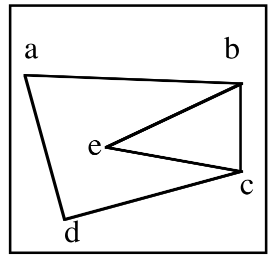
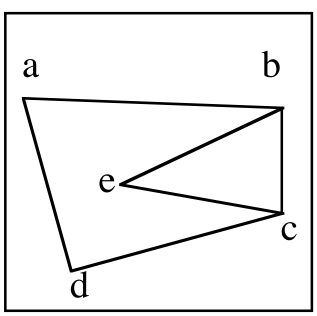
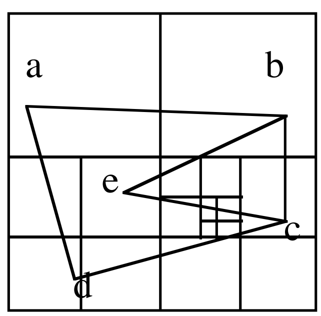
Quadtree representations for polygonal maps - PM quadtrees

- Adaptation of PR quadtrees for representations of collections of line segments which meet or end at vertices.
  - when line segments are inserted into a PM quadtree, nodes are "split" if they do not satisfy certain conditions
  - for PR quadtrees, a block is split if it contains more than one point.
  - the conditions here will be more complicated



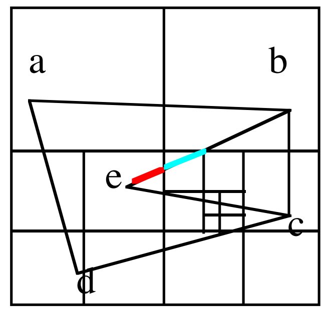
PM_1 quadtree





Definition

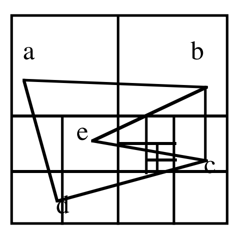
- q-edges
  - when line segments are inserted into a PM quadtree they will intersect several leaf nodes
  - a q-edge is the intersection of a line segment with a quadtree leaf node
  - when a line segment p_1p_2 is inserted into a PM quadtree, we must compute whether the line segment intersects the square regions of the quadtree nodes - clipping
  - for some algorithms, we might also want to know the end points of the intersection of the line segment with the quadtree node.



Definition

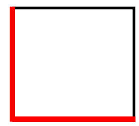
- Four conditions for a leaf node to be a valid PM_1 quadtree leaf node

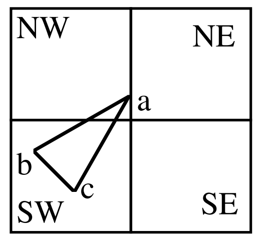
  1) At most one vertex can lie in a leaf node - in which case the leaf node will contain all the q-edges incident on that vertex - this means that the two end points of a line segment must be in different leaf nodes
  2) If a leaf node contains a vertex, it contains no q-edges from line segments NOT incident on that vertex
  3) If a leaf node does not contain a vertex, it contains at most one q-edge
  4) Leaf nodes are maximal - i.e., if a grey node has four sons that are leaves, those leaves could not be merged into a valid leaf.



Vertices on borders of quadtree nodes

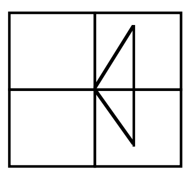
- For a PR quadtree, we adopted the convention that a node is closed on its west and south borders, but open on its north and east borders.
- However, this leads to a problem if a vertex lies on a closed edge of a quadtree block, but its edges emanate into "open" quadrants
  - a belongs to NE
  - q-edges of ab and ac belong to NW, but a does not
  - could never separate q-edges of ab and ac in NW by splitting

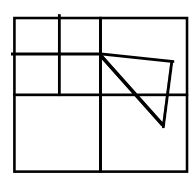




Vertices on borders of quadtree nodes

- Solution: if a vertex lies on the border of a quadtree node, insert it into all of the quadtree nodes that meet at that border and contain that point.
  - vertex can belong to at most four quadtree nodes, if it coincides with a corner of a node
  - vertex can belong to three quadtree nodes if it lies on a midpoint of a quadtree edge, and its neighbor is split.



