实验报告

一、设计思路

1. 范围查询：

为了进行范围查询（查询某个范围内有多少点），设计了一个 Rectangle 类，用于描述每个 KDTree 结点代表的范围。然后从根节点及其对应的范围开始遍历，如果当前节点代表的范围 cell 在目标范围 R 内，则当前节点的所有子树上的节点都在 R 中，就无需再遍历子树，直接加上结点数即可；如果 cell 与 R 相交，则需要递归在左右子树中继续遍历；如果 cell 与 R 没有交集，则当前节点的所有子树上的节点都不在 R 中，也无需遍历了。这样就起到了剪枝的效果，降低了时间复杂度。

```
int rangeCount(Rectangle R, KDNode p, Rectangle cell){  //返回范围cell中的点在范围R内的数量
    if(p.location == NIL) return 0;
    else if(R.isDIsjointFrom(cell))
        return 0;
    else if(R.contains(cell))  //如果cell在R内，那么p的所有子树的结点均在R内
        return p.size;
    else{   //如果相交
        int count = 0;
        if(R.contains(P[p.location])) count+=1;  //如果R区域包含p点，计数

        //左子树不为空时，递归在左子树计数
        if(p.l != NIL)
            count += rangeCount(R, T[p.l], cell.leftPart(p.dim, P[p.location]));
        //右子树不为空时，递归在右子树计数
        if(p.r != NIL)
            count += rangeCount(R, T[p.r], cell.rightPart(p.dim, P[p.location]));
        return count;
    }
}
```

2. 最近邻查询：

最近邻查询要查找与目标点 q 距离最近的点。虽然已经为每个 KDNode 结点分配了对应的范围 cell，但是最近邻不一定在 cell 内，所以需要回溯查找。首先仍然从根节点出发，计算当前节点与 q 的距离，并据此更新最近距离，然后将当前节点代表的范围分为左右两部分，并递归在左右子树中查找。这样，就会先查找到 q 所在的子树，从而更有可能先找到更接近 q 的点，从而起到剪枝的作用。此外，在递归查找左右子树时，优先查找离 q 更近的子树范围，然后据此判断另一棵子树时候还有必要遍历（如果在对应维度上的距离小于当前最好距离才遍历），也起到剪枝的效果。

```
data nearNeighbor(Point q, KDNode p, Rectangle cell, data bestDist){
    if(p.location != NIL){
        data tmp = q.distanceTo(P[p.location]);
        bestDist = min(tmp, bestDist);  //更新最小距离

        int dimension = p.dim;
        Rectangle leftCell = cell.leftPart(dimension, P[p.location]);
        Rectangle rightCell = cell.rightPart(dimension, P[p.location]);
        if(!dimension){
            if(q.x < P[p.location].x){  //如果q离left部分更近
                if(p.l != NIL)bestDist = nearNeighbor(q, T[p.l], leftCell, bestDist);
                if(rightCell.distanceTo(q) < bestDist){  //先更新最近的部分，再以此判断是否需要遍历另一部分
                    if(p.r != NIL)bestDist = nearNeighbor(q, T[p.r], rightCell, bestDist);
                }
            }else{
                if(p.r != NIL)bestDist = nearNeighbor(q, T[p.r], rightCell, bestDist);
                if(leftCell.distanceTo(q) < bestDist){
                    if(p.l != NIL)bestDist = nearNeighbor(q, T[p.l], leftCell, bestDist);
                }
            }
        }else{
            if(q.y < P[p.location].y){
                if(p.l != NIL)bestDist = nearNeighbor(q, T[p.l], leftCell, bestDist);
                if(rightCell.distanceTo(q) < bestDist){
                    if(p.r != NIL)bestDist = nearNeighbor(q, T[p.r], rightCell, bestDist);
                }
            }else{
                if(p.r != NIL)bestDist = nearNeighbor(q, T[p.r], rightCell, bestDist);
                if(leftCell.distanceTo(q) < bestDist){
                    if(p.l != NIL)bestDist = nearNeighbor(q, T[p.l], leftCell, bestDist);
                }
            }
        }

    }
    return bestDist;
}
```

