Universitat Politècnica De Catalunya

Barcelona Tech - UPC

Advanced Data Structures

Final Project

Deletion in Two-Dimensional Quad Trees

Author:
Leonardo Menti

leonardo.menti@estudiantat.upc.edu

June 2022

Contents

1 Introduction

A Quad Tree is an unbalanced tree data structure in which all internal nodes have exactly four child nodes. Quad trees are often used to partition a two-dimensional space by recursively dividing it into four quadrants, commonly referred to as Northeast, Northwest, Southwest, Southeast (respectively numbered 1, 2, 3, 4). Common uses of this type of structures are: representation of images, spatial indexing or sparse data storage, such as storing formatting information for a spreadsheet or matrix calculations. There are different types of quad trees, for example the region quad trees or the point quad trees. The region quad tree represents a partition of space in two dimensions by decomposing the region into four equal quadrants, subquadrants, and so on with each leaf node containing data corresponding to a specific subregion. Point quad trees share the features of all quad trees, but the center of a subdivision is always on a point. In this work I'm focusing about point quad trees.

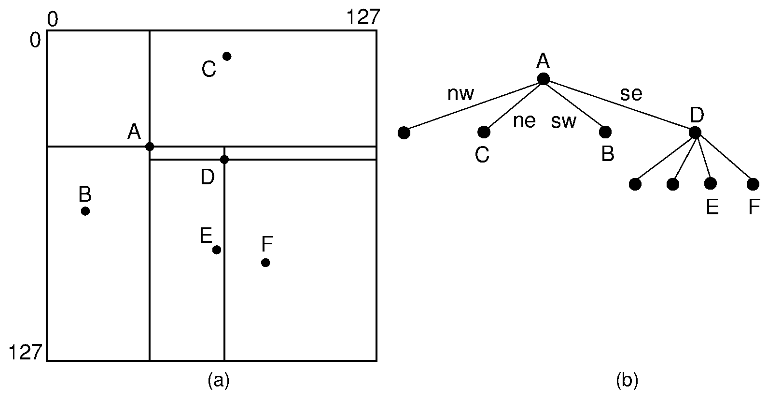

Figure 1: Quad Tree example

In this project I have implemented the operation of deletion in two-dimensional quad trees proposed by Hanan Sarnet from the University of Maryland [1]. This is an algorithm for deletion in two-dimensional quad trees that handles the problem in a manner analogous to deletion in binary search trees. The objective of this algorithm is to reduce the number of nodes that need to be reinserted. In section 2 and section 6 you can find my implementation of Quad Tree, in which I coded all the functions and the relative lines related to the *delete* operation.

In section 3 I tested the algorithm in order to confirm the expectations about the algorithm. I compared the number of re-insertions that are performed using the new algorithm and using the classical method for deletion. In section 4 and 5 there are my observations about the data structure, the implementation and about the experiments that I performed. The full code that implements the quad tree and the experiments code are at the appendix, in section 6.

2 Implementation

I chose to implement the Quad Tree structure in Java and the starting point of my implementation was the code provided by the Algorithms book [2]. This code has a basic implementation of the `QuadTree` class and the related `Node` inner class. I modified and added parameters and functions in order to implement the operation of deletion and to perform the experiments.

I implemented the Quad Tree as a class with the following parameters:

- `public Node root`: represents the root of the tree
- `private final int Lx`: x-dimension represented by the tree
- `private final int Ly`: y-dimension represented by the tree
- `public int n_re_insertion`: used in the experiments section in order to count the number of re-insertions

The class `QuadTree` has also two static dictionaries that helps in retrieving the number of a quadrant starting from its name and vice versa (ex. "NE" : 1):

- `private static final HashMap<String, Integer> quad2dir`
- `private static final HashMap<Integer, String> dir2quad`

I implemented also the inner class `Node`, that represent a node of the quad tree. The class has the following parameters:

- `int x, y`: represents the position of the node
- `Node NW, NE, SE, SW`: quadrant sons of the node
- `int value`: represents the value of the node
- `int direction`: number that says which is the direction, in terms of quadrant, of the node with respect to the parent
- `Node parent`: permits the access to the parent of the node

The very first function that I implemented was `dfs`, in order to have a good output representation of the tree. Then I implemented the following function, that define the concept of a direction opposite to a given direction, a notion used many times in the algorithm.

```java
private int conjugate(int n) {
    return ((n+1)%4)+1;
}
```

According to the algorithm [1], the first step to implement the operation of deletion is to find the proper candidate that will replace the node that has to be deleted.