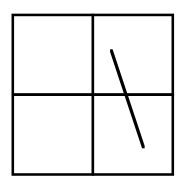


PM quadtree structures

- q-edges are the line segments clipped to the quadtree nodes during insertion
  - p_1 - end point 1
  - p_2 - end point 2
  - name - allows you to retrieve end points of the original line segment to which q-edge belongs
- Nodes of a PM quadtree contain
  - four son pointers
  - nodetype (grey or black)
  - size and location of block
  - dictionary of the q-edges that lie in the block - linked list

Insertion into a PM quadtree

- Cases not handled:
  - isolated points
  - crossing lines (in which case intersections have to be detected and new vertices constructed).
- Basic traversal algorithm
  - traverse tree in preorder
  - clip the line segment to be inserted against the nodes traversed
    - » allows us to rule out parts of the quadtree that will not contain q-edges from this line segment

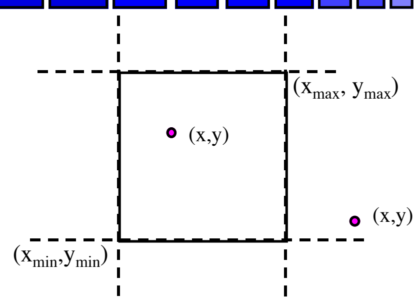


Insertion into a PM quadtree

- If a clipped edge, e, can be inserted into a leaf node, Q, without violating the PM quadtree conditions, then we do so and quit.
- Otherwise, create a list, L, containing e and the q-edges from Q, and recursively insert the edges in L into the four sons of Q

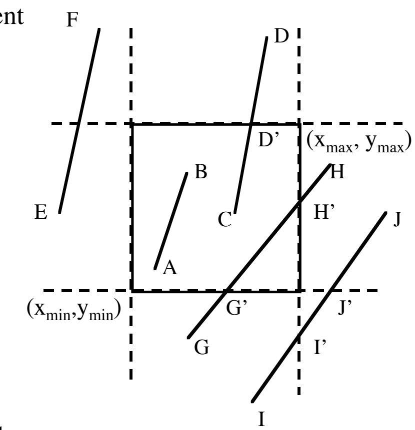
Clipping algorithms

- Point in region
  - $x_{min} \le x \le x_{max}$
  - $y_{min} \le y \le y_{max}$



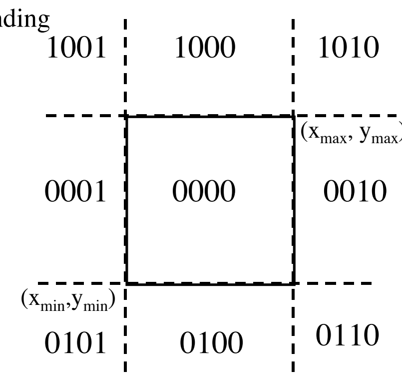
Clipping algorithm - lines

- Considers only end points of the line segment
- Trivial acceptance conditions
  - if both end points lie within the block (AB), then the line segment lies completely within the block.
- Trivial rejection conditions
  - both end points of EF have minimal x coordinate < x_{min}
  - can make similar checks for x_{max}, y_{min}, and y_{max}.
- If trivial rejection criteria are not met, line segment is divided into two segments at a clip edge so that one segment can be trivially rejected.



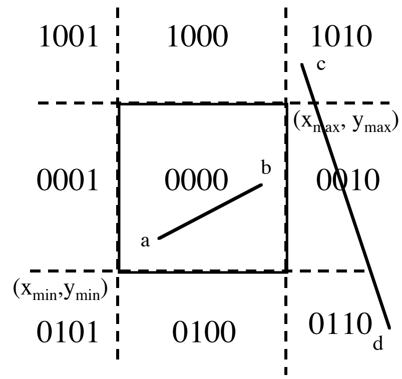
Clipping algorithm - lines

- Outcodes - 4 bit code assigned to the regions of space determined by extensions of edges of block
  - each bit is associated with a direction (left, right, above, below)
  - a bit is set if that region is in the corresponding direction from the block
    - » b_1 - above top edge ($y > y_{max}$)
    - » b_2 - below bottom edge
    - » b_3 - right of right edge
    - » b_4 - left of left edge
  - For point (x,y)
    - » $b_1 = \text{sign}(y-y_{max})$
    - » $b_2 = \text{sign}(y-y_{min})$



