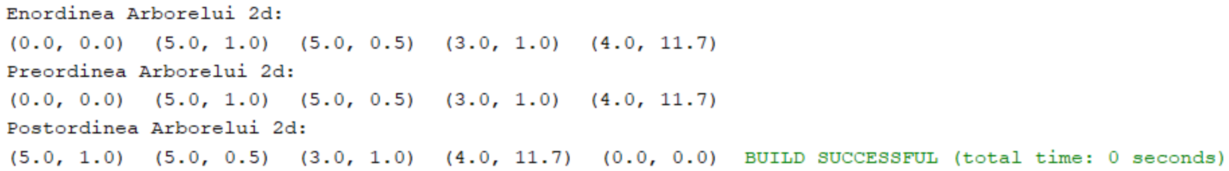
Tudor Vlăduț-Alexandru

Problema 8

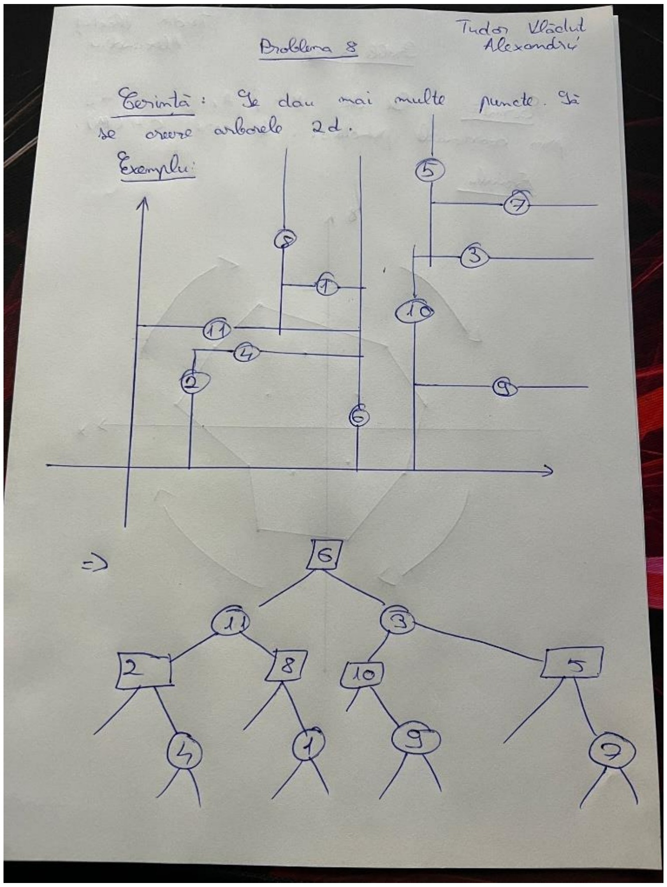
Arbore 2d

Rulaj:



```
Enordinea Arborelui 2d:
(0.0, 0.0)  (5.0, 1.0)  (5.0, 0.5)  (3.0, 1.0)  (4.0, 11.7)
Preordinea Arborelui 2d:
(0.0, 0.0)  (5.0, 1.0)  (5.0, 0.5)  (3.0, 1.0)  (4.0, 11.7)
Postordinea Arborelui 2d:
(5.0, 1.0)  (5.0, 0.5)  (3.0, 1.0)  (4.0, 11.7)  (0.0, 0.0)  BUILD SUCCESSFUL (total time: 0 seconds)
```

Pe foaie:



Tudor Vlăduț
Alexandru

Problema 8

Cerință: Se dau mai multe puncte. Să se creeze arborele 2d.

Exemplu:

