Notes on Project 1

version 1.44

August 28, 2017

1 Definitions

You program will keep a collection of rectangles which we refer to by C and a rectangle-quadtree which we refer to by T. The collection C will contain all the rectangles that have been created so far. T may only contain a subset of C as we may add/remove rectangles to T. Each rectangle has an associated name. The rectangles in C are sorted by their name. Rectangle names may only contain letters and digits. To sort the names you should use the standard collating sequence (i.e. the default ordering of characters/strings). Also, all coordinate values will be integer and in the range $[0, 2^{24}]$.

Definition 1 (Rectangle) *We will represent a rectangle by a pair of coordinates corresponding to its lower left corner and upper right corner. Although we only use integer coordinate values, you should think of these as real coordinate values. For example, a rectangle that spans from* $(2, 1)$ *to* $(4, 2)$ *includes the point* $(3.9, 1.99)$*. Note that a rectangle is closed on its left and bottom side and open on its top and right side. Figure 1 illustrates a rectangle from* $(2, 1)$ *to* $(4, 2)$*. Note that the rectangle* contains *all the points in the gray area and on the black border. Formally, a rectangle that spans from* (x_1, y_1) *to* (x_2, y_2) *includes a point* (x, y) *if and only if* $x_1 \leq x < x_2$ *and* $y_1 \leq y < y_2$*. In particular, this means a rectangle does not contain any of its corners except for its bottom left corner. You will see that this definition of rectangles will simplify your task when you divide a rectangular region to smaller rectangular regions as each point will then belong to exactly one of the sub-regions. Since we are assuming integer coordinate values, the smallest rectangle in this project will be of size* 1×1*.*

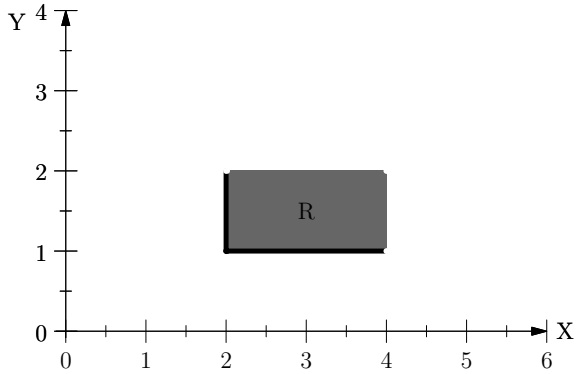

Figure 1: A rectangle from $(2, 1)$ to $(4, 2)$.

Definition 2 (Intersection) *We say two objects (e.g. rectangles) intersect if their overlapping region has a non-zero area. For example in Figure 2, the rectangle* $R1 = ((2, 1), (4, 2))$ *and* $R2 = ((0, 0), (2, 4))$ *do not intersect but* $R3 = ((1, 0), (3, 1))$ *intersects with* $R1$*. Also* $R3$ *and* $R1$ *do not intersect.*

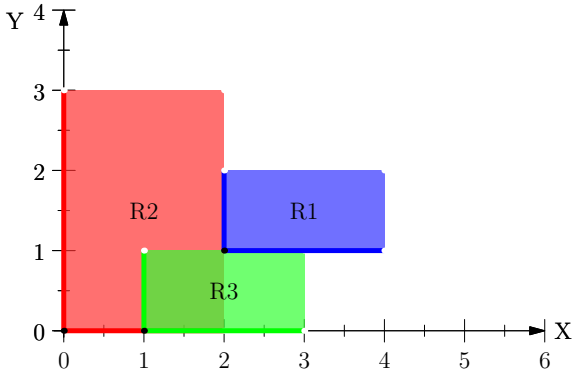

Figure 2: $R2$ and $R3$ intersect, $R1$ touches both $R2$ and $R3$.