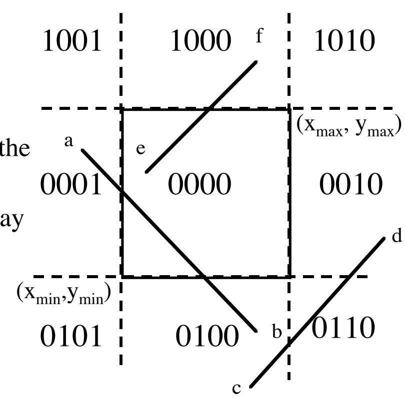
Trivial accept and reject conditions

- If the 4-bit outcodes of both end points are 0000, then the line segment lies inside the box
- Let c_1 be the 4 bit outcode for one end point and c_2 be the outcode for the other end point of a line segment
  - if c_1 AND $c_2 \neq 0$, then the line segment joining c_1 to c_2 can be trivially rejected



Nontrivial rejection

- Line segment will be subdivided into two segments
  - at least one of the two segments will be trivially rejected
- To compute subdivision, determine edge that the line segment crosses
  - cut the segment at this edge and throw away the piece in the outside halfplane
- Bits set in a nonzero outcode correspond to edges crossed
  - if one point lies in outside halfplane of an edge, and the segment is not trivially rejected, then the other end point must lie in the inside halfplane of the same edge, and the line segment must cross it.



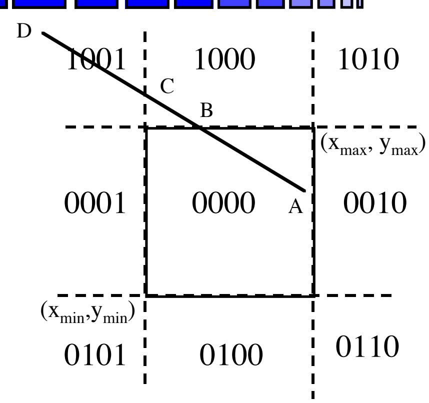
- line from a to b crosses left edge
- line from b to a crosses bottom edge
- line from f to e crosses top edge

Nontrivial rejection

- Compute the outcode of both endpoints
- Find an endpoint that lies outside the box (at least one will or we would have trivially accepted the line segment)
- Test the outcode to find the first edge that is crossed (left-to right scanning of outcode bits).
  - determine the corresponding intersection point
- Clip the line segment from the outside end point to the intersection point, compute the outcode of the intersection point and iterate the algorithm.

Example

- AD
  - A has outcode 0000
  - D has outcode 1001
  - D indicates line segment crosses top and left edge
  - top edge considered first, and AD is clipped to AB.
  - B-s outcode is 0000

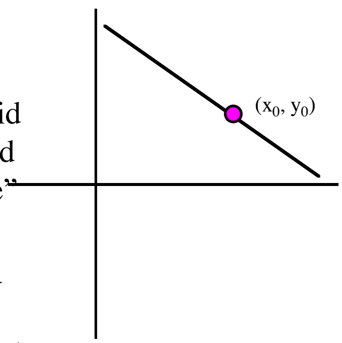


Algorithm details

- Representation of lines

  1) $y = y_0 + dy/dx\ (x-x_0)$

  2) $x = x_0 + dx/dy(y-y_0)$



- Using this representation will avoid the problems typically encountered with vertical lines having “infinite” slope.
  – these lines have $dx/dy = 0$, and the algorithm will only use equation 2 to clip a line segment to an edge

Algorithm

```
done := false;
compoutcode (x0,y0, code0);
compoutcode(x1, y1, code1);
repeat
    if code1 = 0 and code2 = 0 then
     done := true
    else if code1 AND code2 ≠ 0 then
           done := true
    else {clip}
    begin
           codeout := if code1 = 0 then code2
    else code1;{pick one end point outside
    block}
```

```
If TOP in codeout then {divide at top}
    begin
    x := x0 + (x1 - x0)(ymax-y0)/(y1-y0);
    y := ymax
    end
else if BOTTOM in codeout
    begin
    x := x0 + (x1 - x0)(ymin-y0)/(y1-y0);
    y := ymin
    end
```

Algorithm

else if RIGHT in codeout then

begin

$y := y_0 + (y_1-y_0)(x_{max}-x_0)/(x_1-x_0)$

$x := x_{max}$

end

else if LEFT in code out then

begin

$y := y_0 + (y_1-y_0)(x_{min}-x_0)/(x_1-x_0)$

$x := x_{min}$

end

{Next we clip the line to (x,y) and prepare for next pass}

If codeout = $code_0$ then

begin

$x_0 := x$; $y_0 := y$; compoutcode(x_0, y_0, $code_0$)

end

else

begin

$x_1 := x$; $y_1 := y$; compoutcode(x_1, y_1, $code_1$)