Find Candidate The algorithm relies on the same criterion used in binary search tree to determine the "closest" node. I implemented the function that, for each quadrant of the tree rooted at the node to be deleted, follows the branch corresponding to `conjugate(i)` until a node without a sub-tree is encountered. That node is the candidate for the quadrant. The function produce four candidates, one for each quadrant of the tree rooted at the node to be deleted, and at the end call the function `chooseCandidate`.

Choose Candidate This function chooses the "closest" candidate between the four ones found with the previous function. The first attempt is made by trying to choose the candidate that is closer to each of its bordering axes than any other candidate which is on the same side of these axes. This is the so called **Property 1** and in order to implement it, I produced the following auxiliary function:

```
private boolean isCloserToAxis(int x, int y, Node candidate, Node first,
    Node second){
   return (Math.abs(x-candidate.x) < Math.abs(x-first.x) &
       Math.abs(y-candidate.y) < Math.abs(y-second.y));
}
```

The function checks if the *candidate* node of the arguments is closer to the x-axis then the *first* node candidate, and the same with the *second* one with the y-axis. There is the possibility that none of the candidates satisfy the criterion or that several candidates satisfy the condition, as we can see from the Figure 2.

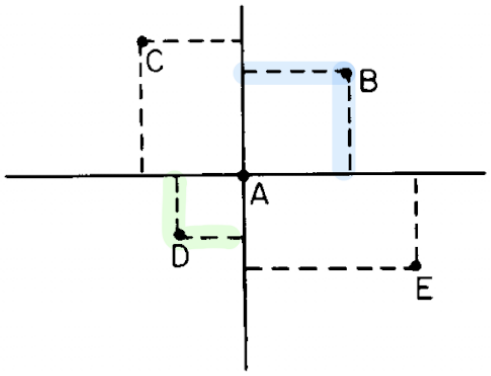

Figure 2: Quad Tree with two nodes being closest to their bordering axes

If only one candidate is found, the function returns that candidate, otherwise the function chooses the candidate following the **Property 2**. The criterion says that we have to choose the candidate with the crosshatched region with the minimum area. In order to implement the condition, I wrote the following function that calculates the crosshatched region for a single candidate:

```
private int calculateCrosshatchedRegion(int x, int y, Node B){
    int dx = Math.abs(y-B.y);
    int dy = Math.abs(x-B.x);
    return Lx*dx + Ly*dy - 2*dx*dy;
}
```

The function calculates the distance from the candidate and the node to be deleted, in terms of x and y. Then calculates the region using also the variables `Lx` and `Ly`, that are the dimension of the space represented by the Quad Tree and they are set during the creation of a new tree. We can see the representation of a crosshatched region in Figure 3.

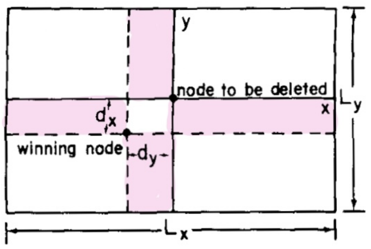

Figure 3: Crosshatched area

At this point the function `chooseCandidate` returns the node with the minimum area of the crosshatched region. After having found the candidate, the delete function replace the old node with the candidate and then performs the function `adj` with the two sub-quadrants adjacent to the quadrant (of the tree rooted at the node to be deleted) containing the candidate node.