Definition 3 (Touch) *We say two objects (e.g. rectangles) touch if their distance is* 0 *but they do not intersect. The distance between two objects is the length of the shortest line connecting them. For example in Figure 2, the rectangle* $R1$ *touches both* $R2$ *and* $R3$*.*

Definition 4 (Horizontal Distance) *The horizontal distance between two objects is defined as the distance between the projection of the two objects on the X-axis. The projection of a rectangle* $R = ((x_1, y_1), (x_2, y_2))$ *on the X-axis is the half-open interval* $[x_1, x_2)$*. So, the horizontal distance between two rectangles is just the distance between their projections on the X-axis. So, given the rectangle* $R' = ((x'_1, y'_1), (x'_2, y'_2))$*, we can compute the horizontal distance of* R *and* R' *by* $\max(x_1 - x'_2, x'_1 - x_2)$*. Notice that if the projections of* R *and* R' *overlap then this distance could be negative.*

Definition 5 (Vertical Distance) *The vertical distance between two objects is defined as the distance between the projections of the two objects on the Y-axis. This is defined similarly to the horizontal distance (See the Def. 4).*

Definition 6 (Quadtree Traversal) *When visiting the children of a quadtree node, you should visit them in the order* NW*,* NE*,* SW*,* SE*. This is important to get the correct ordering of output of some of the operations. Also, when processing a query, you should avoid visiting child nodes that do not intersect/touch/contain the queried rectangle/point.*

2 Input/Output

You program should read from the standard input. We will redirect the standard input to read from a file. The input contains one operation per each line. For each operation in the input, you should first print the operation name along with its argument(s) to the output. You should then process it. For each operation, you should output one line as follows unless stated otherwise. See the sample input/output in the appendix to get the idea of this. Also, note that we will ignore the white space when comparing you output against the correct output so don't worry about spaces.

- If the operation does not produce any result/message then just print the operation and its arguments like:

  `OPERATION(ARG1, ..., ARGN)`

- If the operation produces the output message `MESSAGE` then print the operation name and its argument and its output message like:

  `OPERATION(ARG1, ..., ARGN) : MESSAGE`

You can find an example input/output in the appendix.

3 Operations

The following is a description of the operations:

- `INIT_QUADTREE(int w)`: (OPCODE = 1)
  Initialize a new empty quad-tree T that covers the area from $(0, 0)$ to $(2^w, 2^w)$. Any existing quad-tree should be deleted. Print the message "`initialized a quadtree of width D`" where D is 2^w.

- `DISPLAY()` : (OPCODE = 2)
  This operation should be implemented as specified in the original project description. There will be no automated test for this operation.

- `LIST_RECTANGLES()` : (OPCODE = 3)
  Print the message "`listing N rectangles: `" where N is the total number of rectangles in C. Then, starting from the next line, print a list of all the rectangles in the collection C in the ascending order of their names. Print each rectangle in one line. For each rectangle print its name, its lower left x and y coordinate values and its upper right x and y coordinate values in order and separate them by spaces.

- `CREATE_RECTANGLE(string R, int x1, int y1, int x2, int y2)` : (OPCODE = 4)
  Create a new rectangle with the name `R` and lower left corner at (x_1, y_1) and upper right corner at (x_2, y_2) and add it to the collection C. Note that the rectangle does not need to be within the valid range of the rectangle tree T. Print the message "`created rectangle R`" where R is the name of the rectangle.

- `CREATE_RECTANGLE_RANDOM(string R)` : (OPCODE = 4)
  Create a new rectangle with the name `R` and random coordinate values and add it to C. Print the message "`created rectangle R X1 Y1 X2 Y2`" where R is the name of the rectangle and $(X1, Y1)$ and $(X2, Y2)$ are the coordinate values of its lower left corner and upper right corner, respectively.

- `RECTANGLE_SEARCH(string R)` : (OPCODE = 5)
  Search for all the rectangles in T that intersect with the rectangle `R`. Print the message "`found N rectangles: R1 R2 ... `" where N is the number of rectangles in T that intersect R. and $R1$, $R2$, ... are the names of those rectangles.

  If R itself is in T then it should be printed as well. You should print the intersecting rectangles in the order they were visited first. You should traverse the quadtree int the order of NW, NE, SW, SE to visit/print the rectangles in the correct order. You should avoid visiting quadtree nodes that do not intersect with R.