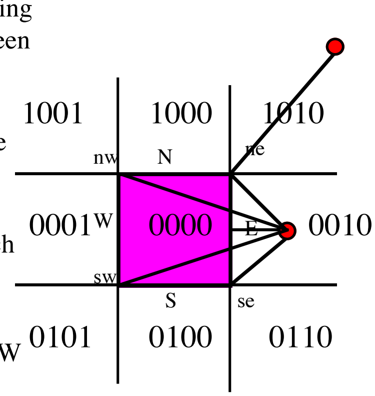
end

end {subdivide}

until done

Outcodes and nearest neighbors

- Nearest neighbor algorithms require finding lower bounds on shortest distances between a point/line and a quadtree node.
- For a point, this shortest distance can be either to a vertex or a side of the quadtree node.
- The outcode tells us whether the shortest distance is to a vertex or a side, and which vertex or side to consider.
  - 1001 - compute distance to nw
  - 0001 - compute distance to segment W
  - 0101 - compute distance to sw



Outcodes and nearest neighbors

if Clip(L,R) then LBdist(L,R) := 0

o_1 = outcode(p_1)

o_2 = outcode(p_2)

if $o_1 = o_2$ then case

/o_1/ = 1001: LBdist(L,R) := pt-to-line(nw,L)

/o_1/ = 1000: LBdist(L,R) := line-to-line(L,N)

•

•

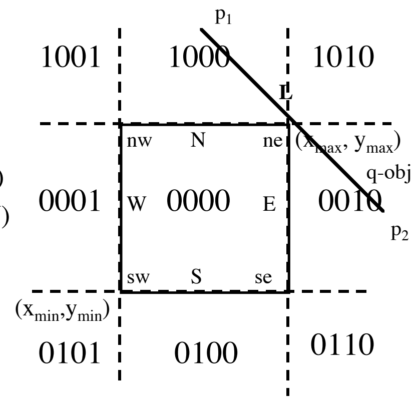
end case

if o_1 AND $o_2 \neq 0$ then case

TOP in o_1: LBdist(L,R) := line-to-line(L,N)

RIGHT in o_1: LBdist(L,R) := line-to-line(L,R)

••• else



Outcodes and nearest neighbors

case

/o_1 = 1001/

case

/o_2 = 0010/ LB.. := pt-to-line(ne,L);

/o_2 = 0110/ LB := min(pt-to-line(ne,L),
pt-to-line(sw,L));

/o_2 = 0100/ LB := pt-to-line(sw,L) end

/o_1 = 1000/

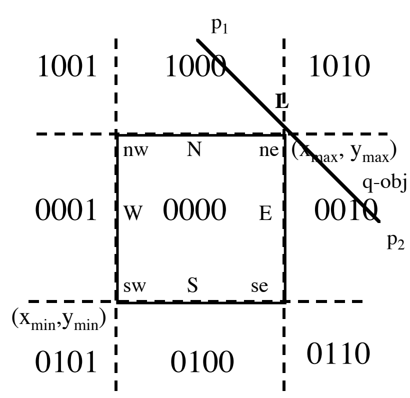
case

/RIGHT in o_2/ LB := pt-to-line(ne, L)

/LEFT in o_2/ LB := pt-to-line(nw,L)

end

•••



PM insertion algorithms

Procedure PM-insert(P,R)

{P is a list of line segments;

R is the root of the quadtree}

L := Clip(P,R) {Find the lines in P that intersect R; not necessary to determine clipped segmentsfor insertion};

if empty(L) then done;

if Leaf(R) then

    begin

    L := Union(L, line-list(R));

    if valid-PM(L,R) then

        begin

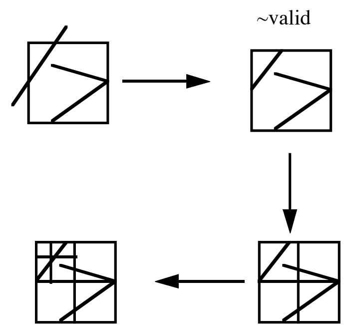
        line-list(R) := L ; return end

else split-node(R);