ADJ The function starts examining the root of the quadrant and checks if the node is outside of the crosshatched region. In order to implement this check, I produced the following auxiliary function to check if a node is outside of the crosshatched region:

```
public boolean isOutsideOfCrossRegion(int x1, int y1, int x2, int y2) {
    int xMin = Math.min(x1, x2);
    int xMax = Math.max(x1, x2);
    int yMin = Math.min(y1, y2);
    int yMax = Math.max(y1, y2);
    return !((x>=xMin & x<=xMax) | (y>=yMin & y<=yMax));
}
```

If the node, let's say J, is outside of the crosshatched region then two sub-quadrants can remain in the quadrant rooted at J. The remaining sub-quadrants are recursively processed using `adj`. In this case, the two sub-quadrants that have to be recursively processed are the two ones that are inside the crosshatched region. The position of these two sub-quadrants depends about the position of the quadrant rooted at J and also about the position of the crosshatched region, i.e. the position of the winner candidate that replaced the deleted node. I noticed that if the winner candidate falls on quadrant i and J are on quadrant j, then we need to recursively processed sub-quadrants *conjugate(i)* and *conjugate(j)*. Going back to the first check, if J is inside the crosshatched region than the function deletes the sub-tree rooted at J and re-inserts all the sub-tree in the quad tree that was rooted at the deleted node. I implemented the following function in order to re-insert a sub-tree in another tree:

```java
public void reinsertQuadrant(Node root, Node j) {
    if (j!=null){
        n_re_insertion++;
        insert(j.x, j.y, j.value);
        reinsertQuadrant(root, j.NE);
        reinsertQuadrant(root, j.NW);
        reinsertQuadrant(root, j.SW);
        reinsertQuadrant(root, j.SE);
    }
}
```

Once the nodes in the quadrants adjacent to the quadrant rooted at the winning candidate have been processed, the algorithm process the nodes inside that quadrant using the function `newRoot`.

NEW ROOT Let's call i the quadrant rooted at the winning candidate. The function applies `adj` to the sub-quadrants adjacent to sub-quadrant i and then recursively re-applies `newRoot` to sub-quadrant *conjugate(i)*. Notice that the direction we're moving is always the same (i). This is done until the sub-quadrant *conjugate(i)* doesn't exist, at this point we're at the winning node, the node replacing the deleted node. At this point the function re-inserts the sub-quadrants adjacent to sub-quadrant i in the quadrants adjacent to quadrant i of the three rooted at the deleted node (using the `reinsertQuadrant` function). Then the function deletes the replacing node from his parent and then make the sub-quadrant i of the tree rooted at the winning candidate replace sub-quadrant *conjugate(i)* of the previous father node of the candidate (always using the `reinsertQuadrant` function).

Delete Below the code of the `delete` function, that performs all the steps explained in the previous paragraphs:

```
public void delete(Node nodeToBeDeleted) {
    Node candidate = findCandidate(nodeToBeDeleted);
    int direction = getStartDirection(nodeToBeDeleted, candidate);
    Node newRoot = replaceNode(nodeToBeDeleted, candidate);

    if (nodeToBeDeleted.parent == null)
        // nodeToBeDeleted is root
        root = newRoot;
    else {
        // nodeToBeDeleted is not root, so it has a parent
        String quad = dir2quad.get(nodeToBeDeleted.direction);
        if (quad.equals("NE")) nodeToBeDeleted.parent.NE = newRoot;
        else if (quad.equals("NW")) nodeToBeDeleted.parent.NW = newRoot;
        else if (quad.equals("SW")) nodeToBeDeleted.parent.SW = newRoot;
        else if (quad.equals("SE")) nodeToBeDeleted.parent.SE = newRoot;
    }

    Node[] quadsForAdj = getClosedQuadrants(nodeToBeDeleted, direction);
    for (Node quad: quadsForAdj)
        adj(nodeToBeDeleted, candidate, newRoot, quad, quad.direction,
            direction);

    newRoot(nodeToBeDeleted, candidate, newRoot,
        nodeToBeDeleted.getQuadFromDir(direction), direction);
}
```

Other auxiliary functions I have implemented other auxiliary functions, used many times in the main ones:

- `public Node getQuadFromDir(int i)`: this function is from the `Node` class and simply returns the proper quadrant son depending from the argument.
- `public String toString()`: produces an output string for the node object.
- `public void insert(int x, int y, int value)`: this function comes from the initial implementation of the Quad Trees [2]. The function inserts a new node inside the tree.
- `public void deleteChildFromParent(Node child, int direction)`: deletes the *child* node from its parent
- `private Node[] getClosedQuadrants(Node j, int direction)`: having the node *j*, the function returns the two sub-quadrants, rooted at *j*, closed to the sub-quadrant of the *direction* gave in the arguments.
- `private Node replaceNode(Node a, Node b)`: creates a new node with *x*, *y* and *value* from the *b* node and with *direction* and *parent* from the *a* node. The function also set the children quadrant as the ones from *a* node. I use this function at the beginning of the `delete` function to replace the node to be deleted with the winning candidate.
- `public Node searchNode(int x, int y, int value)`: searches a node with these values inside the quad tree
- `private int getStartDirection(Node root, Node toFind)`: searches the node *toFind* inside the sub-trees rooted at *root* and returns the number of the quadrants in which the node is found.

