**Data Structures**

**Spring 2020**

**Programming Homework#4**

**Date: 2020. 05. 27**

**ID/Name: 201533661 이승수**

**[PHW#4(kdTree Programming)] (60 points)**

**Using the source code provided as basis, implement the algorithms developed for Lab #4.**

**10 x 10 grid, Insert in order [ (2,3), (5,4), (3,4), (9,6), (4,7), (8,1), (7,2) ]**

**(1)point\_search for a user-specified point**

**-Point\_search point: [ (5,4), (4,7), (10,5) ]**

**(2) range\_search(find all points contained within a specified bounding rectangle)**

* **Range\_search range: xmin(6), ymin(3), width(3), height(4)**

**(3) nearest\_neighbor\_search(given a point, find one or more nearest neighbor points)**

* **Nearest\_neighbor\_search: [ (5,4), (4,7) ]**

**[My Code]**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define MAX\_DIM 2

#define N 1000000

#define rand1() (rand() / (double)RAND\_MAX)

#define rand\_pt(v) { v.x[0] = rand1(); v.x[1] = rand1(); v.x[2] = rand1(); }

struct kd\_node\_t {

double x[MAX\_DIM];

struct kd\_node\_t\* left, \* right;

};

/\* global variable, so sue me \*/

int visited;

/\* functions declaration \*/

void printKdTree(struct kd\_node\_t\* root);

struct kd\_node\_t\* search\_point(struct kd\_node\_t\* root, double x, double y, int depth);

void range\_search(struct kd\_node\_t\* root, double xmin, double ymin, double width, double height);

struct kd\_node\_t\* nearest\_neighbor\_search(struct kd\_node\_t\* nearest, struct kd\_node\_t\* point, int depth, struct kd\_node\_t\* root);

inline double dist(struct kd\_node\_t\* a, struct kd\_node\_t\* b, int dim);

inline void swap(struct kd\_node\_t\* x, struct kd\_node\_t\* y);

struct kd\_node\_t\* find\_median(struct kd\_node\_t\* start, struct kd\_node\_t\* end, int idx);

struct kd\_node\_t\* make\_tree(struct kd\_node\_t\* t, int len, int i, int dim);

void nearest(struct kd\_node\_t\* root, struct kd\_node\_t\* nd, int i, int dim, struct kd\_node\_t\*\* best, double\* best\_dist);

int main(void)

{

int i;

struct kd\_node\_t wp[] = {

{{2, 3}}, {{5, 4}}, {{9, 6}}, {{4, 7}}, {{8, 1}}, {{7, 2}}

};

struct kd\_node\_t\* root, \* found, \* million;

double best\_dist;

root = make\_tree(wp, sizeof(wp) / sizeof(wp[1]), 0, 2);

//printKdTree(root);

struct kd\_node\_t\* result;

printf("\n1. search user-specified points:\n");

// = search\_point(root, 7, 2, 0);//

result = search\_point(root, 5, 4, 0);

result = search\_point(root, 4, 7, 0);

result = search\_point(root, 10, 5, 0);

printf("\n2. find all points contained within a specified bounding rectangle:\n"); //xmin(6), ymin(3), width(3), height(4)

range\_search(root, 6, 3, 3, 4);

printf("\n3. given a point, find one or more nearest neighbor points:\n");//(5,4), (4,7)

struct kd\_node\_t myPoint = { {3,7} };

result = nearest\_neighbor\_search(root, &myPoint, 0, root);

//printf("myNearestPoint: (%g,%g)\n", result->x[0], result->x[1]);

struct kd\_node\_t givenPoint = { {5,4} };

found = 0;

nearest(root, &givenPoint, 0, 2, &found, &best\_dist);

printf("nearest Point from (%g,%g):(%g,%g),distance=%g\n", givenPoint.x[0], givenPoint.x[1], found->x[0], found->x[1], sqrt(best\_dist));

givenPoint.x[0] = 4; givenPoint.x[1] = 7;

found = 0; best\_dist = 0;

nearest(root, &givenPoint, 0, 2, &found, &best\_dist);

printf("nearest Point from (%g,%g):(%g,%g),distance=%g\n", givenPoint.x[0], givenPoint.x[1], found->x[0], found->x[1], sqrt(best\_dist));

printf("\nprogram end");

return 0;