- `INSERT(string R)` : (OPCODE = 6)
  Add the rectangle R to the rectangle tree T. Then print the message "`inserted rectangle R`". If R intersects another rectangle in T or if it is already in T then print the message "`failed: intersects with S`" where S is the name of the rectangle already in T that would intersect with R. If the rectangle R is not entirely contained in the region covered by T then print the message "`failed: out of range`"

- `SEARCH_POINT(int x, int y)` : (OPCODE = 7)
  Find the rectangle in T that contains (x, y) and print the message "`found rectangle R`" where R is the name of the rectangle. If no such rectangle was found then print the message "`no rectangle found`". See Def. 1 for an explanation of when a point belongs to a rectangle.

- `DELETE_RECTANGLE(string R)` : (OPCODE = 8)
  Delete the rectangle R from the quadtree T. If successful, print the message "`deleted rectangle R`" where R is the name of the rectangle being deleted. Note that this operation deletes the rectangle from T not from C. If the rectangle is in C but not in the quadtree T then print the message "`rectangle not found R`".

- `DELETE_POINT(int x, int y)` : (OPCODE = 8)
  Find the rectangle in the quadtree T containing the point (x, y) and then delete it from the quadtree. If successful, print the message "`deleted rectangle R`" where R is the name of the rectangle being deleted. If no such rectangle was found then print the message "`no rectangle found at (x, y)`" where (x, y) is the coordinate values of the point. See Def. 1 for an explanation of when a point belongs to a rectangle.

- `TRACE ON` and `TRACE OFF`:
  These commands enable/disable trace output. When tracing is on, you should print the node number of the quadtree nodes that were visited in the order that they were visited (pre-order) during the operation. The root of the quadtree has node number 0. Children of a quadtree node with number N are numbered as $4N + 1$ to $4N + 4$ in the order NW, NE, SW, SE.

  You should traverse the children of a quadtree node in the order of NW, NE, SW, SE to print the node numbers is the correct order. When trace is on, the node numbers should be printed in the same line between a pair of brackets after the operation name/arguments like:

  ```
  OPERATION(ARG1, ..., ARGN) [NODE1 NODE2 .... ]: MESSAGE
  ```

  Where $NODE1$, $NODE2$, ... are the node numbers for the quadtree nodes that were visited.

  Note that only the following operations should print a trace output: `RECTANGLE_SEARCH`, `SEARCH_POINT`, `TOUCH`, `WITHIN`, `HORIZ_NEIGHBOR`, `VERT_NEIGHBOR`, `NEAREST_RECTANGLE`, `WINDOW` and `NEAREST_NEIGHBOR`. All the other operations are not affected.

- `TOUCH(string R)` : (OPCODE = 9)
  Search for all the rectangles in T that touch the rectangle `R` (but don't intersect it). Print the message "`found N rectangles: R1 R2 ...`" where N is the number of rectangles in T that touch R. and $R1$, $R2$, ... are the names of those rectangles.

  You should print the touching rectangles in the order they were visited first. You should traverse the quadtree int the order of NW, NE, SW, SE to visit/print the rectangles in the correct order.

- `WITHIN(string R, int d)` : (OPCODE = 10)
  Assume that $R = ((x_1, y_1), (x_2, y_2))$. Define the expansion of rectangle R by distance d to be the rectangle $R_{+d} = ((x_1 - d, y_1 - d), (x_2 + d, y_2 + d))$. Search for all the rectangles in T that intersect the donut shaped region contained between R and R_{+d}. A rectangle intersects the region contained between R and R_{+d} if and only if it intersects R_{+d} and it is not *contained* in R. Print the message "`found N rectangles: R1 R2 ...`" where N is the number of rectangles in T that intersect the region in $R_{+d} - R$ and $R1$, $R2$, ... are the names of those rectangles.

  You should print the intersecting rectangles in the order they were visited first. You should traverse the quadtree int the order of NW, NE, SW, SE to visit/print the rectangles in the correct order. Remember that you should avoid visiting quadtree nodes whose region do not intersect the query, for example if a quadtree node is entirely contained in R.

