
Calculate disc_value from objects' mbr be the Halfway
between object m and m + 1 along dimension  dim;
Partition the objects comparing their centroid of mbr
along dim with disc_value;
set in_root.disc_val to disc_value;
Create two new datapages and redistribute objects of the
old datapage into new datapages by comparing centroid of
the mbr with in_root.disc_val;
Create two new leaf nodes;
Father of the old leaf node will point to the in_root;
delete old leaf node and also old datapage;
Link in_root.loson_ptr and in_root.hison_ptr to left_leaf
node and right_leaf node respectively;
Set all other fields of internal node and leaf nodes
as done in insertion algorithm;
return;
end m_split.
```

5.5 ALGORITHM M_Check_leaf

```
Input  : node - page_ptr;
         count - integer ( no. of objects in the datapage);
         search_ty : char ( search_ty = 'i' for intersection
         search,  and  = 'c' for containment search );
Output : get all objects in the data page which  contained
         in / intersect with the query rectangle. Obtain
         the region of intersection and its area.
Comment : Query rectangle and optr passed globally to
          the function.
          optr is the file pointer where intersection
          and area stored.
```

```
M_CHECK_LEAF(node, count)
temp : array of object and its MBR;
/* temp assigned to the datapage to which node is pointing. */
if (search_ty == 'i') then
   {
   for all entities in the datapage (j = 0 to count ) do
polygon_query_intersection(temp[j].poly, query_region,optr);
 return;
}
```

```
if (search_ty == 'c') then
   {
   for all entities in the datapage (j = 0 to count ) do
    if (CONTAINMENT (temp[j].obou, query_region)) then
    polygon_query_intersection(temp[j].poly, query_region,optr);
   return;
   }
end M_CHECK_LEAF.
```

5.6 AGORITHM Polygon_query_intersection

Algorithm for determining the vertices of intersection of a convex pentagon & query rectangle and also the area of the intersecting region.

```
Input : q_rect - is  a query rectangle;
        poly1 -is a convex pentagon with identifier and
        vertices.
        optr - is a pointer to the output file in which
        intersecting polygon & its area stored.
Output : A file of record which consist of intersecting
         polygon and its area.
Comment :  The process is as below :
           all the vertices of the pentagon which are inside
           the query region, stored into an array.
```

If a corner vertex of the query region is inside the pentagon, are also stored in the array. Also store all the intersecting points of pentagon and query rectangle into that array. Get the centroid of the points in the array. Determine the polar angle of the vertices in the array from centroid. Sort the points w.r.t their polar angle. Finally, we shall get the vertices of the intersection. To get the area of intersection , we draw the triangles which are formed by centroid and other two consecutive vertices of intersection. Adding the area of each triangle we get the area of the intersection.

```
polygon_query_intersection (poly1, q_rect, optr)
poly :  array of the vertex of the pentagon;
temp :  array of the vertex of the q_rect;
q_count, a_count :  integer;
arr :  array of vertex;
a_count = q_count = 0;
store the vertices(x,y) of poly1 into poly;
for all vertices in poly ( i = 0 to 5 ) do
if(point_check_in_qregion (poly[i].x, poly[i].y, q_rect)) then
  /* vertex is inside the query rectangle.  */
```

```
        {
        arr[a_count] = poly[i];
        add 1 to a_count;
        }
if ( a_count == 5 ) then
   /* All vertices of the pentagon are inside the query
      rectangle. So, pentagon itself is the intersection. */
   {
   area_of_intersection =  find_area (arr, a_count);
   put the vertices of intersection & area_of_intersection
   into output file through optr;
   return;
  }
   store vertices of q_rect to temp;
   for all vertices in temp (i = 0 to 4) do
     if (point_check_in_poly (temp[i].x,temp[i].y, poly)) then
        {     /*  vertex  is inside the pentagon.  */
        arr[a_count] = temp[i];
        add 1 to each a_count and q_count;
        }
   if ( q_count == 4 ) then
    /* All vertices of the query region are inside the
     pentagon. So, query region itself is the intersection. */
```

```
{
area_of_intersection  =  find_area (arr, q_count);
Put the vertices of intersection & area_of_intersection
into output file through optr;
return;
   }
for each edge of the pentagon do
{
  for each edge of the query_region do
   if(seg_intersect (poly_edge, query_edge,intersect_pt)) then
   {
   store intersect_pt into arr;
   add 1 to a_count;
   }
}
/* Now arr contains all the vertices of the intersection
 but, vertices are not ordered. Also arr may contain
 duplicate points  */
if ( a_count < 3 ) then return;
/* Because intersection of polygon and query region
 should be at least a triangle  */
Let centre(x,y) be the centre of the intersection obtained
by averaging all the vertices of arr;
```