}

inline double

dist(struct kd\_node\_t\* a, struct kd\_node\_t\* b, int dim)

{

double t, d = 0;

while (dim--) {

t = a->x[dim] - b->x[dim];

d += t \* t;

}

return d;

}

inline void swap(struct kd\_node\_t\* x, struct kd\_node\_t\* y) {

double tmp[MAX\_DIM];

memcpy(tmp, x->x, sizeof(tmp));

memcpy(x->x, y->x, sizeof(tmp));

memcpy(y->x, tmp, sizeof(tmp));

}

void printKdTree(struct kd\_node\_t\* root) {

printf("[x:%g, y:%g]\n", root->x[0], root->x[1]); //printf("[nodeID:%c, x:%d, y:%d]\n", root->nodeID, root->x, root->y);

if (root->left != NULL) {

printf("L");

printKdTree(root->left);

}

if (root->right != NULL) {

printf("R");

printKdTree(root->right);

}

}

struct kd\_node\_t\* search\_point(struct kd\_node\_t\* root, double x, double y, int depth) {

if (root == NULL) {

printf("(%g,%g) node cannote be searched\n", x, y);

return NULL;

}

else if (root->x[0] == x && root->x[1] == y) {

printf("(%g,%g) node is searched\n", x, y);

return root;

}

if (depth == 0 && root->x[0] > x)

return search\_point(root->left, x, y, (depth + 1) % 2);

else if (depth == 0 && root->x[0] < x)

return search\_point(root->right, x, y, (depth + 1) % 2);

else if (depth == 1 && root->x[1] > y)

return search\_point(root->left, x, y, (depth + 1) % 2);

else if (depth == 1 && root->x[1] < y)

return search\_point(root->right, x, y, (depth + 1) % 2);

else

return NULL;

}

void range\_search(struct kd\_node\_t\* root, double xmin, double ymin, double width, double height) {

if (xmin <= root->x[0] && root->x[0] <= (xmin + width)) {

if (ymin <= root->x[1] && root->x[1] <= (ymin + height)) {

printf("(%g, %g) node in range\n", root->x[0], root->x[1]);

}

}

if (root->left != NULL) {

range\_search(root->left, xmin, ymin, width, height);

}

if (root->right != NULL) {

range\_search(root->right, xmin, ymin, width, height);

}

}

struct kd\_node\_t\* nearest\_neighbor\_search(struct kd\_node\_t\* nearest, struct kd\_node\_t\* point, int depth, struct kd\_node\_t\* root) {

if (nearest->x[0] == point->x[0] && nearest->x[1] == point->x[1]) {

if (dist(nearest->left, point, 2) < dist(nearest->right, point, 2))

nearest = nearest->left;

else

nearest = nearest->right;

}

if (nearest->left == NULL && nearest->right == NULL)

return nearest;

if (point->x[depth] < nearest->x[depth] && nearest->left == NULL) {

return NULL;//

}

if (point->x[depth] > nearest->x[depth] && nearest->left == NULL) {

return NULL;//

}

else if (point->x[depth] > nearest->x[depth] && dist(point, nearest->left, 2) < dist(point, nearest, 2)) {

return NULL;//no

}

else if (point->x[depth] > nearest->x[depth] && dist(point, nearest->left, 2) > dist(point, nearest, 2)) {

return nearest;

}

else if (point->x[depth] > nearest->x[depth] && dist(point, nearest->right, 2) < dist(point, nearest, 2)) {

return nearest\_neighbor\_search(nearest->right, point, (depth + 1) % 2, root);//ok

}

else if (point->x[depth] > nearest->x[depth] && dist(point, nearest->right, 2) > dist(point, nearest, 2)) {

return nearest;

}

else if (dist(point, nearest->left, 2) == dist(point, nearest, 2)) {

printf("same distance\n");//

return NULL;

}//else

//nearest(root, point, 0, 2, nearest, 10);

return nearest;

}

/\* see quickselect method \*/

struct kd\_node\_t\*

find\_median(struct kd\_node\_t\* start, struct kd\_node\_t\* end, int idx)