3 Experiments and Results

The objective of the new algorithm for deletion is to reduce the number of nodes that need to be re-inserted after the deletion of a node. When deleting a node J, the classical method re-inserts all the sub-tree rooted at J inside the quad tree. I performed an experiment in order to confirm that new algorithm reduces the number of re-insertions. I created two quad trees of the same dimension and then I inserted several nodes with random values inside the trees. At this point I chose a random node of the ones generated before and:

- I delete that node from one tree with using the algorithm that I implemented
- I delete the node from the other tree with the classical method, i.e. re-inserting all the sub-tree inside the quad tree.

I performed this test several times, in order to collect data about the number of re-insertions. Talking about the implementation, I modified the code of `QuadTree` and added a variable that counts the number of re-insertions. I increment the variable every time the `reinsertQuadrant` function is called, because is the one that actually re-insert the nodes when a node is deleted. Having nodes with random values and choosing a random node allowed me to have trees of different shapes and also allowed me to delete nodes that every time are in different levels of the tree.

After collecting the data, I created a python notebook in order to plot them. As we can see from Figure 4, the experiment confirms the expectations: using the new algorithm for deletion the number of re-insertions is much lower then using the classsical method.

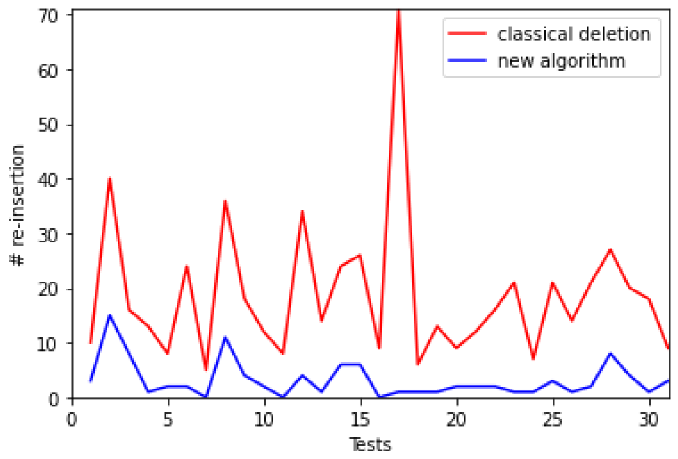

Figure 4: Number of re-insertions comparison

4 Observations

From my point of view the algorithm works well because it handles the problem in the same way as is handled with binary search trees. The algorithm tries to find the "closest" node that will replace the node to be deleted and in this way entire parts of the quad tree remain untouched, re-inserting a lower number of nodes in the quad tree. The implementation of the algorithm follows faithfully the steps explained in the article [1]. I had to implement some extra features that is not clearly specified in the algorithm. The most challenging parts was moving along the tree in the right direction, especially when the algorithm has to recursively perform `adj` only on two sub-quadrants. The implementation of the algorithm can obviously be improved, making some refactoring of the code and adding some extra useful functions. At the moment the algorithm can perform the deletion only on non-leave nodes, as it's explained in the article, so the implementation can be made more stable adding this feature.

5 Conclusion

To sum up, I produced a Java implementation of **Quad Trees**, with operations of insertion, visit of the tree and especially **deletion**. The *delete* operation follows the method explained in the article [1], that replace the node to be deleted with its "closest" node, with the goal of reducing the number of re-insertions. After implementing it, I performed an experiment in order to confirm the expectations, and the results that I obtained verify them. After producing the results, I created a python notebook in order to create a plot that compares the number of re-insertions between the new algorithm and the classical method. The new algorithm for deletion substantially reduces the number of re-insertions.