```
Calculate the polar angle of each vertex of intersection
from centre;
Order the vertices of intersection by ordering polar angles;
Remove all the duplicate vertices having same polar angle.
Hence get an array say, final_array of ordered vertices of
intersection and no. of vertices say, final_count;
/* Next we calculate the area of intersection */
area_of_intersection = find_area(final_array, final_count);
put final_array and area_of_intersection into output file;
return;
end_polygon_query_intersection.
```

5.7 ALGORITHM Point_check_in_qregion

Algorithm for checking a point whether it is inside of an isothetic query rectangle or not.

```
Input : pt(x,y) is a point;
        q_rect is a query rectangle;
Output : if pt(x,y) inside the query region then return 1,
           else return 0;
point_check_in_qregion (pt.x, pt.y, q_rect)
if ((q_rect[0] < pt.x) and (q_rect[1] > pt.x) and
   (q_rect[2] < pt.y) and (q_rect[3] > pt.y)) then return (1);
```

```
else    return (0);
   /*  point outside of the query_rectangle */
end_point_check_in_qregion .
```

5.8 AGORITHM Point_check_in_poly

Algorithm for checking a point whether it is inside of a convex pentagon or not.

```
Input  : pt(x,y) is a point;
         poly is an array of vertices of a convex pentagon;
Output : if pt(x,y) inside the pentagon then  return 1,
         else return 0;
Comment : procedure is as follows :
```

Consider a line passes through the given point pt(x,y) and parallel to the x_axis. Let the line cuts at j number of points to the edges of the pentagon. Number of cut_points to the right of the given point pt(x,y) is rsol_count. If j = 0, point pt(x,y) is outside of the pentagon. Value of j will be atmost 2. If rsol_count is 1, then the point inside the pentagon, otherwise outside of the pentagon.

```
point_check_in_poly (pt.x, pt.y, poly)
k, rsol_count, sol_count : integer;
pt1, pt2, sol, sol1[2]   : point;
Initialize rsol_count, sol_count to zero;
for all edges of the pentagon (i = 0 to 4 ) do  {
    pt1 = poly[i];
    k = (i + 1) mod 5 ;
    pt2 = poly[k];
    if (pt1.y == pt2.y) then go to end of the for_loop ;
     /* edge of the pentagon parallel to the x_axis.  */
     else
       if ( pt1.x == pt2.x ) then  {
         /* edge of the pentagon perpendicular to the x_axis. */
            sol.x = pt1.x;
            sol.y = pt.y;
            }
   else  {
     /* edge of the pentagon cuts the line which passes
     through the given point and parallel to the x_axis. */
        sol.y = pt.y;
        sol.x = pt1.x + ((pt2.x - pt1.x)*(sol.y - pt1.y)) /
            (pt2.y - pt1.y);
   }
```

```
d1 = distance (pt1, sol);
d2 = distance (pt2, sol);
d = distance (pt1, pt2);
if ( | d - d1 - d2 | is less then very small quantity ) then
   {
    /* Line which passes through the given point &
    parallel to the x_axis cuts internally to the
    edge segment of the pentagon. */
    store sol to sol1[sol_count];
    add 1 to sol_count;
    if ( sol_count == 2) then go out of the for_loop;
/* Since line which is parallel to x_axis & passes
   through the given point cuts atmost two times to
     the edges of the pentagon */
 } /* end of forloop */
if (sol_count == 0) then return(0);
 /* point is outside of the loop */
Calculate rsol_count, the number of cut points to the
right of the given point;
if ( rsol_count == 1) then return(1);
/* point is inside the pentagon */
end_point_check_in_poly.
```

5.9 ALGORITHM Find_area

Algorithm for finding area of a convex polygon.