二、使用的数据结构

1. Point 类 ： 表示一个点，有 x、y 坐标。distanceTo 函数计算当前点与 p 的距离

```
class Point{
    public:
        int x, y;
        Point() {}
        Point(int x, int y): x(x), y(y) {}
        data distanceTo(Point p){  //当前点与点p之间的距离
            return (data)(x-p.x)*(data)(x-p.x)+(data)(y-p.y)*(data)(y-p.y)
        }
};
```

2. KDNode 类：表示一个 KDTree 结点，具体意义如下：

```
class KDNode{   //KDTree的结点
    public:
        int location; //表示该节点对应的Point在列表中的位置
        int size;  //表示以该节点为根的子树的节点数
        int dim; //表示该结点的维度信息
        int p, l, r;  //指示父节点、左右子节点在KDNode列表中的位置
        KDNode() {}
};
```

3. Rectangle 类：用左下点和右上点描述每个 KDNode 结点代表的范围。还有 contains、isDisjointFrom 等函数计算该区域与某个点或某个区域的相交关系。leftPart 和 rightPart 将该范围分割成两部分。

```
class Rectangle{  //描述每个结点的范围
public:
    Point low;  //左下点
    Point high;  //右上点
    Rectangle(Point low1, Point high1): low(low1), high(high1){}
    bool contains(Point q){  //该范围是否包含点q …
    bool contains(Rectangle c){   //该范围是否包含区域c …
    bool isDIsjointFrom(Rectangle c){   //该范围与区域c是否没有交集 …
    Rectangle leftPart(int dim, Point s){   //在维度dim上，用s将该区域分割，返回左半部分 …
    Rectangle rightPart(int dim, Point s){  //在维度dim上，用s将该区域分割，返回右半部分 …
    data distanceTo(Point q){   //点q与该区域的距离，分多钟情况计算 …
};
```

为了节省空间，没有采用指针的方式表示，而是建立了 Point 和 KDNode 的两个数组用于存放所有的点和 KDTree 结点。结点对应的点在 P 中的位置和左右子节点在 T 中的位置都以下标的形式表示。

```
Point P[MAX];   //存放所有的Point
KDNode T[MAX];  //存放所有的KdTree结点
```

三、测试结果

按照说明修改部分代码，测试 yoj 790 和 791 题，均通过，证明程序正确。

#790 二维查找树	✔ Accepted	100	711 ms	928 KB	cpp / 3538 B	李馨雨(2020202279)	2022/4/12 16:41
#791 平面最近点查询	✔ Accepted	100	891 ms	3576 KB	cpp / 6684 B	李馨雨(2020202279)	2022/4/12 18:46

四、代码

代码详见“KDTree.cpp”。其中递归建立 KDTree 的代码如下，每次都选择中位数作为新的根节点，再递归建立左右子树。

```
int makeKDTree(int l, int r, int depth){   //建树
    if(!(l < r) ) return NIL;
    int mid = (l + r) / 2;
    int t = np++; //指示KDNode队列中的下标位置
    if(depth % 2 == 0){ //不同的维度按照不同规则排序
        sort(P + l, P + r, lessX);
        T[t].dim = 0;
    } else {
        sort(P + l, P + r, lessY);
        T[t].dim = 1;
    }
    T[t].location = mid; //下标t的结点对应的Point的位置在P队列中是mid
    T[t].size = r-l; //子树的节点个数
    T[t].l = makeKDTree(l, mid, depth + 1);  //递归建立左子树
    T[T[t].l].p = t;
    T[t].r = makeKDTree(mid + 1, r, depth + 1); //递归建立右子树
    T[T[t].r].p = t;

    return t;
}
```