6 Appendix

QuadTree

```java
import java.util.HashMap;

public class QuadTree{

    public Node root;
    private final int Lx;
    private final int Ly;
    public int n_re_insertion;

    private static final HashMap<String, Integer> quad2dir;
    static {
        quad2dir = new HashMap<>();
        quad2dir.put("NE", 1);
        quad2dir.put("NW", 2);
        quad2dir.put("SW", 3);
        quad2dir.put("SE", 4);
    }

    private static final HashMap<Integer, String> dir2quad;
    static {
        dir2quad = new HashMap<>();
        dir2quad.put(1, "NE");
        dir2quad.put(2, "NW");
        dir2quad.put(3, "SW");
        dir2quad.put(4, "SE");
    }

    public QuadTree(int x, int y) {
        root = null;
        Lx = x;
        Ly = y;
        n_re_insertion = 0;
    }

    public static class Node {
        int x, y;             // x- and y- coordinates
        Node NW, NE, SE, SW; // four subtrees
        int value; // associated data
        int direction; // direction of quadrant (1=NE, 2=NW, 3=SW, 4=SE)
        Node parent; // parent node

        Node(int x, int y, int value, int direction, Node parent) {
            this.x = x;
            this.y = y;
            this.value = value;
            this.direction = direction;
            this.parent = parent;
        }

        public Node getQuadFromDir(int i){
            if (i==1) return NE;
            else if (i==2) return NW;
            else if (i==3) return SW;
            else if (i==4) return SE;
            else return null;
        }

        @Override
        public String toString(){
            return "[ X=" + x + " Y=" + y + "] value=" + (char)
                 (value+65) + " direction=" + direction
                    + "   ( parent=" + parent + " )";
        }

        public boolean isOutsideOfCrossRegion(int x1, int y1, int x2,
             int y2) {
            int xMin = Math.min(x1, x2);
            int xMax = Math.max(x1, x2);
            int yMin = Math.min(y1, y2);
            int yMax = Math.max(y1, y2);
            return !((x>=xMin & x<=xMax) | (y>=yMin & y<=yMax));
        }
    }

    /***********************************************************************
     *  Insertion
     ***********************************************************************/

    public void insert(int x, int y, int value) {
        if (x <= Lx & y <= Ly)
            root = insert(root, x, y, value, 0, null);
    }

    private Node insert(Node h, int x, int y, int value, int direction,
         Node parent) {
        if (h == null)
            return new Node(x, y, value, direction, parent);
        else if ( less(x, h.x) && less(y, h.y)) h.SW = insert(h.SW, x,
             y, value, 3, h); // SW
        else if ( less(x, h.x) && !less(y, h.y)) h.NW = insert(h.NW, x,
             y, value, 2, h); // NW
        else if (!less(x, h.x) && less(y, h.y)) h.SE = insert(h.SE, x,
             y, value, 4, h); // SE
        else if (!less(x, h.x) && !less(y, h.y)) h.NE = insert(h.NE, x,
             y, value, 1, h); // NE
        return h;
```

```
}

/*********************************************************************
 *  Deletion
 **********************************************************************/

public void delete(Node nodeToBeDeleted) {
    Node candidate = findCandidate(nodeToBeDeleted);
    int direction = getStartDirection(nodeToBeDeleted, candidate);
    Node newRoot = replaceNode(nodeToBeDeleted, candidate);

    if (nodeToBeDeleted.parent == null)
        // nodeToBeDeleted is root
        root = newRoot;
    else {
        // nodeToBeDeleted is not root, so it has a parent
        String quad = dir2quad.get(nodeToBeDeleted.direction);
        if (quad.equals("NE")) nodeToBeDeleted.parent.NE = newRoot;
        else if (quad.equals("NW")) nodeToBeDeleted.parent.NW =
             newRoot;
        else if (quad.equals("SW")) nodeToBeDeleted.parent.SW =
             newRoot;
        else if (quad.equals("SE")) nodeToBeDeleted.parent.SE =
             newRoot;
    }

    Node[] quadsForAdj = getClosedQuadrants(nodeToBeDeleted,
         direction);
    for (Node quad: quadsForAdj)
        adj(nodeToBeDeleted, candidate, newRoot, quad,
             quad.direction, direction);

    newRoot(nodeToBeDeleted, candidate, newRoot,
         nodeToBeDeleted.getQuadFromDir(direction), direction);
}

/*********************************************************************
 *  ADJ and NEWROOT for Deletion
 **********************************************************************/

private void adj(Node nodeToBeDeleted, Node candidate, Node newRoot,
     Node j, int quad_direction, int start_direction) {