```
Input  : arr - array of vertex(x, y) of a convex polygon
       : v_count - no. of vertices of the polygon.
Output : area of a polygon.
Comment  : Area of a convex polygon can be found by
           adding area of triangles which are formed
           by three points, centre & two consecutive
           vertices of the polygon.

find_area (arr, v_count)
int i, k;
Initialize centre.x = centre.y = 0;
sum = 0;
for all vertices ( i = 0 to v_count - 1) do
 {
  centre.x = centre.x + arr[i].x;
  centre.y = centre.y + arr[i].y;
 }
centre.x = centre.x / v_count;
centre.y = centre.y / v_count;
```

```
for (i = 0 to v_count - 1) do
 {
  point1 = arr[i]; k = (i + 1) mod v_count;
  point2 = arr[k];
  sum = sum + area_triangle (centre, point1, point2);
  }
return (sum);
end_find_area.
```

5.10 RESULTS

Table - 2.1 and Fig - 2.1 in the last section shows the record of objects and geographical represention of objects. With the records shown in Table - 2.1 the results of the query are shown below :

```
Input lower bound of x for map space :   -1
Input upper bound of x for map space :   70
Input lower bound of y for map space :   -1
Input upper bound of y for map space :   70
```

RESULT OF INTERSECTION SEARCH :

Search type = i

QUERY REGION : Lower bound of X = 12.00

Upper bound of X = 42.00

Lower bound of Y = 8.00

Upper bound of Y = 32.00

The object-name, vertices of the intersection between query rectangle and object, area of intersection are shown below :