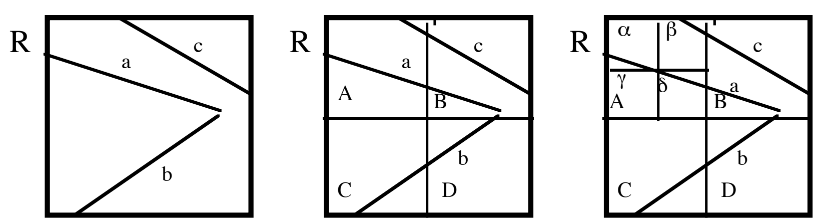
for dir in [NW, NE, SW, SE] do

    PM-insert(L, dir(R));

end



Insertion example



list(r) = {a,b}
L = UNION(c, {a,b})
= {a,b,c}
~valid({a,b,c}, R)
so, Insert({a,b,c}) into
new sons of R

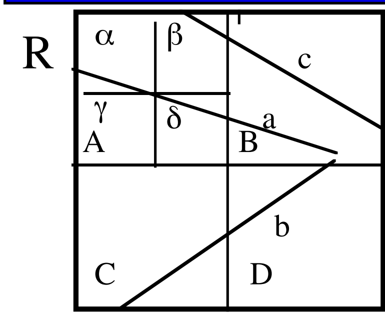
Insert({a,b,c}, A)
L = Clip({a,b,c}, A)
= (c,a)
L= UNION({a,,c},{a})
= {a,,c}
~valid({c,a}, L)

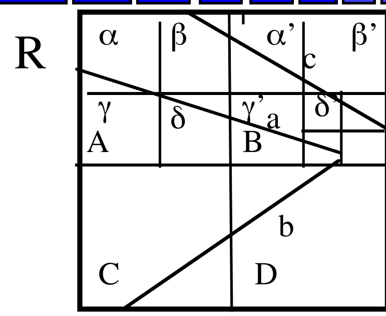
Insert({a,c}, α)
L = Clip {a,c}, α) = a
Valid, so List(α) = a

Insert({a,c}, β)
L = Clip({a,c}, β) = c
Valid, so List(β) = c

γ and δ also valid

Insertion example





Insert({a,b,c}, B)
L = Clip({a,b,c}, B)
 = {a,b,c}
Not valid, so split and
 insert into sons

α’, β’, and γ’ will be valid
But δ’ will not, so {a,b,c} will
be inserted into new sons of δ’.

Checking validity

Boolean valid-PM(L,S);

{L points to a list of records; each record on L contains pointers to a <u>line-segment</u> and the <u>next</u> element on L}

if empty(L) then return (true);

p_1 := end-point$_1$(line_segment(L))

p_2 := end-point$_2$(line_segment(L))

if one-element(L) then return
 (~[pt-in-square(p_1, S) AND
 pt-in-square(p_2, S)])

{if L contains only a single line segment, its end points must be in different quadtree nodes}

{Final case - make sure all line segments meet at either p_1 or p_2}

{Check first line and make sure exactly of its endpoints is in S}

if ~[pt-in-square(p_1, S) EOR pt-in-square(p_2, S)] then return (false)

{determine which end point lies in S - this must be the common vertex}

p := if pt-in-square(p_1, S) then p_1 else p_2

{Two conditions to check for each remaining line segment in L: only only end point can be in S and it must be coincident with p}

Checking validity

list-rec := (next(L));

while next(list-rec) $\neq$ NIL do

begin

list-rec := next(list-rec)

p'_1 := end-point$_1$(line_segment(list-rec))

p'_2 := end-point$_2$(line_segment(list-rec))

if p $\neq$ p'_1 AND p $\neq$ p'_2 then return (false) {p **must** match one end point of line segment}

if p = p'_1 and pt-in-square(p'_2, S) then return (false) {in this case both end points of the next line on the list are in S - not allowed}

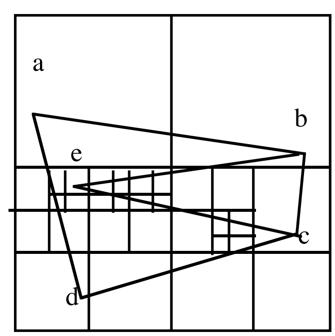
if p = p'_2 and pt-in-square(p'_1, S) then return (false) {symmetric case}