    // if J in inside the Crosshatched Region do ADJ just on 2
         sub-quadrants
    System.out.println("\n***ADJ*** " + j);
    if (j!=null){
        if (j.isOutsideOfCrossRegion(nodeToBeDeleted.x,
             nodeToBeDeleted.y, candidate.x, candidate.y)){
            System.out.println("outside");
            if (j.getQuadFromDir(conjugate(start_direction))!=null)
                adj(nodeToBeDeleted, candidate, newRoot,
                     j.getQuadFromDir(conjugate(start_direction)),
                     quad_direction, start_direction);
            if (j.getQuadFromDir(conjugate(quad_direction))!=null)
                adj(nodeToBeDeleted, candidate, newRoot,
                     j.getQuadFromDir(conjugate(quad_direction)),
                     quad_direction, start_direction);
        }
        else {
            System.out.println("inside");

            // delete quadrant i (subtree rooted at J) from the parent
            deleteChildFromParent(j, j.direction);

            // reinsert all quadrant i (subtree rooted at J) on tree
                that was rooted on A
            reinsertQuadrant(newRoot, j);
        }
    }
}

private void newRoot(Node nodeToBeDeleted, Node candidate, Node
     newRoot, Node rootOfQuadrant, int direction) {
    System.out.println("\n***NEW ROOT***\n");
    // process nodes in quadrant i
    // i = direction

    // apply ADJ to the sub-quadrants adjacent to sub-quadrant i
    System.out.println("Root quad: " + rootOfQuadrant);
    Node[] subquadrants = ((direction == 1) | (direction == 3)) ?
            new Node[]{rootOfQuadrant.NW, rootOfQuadrant.SE} : new
                Node[]{rootOfQuadrant.NE, rootOfQuadrant.SW};
    for (Node subq : subquadrants){
        System.out.println("Subq: " + subq);
        if (subq!=null)
            adj(nodeToBeDeleted, candidate, newRoot, subq,
                conjugate(subq.direction), direction);
    }

    Node[] subqsToBeReinserted;
    Node[] subqsWhereToInsert;

    // apply NEWROOT to sub-quadrant conjugate(i)
    if (rootOfQuadrant.getQuadFromDir(conjugate(direction)) != null)
        newRoot(nodeToBeDeleted, candidate, newRoot,
             rootOfQuadrant.getQuadFromDir(conjugate(direction)),
             direction);
    else{
        subqsToBeReinserted = getClosedQuadrants(candidate,
            direction);
        subqsWhereToInsert = getClosedQuadrants(nodeToBeDeleted,
            direction);

        for (int i=0;i<2;i++)
            reinsertQuadrant(subqsWhereToInsert[i],
                subqsToBeReinserted[i]);

        // delete B from its parent
        deleteChildFromParent(candidate, conjugate(direction));

        Node sub_quadrant_i = candidate.getQuadFromDir(direction);
        reinsertQuadrant(candidate.parent.getQuadFromDir(conjugate(direction)),
            sub_quadrant_i);
    }
}

public void reinsertQuadrant(Node root, Node j) {
    if (j!=null){
        n_re_insertion++;
        insert(j.x, j.y, j.value);
        reinsertQuadrant(root, j.NE);
        reinsertQuadrant(root, j.NW);
        reinsertQuadrant(root, j.SW);
        reinsertQuadrant(root, j.SE);
    }
}

public void deleteChildFromParent(Node child, int direction){
    String quad = dir2quad.get(direction);
    switch (quad) {
        case "NE":
            child.parent.NE = null; break;
        case "NW":
            child.parent.NW = null; break;
        case "SW":
            child.parent.SW = null; break;
        case "SE":
            child.parent.SE = null; break;
    }
}

private Node[] getClosedQuadrants(Node j, int direction) {
    // return sub-quadrants close to sub-quadrant direction, inside
        quadrant of J
    String quad = dir2quad.get(direction);
    if (quad.equals("NE") | quad.equals("SW"))
        return new Node[]{j.NW, j.SE};
    else
        return new Node[]{j.NE, j.SW};
}

/***********************************************************************
 *  Property 1 and 2 for Deletion
 ***********************************************************************/

private int conjugate(int n) {
    return ((n+1)%4)+1;
}

private Node findCandidate(Node nodeToBeDeleted) {
    Node[] candidates = new Node[4];
    for (int i=0; i<4; i++){
        Node quad = nodeToBeDeleted.getQuadFromDir(i+1);
        candidates[i] = quad==null? null : findCandidate(quad,
             conjugate(i+1));
        System.out.println("Candidate from " + dir2quad.get(i+1) + ":
             " + candidates[i]);
    }
    return chooseCandidate(candidates, nodeToBeDeleted.x,
         nodeToBeDeleted.y);
}

private Node findCandidate(Node node, int direction){
    try {
        while(node.getQuadFromDir(direction) != null) {