Implementare Java:

```
package problema8;
import java.io.IOException;
class KD2DNode
{
    int axis;
    double[] x;
    int id;
    boolean checked;
    boolean orientation;
    KD2DNode Parent;
    KD2DNode Left;
    KD2DNode Right;
    public KD2DNode(double[] x0, int axis0)
    {
        x = new double[2];
        axis = axis0;
        for (int k = 0; k < 2; k++)
            x[k] = x0[k];
        Left = Right = Parent = null;
        checked = false;
        id = 0;
    }
    public KD2DNode FindParent(double[] x0)
    {
        KD2DNode parent = null;
        KD2DNode next = this;
        int split;
        while (next != null)
        {
            split = next.axis;
            parent = next;
            if (x0[split] > next.x[split])
                next = next.Right;
            else
                next = next.Left;
        }
        return parent;
    }

    public KD2DNode Insert(double[] p)
    {
        x = new double[2];
        KD2DNode parent = FindParent(p);
        if (equal(p, parent.x, 2) == true)
            return null;
        KD2DNode newNode = new KD2DNode(p, parent.axis + 1 < 2 ? parent.axis + 1 : 0);
        newNode.Parent = parent;
        if (p[parent.axis] > parent.x[parent.axis])
        {
            parent.Right = newNode;
            newNode.orientation = true; //
        } else
        {
            parent.Left = newNode;
            newNode.orientation = false; //
        }
        return newNode;
    }
    boolean equal(double[] x1, double[] x2, int dim)
    {
        for (int k = 0; k < dim; k++)
        {
            if (x1[k] != x2[k])
```

```
                return false;
            }
            return true;
        }
        double distance2(double[] x1, double[] x2, int dim)
        {
            double S = 0;
            for (int k = 0; k < dim; k++)
                S += (x1[k] - x2[k]) * (x1[k] - x2[k]);
            return S;
        }
    }
    class KD2DTree
    {
        KD2DNode Root;
        int TimeStart, TimeFinish;
        int CounterFreq;
        double d_min;
        KD2DNode nearest_neighbour;
        int KD_id;
        int nList;
        KD2DNode CheckedNodes[];
        int checked_nodes;
        KD2DNode List[];
        double x_min[], x_max[];
        boolean max_boundary[], min_boundary[];
        int n_boundary;

        public KD2DTree(int i)
        {
            Root = null;
            KD_id = 1;
            nList = 0;
    List = new KD2DNode[i];
    CheckedNodes = new KD2DNode[i];
    max_boundary = new boolean[2];
    min_boundary = new boolean[2];
    x_min = new double[2];
    x_max = new double[2];
}

public boolean add(double[] x)
{
    if (nList >= 2000000 - 1)
        return false;

    if (Root == null)
    {
        Root = new KD2DNode(x, 0);
        Root.id = KD_id++;
        List[nList++] = Root;
    } else
    {
        KD2DNode pNode;
        if ((pNode = Root.Insert(x)) != null)
        {
            pNode.id = KD_id++;
            List[nList++] = pNode;
        }
    }

    return true;
}

public KD2DNode find_nearest(double[] x)
{
    if (Root == null)
        return null;
    checked_nodes = 0;
    KD2DNode parent = Root.FindParent(x);
    nearest_neighbour = parent;
    d_min = Root.distance2(x, parent.x, 2);
    if (parent.equal(x, parent.x, 2) == true)
        return nearest_neighbour;
    search_parent(parent, x);
    uncheck();
    return nearest_neighbour;
}

public void check_subtree(KD2DNode node, double[] x)
{
    if ((node == null) || node.checked)
        return;
    CheckedNodes[checked_nodes++] = node;
    node.checked = true;
    set_bounding_cube(node, x);
    int dim = node.axis;
    double d = node.x[dim] - x[dim];
    if (d * d > d_min)
    {
        if (node.x[dim] > x[dim])
            check_subtree(node.Left, x);
        else
            check_subtree(node.Right, x);
    } else
    {
        check_subtree(node.Left, x);
        check_subtree(node.Right, x);
    }
```

```
}
public void set_bounding_cube(KD2DNode node, double[] x)
{
    if (node == null)
        return;
    int d = 0;
    double dx;
    for (int k = 0; k < 2; k++)
    {
        dx = node.x[k] - x[k];
        if (dx > 0)
        {
            dx *= dx;
            if (!max_boundary[k])
            {
                if (dx > x_max[k])
                    x_max[k] = dx;
                if (x_max[k] > d_min)
                {
                    max_boundary[k] = true;
                    n_boundary++;
                }
            }
        } else
        {
            dx *= dx;
            if (!min_boundary[k])
            {
                if (dx > x_min[k])
                    x_min[k] = dx;
                if (x_min[k] > d_min)
                {
                    min_boundary[k] = true;
                    n_boundary++;
                }
            }
        }
        d += dx;
        if (d > d_min)
            return;
    }
    if (d < d_min)
    {
        d_min = d;
        nearest_neighbour = node;
    }
}

public KD2DNode search_parent(KD2DNode parent, double[] x)
{
    for (int k = 0; k < 2; k++)
    {
        x_min[k] = x_max[k] = 0;
        max_boundary[k] = min_boundary[k] = false; //
    }
    n_boundary = 0;
    KD2DNode search_root = parent;
    while (parent != null && (n_boundary != 2 * 2))
    {
        check_subtree(parent, x);
        search_root = parent;
        parent = parent.Parent;
    }
    return search_root;
}
```

```
public void uncheck()
{
    for (int n = 0; n < checked_nodes; n++)
        CheckedNodes[n].checked = false;
}

public void inorder()
{
    inorder(Root);
}

private void inorder(KD2DNode root)
{
    if (root != null)
    {
        inorder(root.Left);
        System.out.print("(" + root.x[0] + ", " + root.x[1] + ")  ");
        inorder(root.Right);
    }
}

public void preorder()
{
    preorder(Root);
}

private void preorder(KD2DNode root)
{
    if (root != null)
    {
        System.out.print("(" + root.x[0] + ", " + root.x[1] + ")  ");
        inorder(root.Left);
        inorder(root.Right);
        }
    }

    public void postorder()
    {
        postorder(Root);
    }

    private void postorder(KD2DNode root)
    {
        if (root != null)
        {
            inorder(root.Left);
            inorder(root.Right);
            System.out.print("(" + root.x[0] + ", " + root.x[1] + ")  ");
        }
    }
}

public class KDTree_TwoD_Data
{
    public static void main(String args[]) throws IOException
    {
        int numpoints = 5;
        KD2DTree kdt = new KD2DTree(numpoints);
        double x[] = new double[2];
        x[0] = 0.0;
        x[1] = 0.0;
        kdt.add(x);
        x[0] = 3;
        x[1] = 1;
        kdt.add(x);
        x[0] = 4;
        x[1] = 11.7;
        kdt.add(x);
        x[0] = 5;
        x[1] = 0.5;
        kdt.add(x);
        x[0] = 5;
        x[1] = 1;
        kdt.add(x);
        System.out.println("Enordinea Arborelui 2d: ");
        kdt.inorder();
        System.out.println("\nPreordinea Arborelui 2d: ");
        kdt.preorder();
        System.out.println("\nPostordinea Arborelui 2d: ");
        kdt.postorder();
    }
}
```