a (12.0 11.0) (12.00 8.0) (14.0 8.0) Area = 3.000

b (25.0 27.0) (15.0 24.0) (12.0 22.8) (12.0 17.9)
(15.0 17.0) (22.0 20.0) Area = 72.850

c (19.0 32.0) (12.0 32.0) (12.0 27.7) (16.0 29.0) Area = 19.167

d (32.0 15.0) (27.0 15.0) (23.0 11.0) (27.8 8.0)
(33.0 8.0) (36.0 11.0) Area = 63.300

e (42.0 24.3) (40.0 27.0) (36.0 28.0) (39.0 18.0) (42.0 17.5)
Area = 39.083

m (42.0 32.0) (38.5 32.0) (40.0 31.0) Area = 1.750

RESULT OF CONTAINMENT SEARCH :

Search type = c

QUERY REGION :

Lower bound of X = 10.00

Upper bound of X = 47.00

Lower bound of Y = 4.00

Upper bound of Y = 40.00

The object-name, vertices of the intersection between query rectangle and object, area of intersection are shown below :

c (21.0 34.0) (15.0 35.0) (10.0 33.0) (10.0 27.0)
(16.0 29.0) Area = 51.500

d (23.0 11.0) (31.0 6.0) (36.0 11.0) (32.0 15.0)
(27.0 15.0) Area = 68.500

e (39.0 18.0) (45.0 17.0) (43.0 23.0) (40.0 27.0)
(36.0 28.0) Area = 51.000

m (37.0 33.0) (40.0 31.0) (42.0 32.0) (42.0 35.0)
(39.0 37.0) Area = 19.000

6 Tables and Figures

Table - 1.1

Table showing the object name and its lower left-corner & upper right-corner vertices.

object name	lower left-corner vertex		upper right-corner vertex	
a	0.00	30.00	10.00	40.00
b	5.00	5.00	15.00	10.00
c	10.00	20.00	15.00	25.00
d	10.00	55.00	30.00	60.00
e	15.00	40.00	30.00	45.00
f	20.00	30.00	25.00	35.00
g	25.00	15.00	30.00	25.00