            node = node.getQuadFromDir(direction);
        }
        return node;
    } catch (NullPointerException npe){
        return node;
    }
}

private Node chooseCandidate(Node[] candidates, int x, int y) {

    // Property 1
    boolean isNECloser = isCloserToAxis(x,y,
            candidates[quad2dir.get("NE")-1],
            candidates[quad2dir.get("SE")-1],
            candidates[quad2dir.get("NW")-1]);
    boolean isNWCloser = isCloserToAxis(x,y,
            candidates[quad2dir.get("NW")-1],
            candidates[quad2dir.get("SW")-1],
            candidates[quad2dir.get("NE")-1]);
    boolean isSWCloser = isCloserToAxis(x,y,
            candidates[quad2dir.get("SW")-1],
            candidates[quad2dir.get("NW")-1],
            candidates[quad2dir.get("SE")-1]);
    boolean isSECloser = isCloserToAxis(x,y,
            candidates[quad2dir.get("SE")-1],
            candidates[quad2dir.get("NE")-1],
            candidates[quad2dir.get("SW")-1]);

    int c=0;
    Node candidate = null;

    if (isNECloser){
        c=+1;
        candidate = candidates[quad2dir.get("NE")-1];
    }
    if (isNWCloser){
        c=+1;
        candidate = candidates[quad2dir.get("NW")-1];
    }
    if (isSWCloser){
        c=+1;
        candidate = candidates[quad2dir.get("SW")-1];
    }
    if (isSECloser){
        c=+1;
        candidate = candidates[quad2dir.get("SE")-1];
    }

    if (c==1)
        return candidate;
    else{
        // Property 2
        int minRegion = Integer.MAX_VALUE;
        int index = 0;
        for(int i=0; i<candidates.length;i++){
            int currRegion = calculateCrosshatchedRegion(x, y,
                 candidates[i]);
            if (currRegion < minRegion){
                minRegion = currRegion;
                index = i;
            }
        }
        return candidates[index];
    }
}

private boolean isCloserToAxis(int x, int y, Node candidate, Node
     first, Node second){
    return (Math.abs(x-candidate.x) < Math.abs(x-first.x) &
         Math.abs(y-candidate.y) < Math.abs(y-second.y));
}
```

```java
private int calculateCrosshatchedRegion(int x, int y, Node B){
    int dx = Math.abs(y-B.y);
    int dy = Math.abs(x-B.x);
    return Lx*dx + Ly*dy - 2*dx*dy;
}

private Node replaceNode(Node a, Node b){
    Node ret = new Node(b.x, b.y, b.value, a.direction, a.parent);
    // setting children
    ret.NE = a.NE;
    ret.NW = a.NW;
    ret.SW = a.SW;
    ret.SE = a.SE;
    // changing parent of children
    ret.NE.parent = ret;
    ret.NW.parent = ret;
    ret.SW.parent = ret;
    ret.SE.parent = ret;
    return ret;
}

/************************************************************************
 *  DFS & Search
 ************************************************************************/

public void dfs(){
    dfs(root, "root", 0);
}

private static void dfs(Node node, String quad, int level){
    if (node!=null){
        System.out.printf("LVL %d %s [X=%d, Y=%d] value=%d\n", level,
            quad, node.x, node.y, node.value);
        dfs(node.NE, "NE", level+1);
        dfs(node.NW, "NW", level+1);
        dfs(node.SW, "SW", level+1);
        dfs(node.SE, "SE", level+1);
    }
}

public Node searchNode(int x, int y, int value){
    return searchNode(root, x, y, value);
}

public Node searchNode(Node curr, int x, int y, int value){
    if (curr!=null)
        if (curr.x==x && curr.y==y && curr.value==value)
            return curr;
        else if ( less(x, curr.x) && less(y, curr.y)) return
            searchNode(curr.SW, x, y, value); // SW
            else if ( less(x, curr.x) && !less(y, curr.y)) return
                searchNode(curr.NW, x, y, value); // NW
            else if (!less(x, curr.x) && less(y, curr.y)) return
                searchNode(curr.SE, x, y, value); // SE
            else if (!less(x, curr.x) && !less(y, curr.y)) return
                searchNode(curr.NE, x, y, value); // NE
        return curr;
    }

    private int getStartDirection(Node root, Node toFind){
        if (searchNode(root.NE, toFind.x, toFind.y, toFind.value)!=null)
            return 1;
        if (searchNode(root.NW, toFind.x, toFind.y, toFind.value)!=null)
            return 2;
        if (searchNode(root.SW, toFind.x, toFind.y, toFind.value)!=null)
            return 3;
        if (searchNode(root.SE, toFind.x, toFind.y, toFind.value)!=null)
            return 4;
        return -1;
    }

    /***************************************************************************
     *  helper comparison functions
     ***************************************************************************/

    private boolean less(int k1, int k2) { return k1 < k2; }
}
```