- `HORIZ_NEIGHBOR(string R)` : (OPCODE = 11)
  Find the rectangle in the quadtree T with the minimum non-negative horizontal distance to R. In other words, find the rectangle in T whose projection on the X-axis is closest to the projection of R on the X-axis but does not intersect it. For a definition of horizontal distance see Def. 4. If there are multiple rectangles that have the minimum non-negative horizonal distance to R then choose the one that was visited first during the traversal of the quadtree.

  Note that to implement this operation correctly you will need to use a priority queue. The priority queue will contain the nodes to be visited in the increasing order of their horizontal distance from the query rectangle. That is, nodes that have smaller horizontal distance to the query rectangle should be visited first. All the nodes with negative horizontal distance should be treated as having a horizontal distance of 0. If there are multiple nodes with the same horizontal distance then the one with a smaller node number should be visited first. You should avoid visiting quadtree nodes that are farther from the query rectangle than the nearest rectangle found so far.

You should print the message "`found rectangle S`" where S is the result of the query. If no such rectangle was found (i.e. if the quadtree is either empty or if all the rectangle in the quadtree have negative horizontal distance to the query rectangle) then print the message "`no rectangle found`".

- `VERT_NEIGHBOR(string R)` : (OPCODE = 11)
  Find the rectangle in the quadtree T with the minimum non-negative vertical distance to R. For a definition of vertical distance see Def. 5. If there are multiple rectangles that have the minimum non-negative vertical distance to R then choose the one that was visited first during the traversal of the quadtree. Traversal order and output message is similar to `HORIZ_NEIGHBOR` except that instead of horizontal distance, vertical distance should be used.

- `NEAREST_RECTANGLE(int x, int y)` : (OPCODE = 12)
  Find the rectangle in the quadtree T with the minimum distance to the point (x, y). The distance of a rectangle to the point (x, y) is the length of the shortest line segment connecting the two. In particular, if the point lies on the boundary or inside a rectangle then its distance to the rectangle by the previous definition is 0. If there are multiple rectangles that have the minimum distance to (x, y) then choose the one that was visited first during the traversal of the quadtree. Traversal order and output message is similar to `HORIZ_NEIGHBOR` except that instead of horizontal distance, euclidian distance to (x, y) should be used. Also note that this operation should always find some rectangle unless the quadtree is empty!

- `WINDOW(int x1, int y1, int x2, int y2)` : (OPCODE = 13)
  Search for all the rectangles in T that are entirely contained in the rectangle with lower left corner at (x_1, y_1) and upper right corner at (x_2, y_2). Print the message "`found N rectangles: R1 R2 ...` " where N is the number of rectangles in T that are entirely within the query rectangle and $R1$, $R2$, ... are the names of those rectangles.

  You should print the contained rectangles in the order they were visited first. You should traverse the quadtree in the order of NW, NE, SW, SE to visit/print the rectangles in the correct order.

- `NEAREST_NEIGHBOR(string R)` : (OPCODE = 14)
  Find the rectangle in the quadtree T that is closest to R but does not intersect R. The distance between two rectangles is the length of the shortest line segment connecting them. If there are multiple rectangles that have the minimum distance to R then choose the one that was visited first during the traversal of the quadtree. Traversal order and output message is similar to `HORIZ_NEIGHBOR` except that instead of horizontal distance, the euclidian distance of the quadtree nodes to R should be used. If a quadtree node intersects with R then its distance to R is considered to be 0. Also note that you should not visit quadtree nodes that are entirely contained in R because they cannot possibly contain a rectangle that does not intersect with R.

- `LEXICALLY_GREATER_NEAREST_NEIGHBOR(string R)` : (OPCODE = 15)
  This is similar to `NEAREST_NEIGHBOR` except that you should only consider the rectangles in T whose names are lexicographically greater than R. The traversal order and output message is also similar. Note that if there are multiple rectangles with the same minimum distance to R, you should choose the one that was visited first during the traversal of the quadtree.