h	20.00	0.00	40.00	10.00
i	50.00	10.00	60.00	15.00
j	40.00	20.00	50.00	25.00
k	35.00	30.00	40.00	40.00
l	40.00	50.00	45.00	55.00
m	55.00	35.00	60.00	45.00
n	45.00	35.00	50.00	45.00

Table - 2.1

Table showing the object name and its vertices

object	vertex									
name	(1)		(2)		(3)		(4)		(5)	
a	5.0	5.0	11.0	5.0	14.0	8.0	12.0	11.0	7.0	10.0
b	5.0	20.0	15.0	17.0	22.0	20.0	25.0	27.0	15.0	24.0
c	10.0	27.0	16.0	29.0	21.0	34.0	15.0	35.0	10.0	33.0
d	23.0	11.0	31.0	6.0	36.0	11.0	32.0	15.0	27.0	15.0
e	39.0	18.0	45.0	17.0	43.0	23.0	40.0	27.0	36.0	28.0
f	46.0	30.0	50.0	27.0	59.0	29.0	56.0	33.0	51.0	32.0
g	25.0	40.0	27.0	36.0	31.0	33.0	35.0	40.0	31.0	43.0
h	40.0	45.0	41.0	43.0	47.0	47.0	49.0	51.0	45.0	50.0
i	45.0	38.0	50.0	39.0	53.0	42.0	50.0	44.0	47.0	42.0
j	20.0	44.0	27.0	44.0	33.0	46.0	35.0	52.0	25.0	52.0
k	15.0	49.0	16.0	46.0	19.0	47.0	19.0	50.0	17.0	52.0
l	44.0	9.0	50.0	7.0	53.0	12.0	56.0	20.0	49.0	22.0
m	37.0	33.0	40.0	31.0	42.0	32.0	42.0	35.0	39.0	37.0

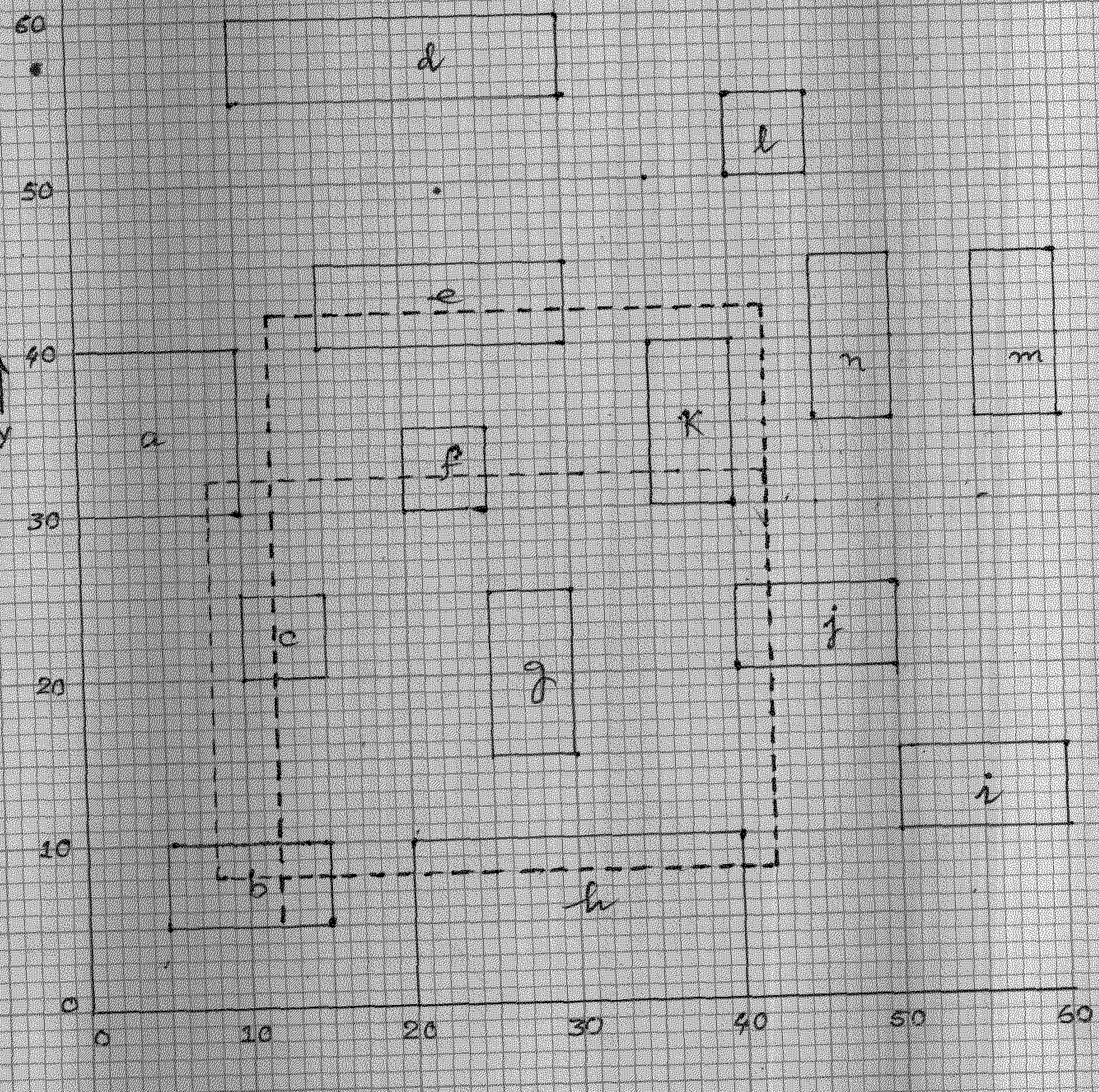
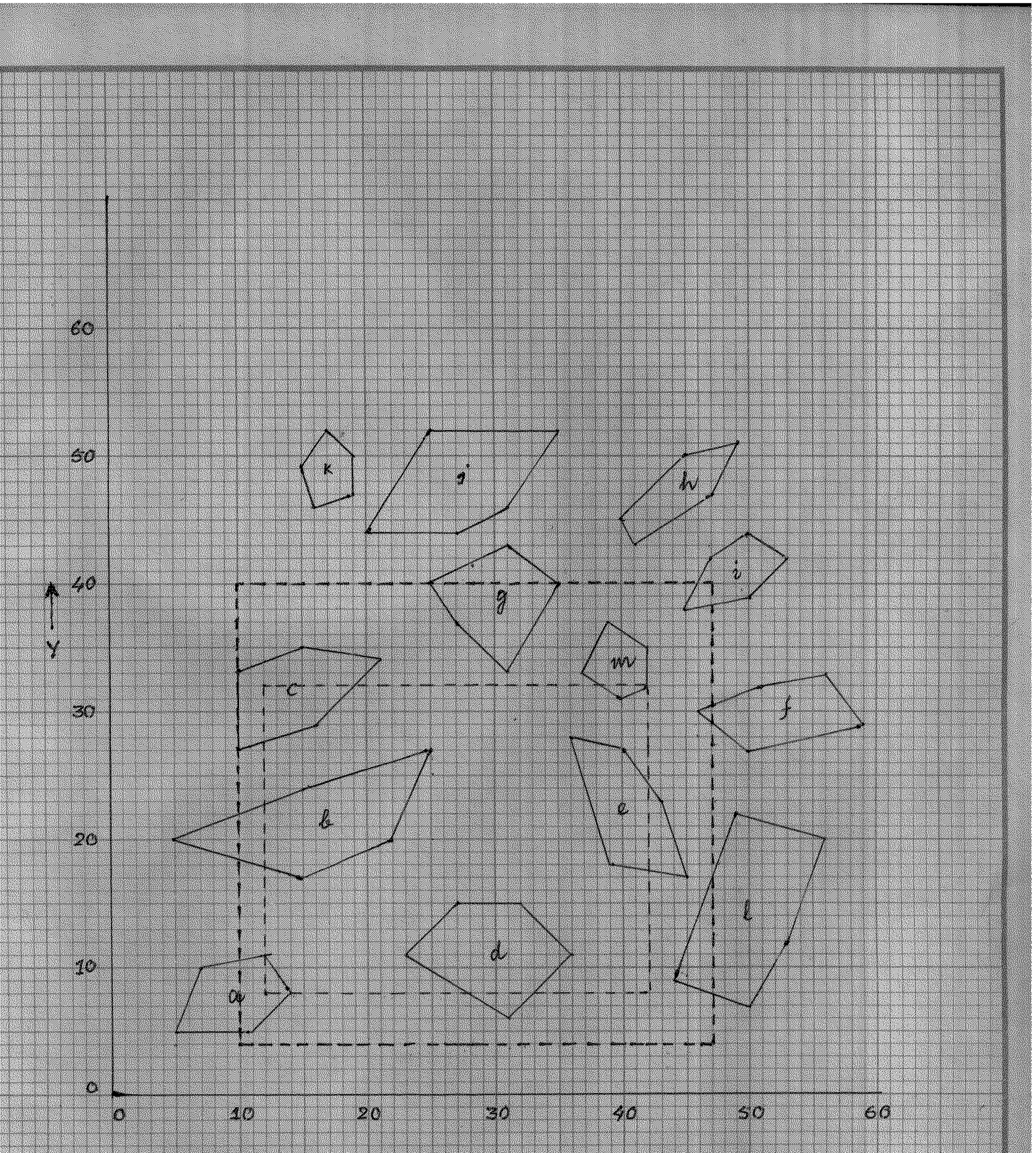


Fig- 1.1



Query Rectangle for Intersection Search.

Query Rectangle for Containment Search.

Fig - 2.1

7 REFERENCES

1. Efficient Query processing in Geographic Information Systems [Springer-Verlag ,1990, (Lecture Notes in Computer Science, V471], By Beng Chin Ooi.

2. Computational Geometry an Introduction [Springer-Verlag], By Franco P. Preparata and Michael Lan Shamos.