end while

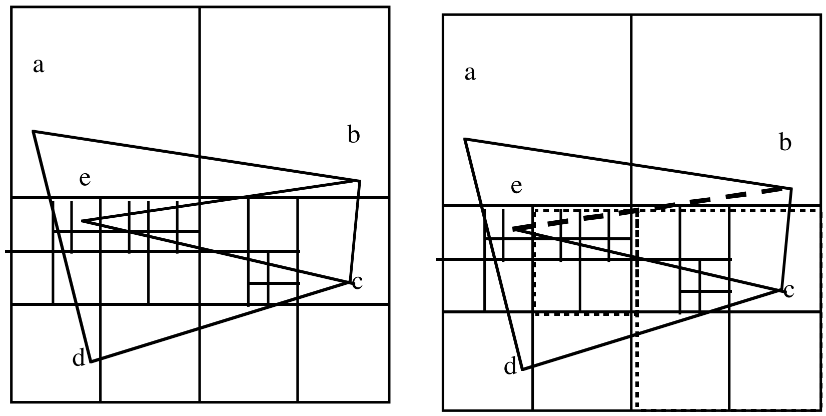
return (true)

Deletion from PM quadtrees

- Similar to deletion from a PR quadtree
  - delete line from all the leaf nodes through which it passes
  - check grey nodes to see if they can be merged
- Two configurations of edges lead to merging
  - deletion of a line segment causes all of the sons of a node to have 0 or 1 common edge passing through them
  - deletion of a line segment creates a nonleaf node in which all of the edges meet at a common vertex
  - consider deletion of “be”

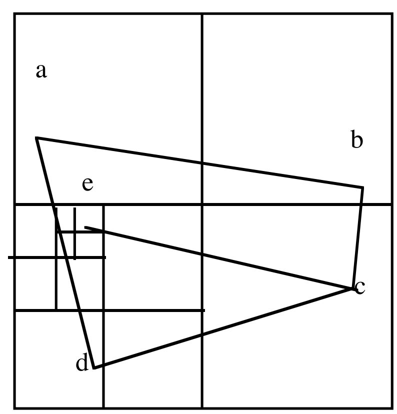


Deletion example



Quadtrees for polygonal maps

Deletion example



Quadtrees for polygonal maps

Deletion algorithm

procedure PM-delete(LS, R)

{delete the line segment, LS, from the PM quadtree rooted at R}

LS := clip(LS, R);

if LS = nil then return {LS does not fall in the block corresponding to R}

If Gray(R) then (R is a grey node in the PM quadtree}

    begin

    for direction := [NW,NE,SW,SE] do PM-delete(LS, R) {delete LS from each son}

Deletion algorithm

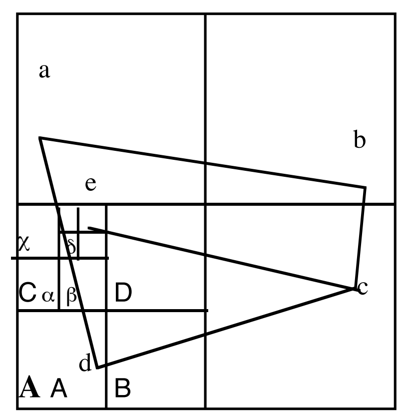
```
if possible_PM_merge(R)  {merge possible if one son is a leaf}then
        begin
        new-dictionary_R := NIL;
        if PM-merge (R, R, new-dictionary_R )  then
        begin
        dictionary(R) := new-dictionary_R
        nodetype(R) := leaf
        end
    end
end
else  {leaf node, just delete LS from its dictionary}
    dictionary(R) := dictionary(R) - LS
```

PM merge

- Once we have, recursively, completed the deletion of a line segment from the four sons of a quadtree node, R, we need to determine if those four sons should be "collapsed"
- Straightforward way to do this is to compute the union of the line segments in those four sons, and using PM check to determine if they are a valid collection of line segments for R. If so, we collapse.
- However, this is very inefficient
  - any one of the sons of R might contain a collection of line segments which would be an illegal configuration in R.
  - if so, the union will certainly not be a legal configuration

Example

- Check if **A** is a valid PM node after deletion of be
  - First check if the collection of line segments in the subtree rooted at C is a valid set for **A**
    - » But first check if the set of lines in the subtree rooted at δ is a valid set for **A**
    - » It isn't, so the merge at **A** will fail



PM-merge

Boolean procedure PM-merge (P, R, line-list)

{ Determine if the set of line segments in the PM quadtree rooted at P form a valid PM_1 quadtree node for the square region represented by R.}

```
if Leaf (P) then
    begin
    line-list := union(line-list, dictionary(P))
    return (true)
    end
for dir = (NW, SW, SE, SW) if ~PM1-merge(son(P, dir), R, line-list) then
    return false;
return valid-PM(line-list, R)
```