- `LABEL()` : (OPCODE = 16)
  You should do a connected component labeling of the rectangles in the quadtree. Two rectangles are considered to be connected if they are touching (either a side or a corner). You should then print the message "`found N connected components:` " where N is the total number of connected components. Then, starting from the next line, print a list of all the rectangles in the quadtree in the ascending order of their names. For each rectangle print one line containing its name followed by the name of the rectangle in its connected component that has the lexicographically smallest name in that connected component.

4 Hints

In the following, we represent every point by its x and y coordinate values and every rectangle by a pair of points. For example, $R = (p_1, p_2)$, is a rectangle with lower left corner at $p_1 = (x_1, y_1)$ and upper right corner at $p_2 = (x_2, y_2)$.

Because we represent points by two dimensional vectors, we can define algebraic operations on points. For example given two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, we can define $p_1 + p_2$ to be the point $(x_1 + x_2, y_1 + y_2)$. Similarly, we can define scalar multiplication (e.g. $5p_1 = (5x_1, 5y_1)$), subtraction, etc. Given a vector $v = (x, y)$, we denote the length of the vector v by $||v||$ which is given by $||v|| = \sqrt{(x^2 + y^2)}$. Notice that for a point $p = (x, y)$, $||p||$ is the length of the vector from $(0, 0)$ to p.

- **Comparing Points:**
  Let $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$. We say $p_1 < p_2$ if and only if both $x_1 < x_2$ and $y_1 < y_2$. Similarly, we say $p_1 \leq p_2$ if and only if both $x_1 \leq x_2$ and $y_1 \leq y_2$. Note that based on this definition it is possible that neither $p_1 \leq p_2$ nor $p_2 \leq p_1$ (e.g. consider the two points $p_1 = (0, 1)$ and $p_2 = (1, 0)$).

- **Inside:**
  Given a point p and a rectangle $R = (p_1, p_2)$, the point p is *inside* R if and only if $p_1 \leq p < p_2$.

- **Min/Max:**
  Given two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, we define the min and max operations as follows:

$$\min(p_1, p_2) = (\min(x_1, x_2), \min(y_1, y_2))$$
$$\max(p_1, p_2) = (\max(x_1, x_2), \max(y_1, y_2))$$

  Similarly, we can define the min and max for more that two points (e.g. $\max(p_1, p_2, \cdots, p_n)$). In other words, the minimum/mamximum of several points is a point that has the minimum/maximum of their coordinate values.

- **Valid Rectangle:**
  Given a rectangle $R = (p_1, p_2)$, we say a rectangle is *valid* if and only if $p_1 \leq p_2$, otherwise the rectangle R is *invalid*.

- **Empty Rectangle:**
  A rectangle $R = (p_1, p_2)$ is *empty* if and only if R is a valid rectangle and $p_1 < p_2$ is false. In other words, R is a *valid* but *empty* rectangle if and only if $p_1 \leq p_2$ is true but $p_1 < p_2$ is false.

- **Intersection:**
  Given two rectangles $R = (p_1, p_2)$ and $R' = (p'_1, p'_2)$, let $p''_1 = \max(p_1, p'_1)$ and $p''_2 = \min(p_2, p'_2)$. Then R and R' *intersect* if and only if the rectangle $R'' = (p''_1, p''_2)$ is a *valid* and not *empty* rectangle. In other words, they intersect if and only if $p''_1 < p''_2$. If they do intersect then their intersection is given by the rectangle R'' defined above.

- **Touch:**
  Given two rectangles $R = (p_1, p_2)$ and $R' = (p'_1, p'_2)$, we say that R and R' *touch* if their intersection is a valid but empty rectangle. In other words: Let $p''_1 = \max(p_1, p'_1)$ and $p''_2 = \min(p_2, p'_2)$. Then R and R' touch if and only if $p''_1 \leq p''_2$ is true but $p''_1 < p''_2$ is false.

- **Rectangle Containment:**
  Given two rectangles $R = (p_1, p_2)$ and $R' = (p'_1, p'_2)$, we say R' is contained in R if and only if $p_1 \leq p'_1$ and $p'_2 \leq p_2$. This is also equivalent to saying R' is contained in R if and only if the intersection of R' and R is R' itself.