Test

```
import java.io.File;
import java.io.FileWriter;

public class Test {
    public static void main(String[] args){
        int N = 100;
        File myObj = new File("results.txt");

        for (int j=0;j<1000;j++){
            try{
                QuadTree qt = new QuadTree(N, N);
                QuadTree qt2 = new QuadTree(N,N);
                int[][] nodes = new int[N][3];

                for (int i = 0; i < N; i++) {
                    Integer x = (int)(Math.random() * N + 1);
                    Integer y = (int)(Math.random() * N + 1);
                    nodes[i] = new int[]{x, y, i};
                }
                for (int[] node: nodes){
                    qt.insert(node[0], node[1], node[2]);
                    qt2.insert(node[0], node[1], node[2]);
                }

                int[] nodeToBeDeleted = nodes[(int) (100 *
                    Math.random())];
                int x = nodeToBeDeleted[0];
                int y = nodeToBeDeleted[1];
                int v = nodeToBeDeleted[2];
                qt.delete(qt.searchNode(x,y,v));
                QuadTree.Node node = qt2.searchNode(x,y,v);
                qt2.deleteChildFromParent(node, node.direction);
                qt2.reinsertQuadrant(qt2.root, node.SE);
                qt2.reinsertQuadrant(qt2.root, node.SW);
                qt2.reinsertQuadrant(qt2.root, node.NE);
                qt2.reinsertQuadrant(qt2.root, node.NW);

                FileWriter myWriter = new FileWriter("results.txt", true);
                myWriter.write(qt.n_re_insertion + ";" +
                    qt2.n_re_insertion + "\n");
                myWriter.close();
            } catch (Exception e){
                continue;
            }
        }
    }
}
```

Plot

```
import matplotlib.pyplot as plt

def get_axes(filename, deletion):

    file = open(filename, "r")

    ys = []
    for row in file.readlines():
        x,y = row.split(';')
        if deletion:
            print(x)
            ys.append(int(x))
        else:
            ys.append(int(y[:-1]))
    return ys

xs = list(range(1,32))
ys_deletion = get_axes("results.txt", True)
ys_reinsertion = get_axes("results.txt", False)

ax = plt.gca()
ax.set_xlim([0, 31])
ax.set_ylim([0, max(max(ys_deletion), max(ys_reinsertion))])
plt.xlabel("Tests")
plt.ylabel("# re-insertion")
plt.plot(xs, ys_reinsertion, color='red', label='classical deletion')
plt.plot(xs, ys_deletion, color='blue', label='new algorithm')
plt.legend(loc="upper right")
```

References

[1] Deletion in Two-Dimensional Quad Trees:
https://dl.acm.org/doi/pdf/10.1145/359038.359043

[2] QuadTree:
https://algs4.cs.princeton.edu/92search/QuadTree.java.html