{

if (end <= start) return NULL;

if (end == start + 1)

return start;

struct kd\_node\_t\* p, \* store, \* md = start + (end - start) / 2;

double pivot;

while (1) {

pivot = md->x[idx];

swap(md, end - 1);

for (store = p = start; p < end; p++) {

if (p->x[idx] < pivot) {

if (p != store)

swap(p, store);

store++;

}

}

swap(store, end - 1);

/\* median has duplicate values \*/

if (store->x[idx] == md->x[idx])

return md;

if (store > md) end = store;

else start = store;

}

}

struct kd\_node\_t\*

make\_tree(struct kd\_node\_t\* t, int len, int i, int dim)

{

struct kd\_node\_t\* n;

if (!len) return 0;

if ((n = find\_median(t, t + len, i))) {

i = (i + 1) % dim;

n->left = make\_tree(t, n - t, i, dim);

n->right = make\_tree(n + 1, t + len - (n + 1), i, dim);

}

return n;

}

void nearest(struct kd\_node\_t\* root, struct kd\_node\_t\* nd, int i, int dim,

struct kd\_node\_t\*\* best, double\* best\_dist)

{

double d, dx, dx2;

if (!root) return;

d = dist(root, nd, dim);

dx = root->x[i] - nd->x[i];

dx2 = dx \* dx;

visited++;

if (!\*best || d < \*best\_dist) {

\*best\_dist = d;

\*best = root;

}

/\* if chance of exact match is high \*/

if (!\*best\_dist) return;

if (++i >= dim) i = 0;

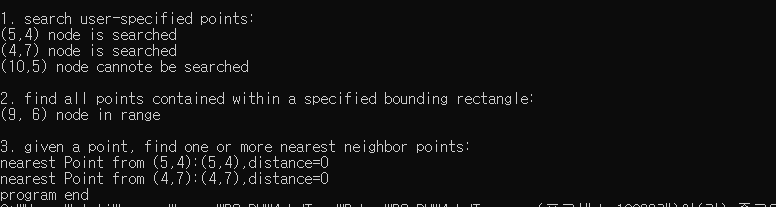
nearest(dx > 0 ? root->left : root->right, nd, i, dim, best, best\_dist);

if (dx2 >= \*best\_dist) return;

nearest(dx > 0 ? root->right : root->left, nd, i, dim, best, best\_dist);

}

**[Result]**



**[PHW#4(region QuadTree pseudocode)] (30 points)**

**Write an algorithm in pseudocode form for each of the following functions: (include the function header)**

**(1) define the node struct**

**(2) build\_tree(build a quad tree, given N points)**

**-Assume that each node does NOT store the bounding box information**

**(i.e., you must compute the information as you build the tree and also**

**later search for points)**

**-Assume the x, y values of all are integers between 1 and 20, and the**

**original map is 21 x 21.**

**(3) point\_search(search for a point that exists, and for a point that does not exist)**

**(2) range\_search(find all points contained within a specified bounding rectangle)**

**[My pseudocode]**

/\* I will rotate quad tree in clock-wise order\*/

**struct point** { int x; int y; }

**struct node** {

struct point pos;

struct node\*\* child[4]; }

**struct node build\_tree (node rootNode)** {

struct node nodeIn;

if array’s size >1:

for idx in range(4):

nodeIn=buildNode (rootNode.child[idx] ,n, k)

build\_tree(nodeln);

}

**Struct node point\_search(node rootNode, node pointNode, int depth)** {

Int width=20 / pow(2, depth)

If pointNode.pos == rootNode.pos:

Return rootNode; //return rootNode if searched.

For idx in range(4):

If distance( rootNode.child[idx], pointNode) < (width/2):

Return Point\_search(rootNode.child[idx], pointNode, depth);

Return Null; //return null if pointNode is not searched.

}

**Void range\_search(node rootNode, int xmin, int ymin, int xmax, int ymax)** {

If (xmin<rootNode.pos.x<xmax && ymin<rootNode.pos.y<ymax)

Print rootNode

For idx in range(4): //print nodes only in range while preorder traversal.

If rootNode.child[idx] != NULL:

Range\_search(rootNode.child[idx], xmin, ymin, xmax, ymax)

}