- **Point-Point Distance:**
  Given two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ let $d = p_1 - p_2$ be their difference vector (i.e. the vector connecting p_1 to p_2). The distance between p_1 and p_2 is given by $||d||$. In other words the distance between p_1 and p_2 is $||p_1 - p_2||$ which is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

- **Point-Rectangle Distance:**
  Given a point p and a rectangle $R = (p_1, p_2)$, let $d = \max(p_1 - p, p - p_2, (0,0))$ be their difference vector. Then, the distance between p and R is given by $||d||$.

- **Rectangle-Rectangle Distance:**
  Given two rectangles $R = (p_1, p_2)$ and $R' = (p'_1, p'_2)$, let $d = \max(p_1 - p'_2, p'_1 - p_2, (0,0))$ be their difference vector. Then, the distance between R and R' is given by $||d||$.

- **Rectangle-Rectangle Vertical/Horizontal Distance:**
  See Def. 4 and Def. 5.

You will need to use a priority queue to implement some of the operation (i.e. `HORIZ_NEIGHBOR`, `VERT_NEIGHBOR`, `NEAREST_RECTANGLE`, `NEAREST_NEIGHBOR`, and `LEXICALLY_GREATER_NEAREST_NEIGHBOR`). The following is a pseudo-code which should give you an idea of how you should implement these operations. Note that the following may miss the details specific to each operation so you may need to modify it or add to it to implement each operations correctly.

1em boxed1em

```
Find_Nearest (Object query);
Let Q be a priority queue of quad-tree nodes ;
min_dist ← ∞ ;
closest_rect ← null ;
Push the root of the quad-tree into Q ;
while Q is not empty do
    p ← pop the next quad-tree node from the head of Q ;
    d ← distance of p from the query;
    if d < min_dist then            // You may need to check for other conditions as well
        if tracing is enabled then
            print the quad-tree node number of p ;
        if p is a gray node then
            Push all of the child nodes of p into Q ;
        else if p is a black node then
            r ← the rectangle in p ;
            d' ← distance of r from the query;
            else if d' < min_dist then      // You need to check for other conditions as well
                min_dist ← d' ;
                closest_rect ← p ;
end
return closest_rect ;
```

Notice that Q is a priority queue of quad-tree nodes in which nodes with shorter distance to the *query* come first. If two nodes have the same distance to the *query* then the one with a lower quad-tree number comes first. Also, notice that *distance* is defined based on the operation you are implementing. For each specific operation you may need to check for other conditions. For example, if you are implementing the `VERT_NEIGHBOR` then at line 1 you should also check that the image of p on the vertical axis is not entirely *contained* in the image of the *query* on the vertical axis (because if it is so then p cannot possible contain the solution). Similarly, at line 1 you should check that the image of r on the vertical axis does not overlap the image of the *query* on the vertical axis (remember that this is for `VERT_NEIGHBOR`, for other operations you need to check for other conditions).

A Sample Input/Output

Here is a sample input:

```
INIT_QUADTREE(8)
LIST_RECTANGLES()
CREATE_RECTANGLE(R1,5,5,25,25)
CREATE_RECTANGLE(R2,20,20,31,31)
CREATE_RECTANGLE(R3,30,30,40,40)
CREATE_RECTANGLE(R4,200,200,210,210)
TRACE ON
INSERT(R4)
SEARCH_POINT(1,1)
INSERT(R1)
INSERT(R4)
INSERT(R2)
SEARCH_POINT(1,1)
INSERT(R3)
SEARCH_POINT(5,5)
TRACE OFF
LIST_RECTANGLES()
RECTANGLE_SEARCH(R2)
SEARCH_POINT(1,1)
INSERT(R2)
SEARCH_POINT(7,7)
INSERT(R2)
DELETE_POINT(10,10)
INSERT(R2)
INIT_QUADTREE(8)
INSERT(R4)
INSERT(R4)
CREATE_RECTANGLE(S1,5,5,10,10)
CREATE_RECTANGLE(S2,10,10,13,13)
CREATE_RECTANGLE(S3,14,14,20,20)
CREATE_RECTANGLE(S4,13,2,14,24)
INSERT(S1)
INSERT(S3)
TOUCH(S2)
TRACE ON
HORIZ_NEIGHBOR(R2)
TRACE OFF
VERT_NEIGHBOR(R4)
INSERT(R3)
VERT_NEIGHBOR(R4)
INSERT(R4)
INSERT(R4)
INSERT(R4)
NEAREST_RECTANGLE(27,26)
NEAREST_RECTANGLE(50,50)
NEAREST_RECTANGLE(130,130)
NEAREST_RECTANGLE(160,160)
INSERT(S2)
LABEL()
INSERT(S4)
LABEL()
```

Here is the corresponding output:

```
INIT_QUADTREE(8): initialized a quadtree of width 256
LIST_RECTANGLES(): listing 0 rectangles:
CREATE_RECTANGLE(R1,5,5,25,25): created rectangle R1
CREATE_RECTANGLE(R2,20,20,31,31): created rectangle R2
CREATE_RECTANGLE(R3,30,30,40,40): created rectangle R3
CREATE_RECTANGLE(R4,200,200,210,210): created rectangle R4
TRACE ON
INSERT(R4): inserted rectangle R4
SEARCH_POINT(1,1)[ 0]: no rectangle found
INSERT(R1): inserted rectangle R1
INSERT(R4): failed: intersects with R4
INSERT(R2): failed: intersects with R1
SEARCH_POINT(1,1)[ 0 3]: no rectangle found
INSERT(R3): inserted rectangle R3
SEARCH_POINT(5,5)[ 0 3 15 63 255]: found rectangle R1
TRACE OFF
LIST_RECTANGLES(): listing 4 rectangles:
R1 5 5 25 25
R2 20 20 31 31
R3 30 30 40 40
R4 200 200 210 210
RECTANGLE_SEARCH(R2): found 2 rectangles: R1 R3
SEARCH_POINT(1,1): no rectangle found
INSERT(R2): failed: intersects with R1
SEARCH_POINT(7,7): found rectangle R1
INSERT(R2): failed: intersects with R1
DELETE_POINT(10,10): deleted rectangle R1
INSERT(R2): failed: intersects with R3
INIT_QUADTREE(8): initialized a quadtree of width 256
INSERT(R4): inserted rectangle R4
INSERT(R4): failed: intersects with R4
CREATE_RECTANGLE(S1,5,5,10,10): created rectangle S1
CREATE_RECTANGLE(S2,10,10,13,13): created rectangle S2
CREATE_RECTANGLE(S3,14,14,20,20): created rectangle S3
CREATE_RECTANGLE(S4,13,2,14,24): created rectangle S4
INSERT(S1): inserted rectangle S1
INSERT(S3): inserted rectangle S3
TOUCH(S2): found 1 rectangles: S1
TRACE ON
HORIZ_NEIGHBOR(R2)[ 0 1 3 13 15 61 63 254]: found rectangle S3
TRACE OFF
VERT_NEIGHBOR(R4): found rectangle S3
INSERT(R3): inserted rectangle R3
VERT_NEIGHBOR(R4): found rectangle R3
INSERT(R4): failed: intersects with R4
INSERT(R4): failed: intersects with R4
INSERT(R4): failed: intersects with R4
NEAREST_RECTANGLE(27,26): found rectangle R3
NEAREST_RECTANGLE(50,50): found rectangle R3
NEAREST_RECTANGLE(130,130): found rectangle R4
NEAREST_RECTANGLE(160,160): found rectangle R4
INSERT(S2): inserted rectangle S2
```

```
LABEL(): found 4 connected components:
R3 R3
R4 R4
S1 S1
S2 S1
S3 S3
INSERT(S4): inserted rectangle S4
LABEL(): found 3 connected components:
R3 R3
R4 R4
S1 S1
S2 S1
S3 S1
S4 S1
```