Assignment 5 Report
Jason Yang and Brian Giusti

==

Destroy Function for KDTree

==

The destroy function of the KD-Tree is to delete all nodes and data from the tree, clearing all the data. To do this we have a node (p) and start at the root. From there the function recursively calls itself to transverse itself down the tree, checking if p exists every time. We chose to work left to right. The function is optimized by deleting the leaf nodes first so no pointer need to be changed. From there the tree is deleted from the leaf nodes on the left up to the root. Then the same for the right side. Finally the root is deleted fully destroying the tree.

==

Insert Function for KDTree

==

The insert function adds items to the tree. This can be adding a single node to an existing tree or in our case creating a new tree. The first node in a empty tree is the root. From there the x coordinate is compared to the x coordinate of the root and transverses down the tree checking each nodes x coordinate. If it is less it is sent left, if greater then it goes right. For ones with equal x coordinates the y values are compared. This is done every time the function is called and a new node is added.

==

Printing Neighbors for KDTree

==

Printing neighbors is faster when using a 2D tree, because of the way the tree is structured. Each node divides the map into a rectangular section. At each level, the depth (I added a depth parameter the class) is checked to determine which direction to recurse in - that is, which parameter to check at each node: the longitude or the latitude. My recursive helper function helpPrint skips checking nodes that are outside of the square of length 2*radius that is centered on the center point. I use the distance function to compare a node’s latitude or longitude with that of the center. If that distance is greater than the radius, the function recurses in the direction that would move it closer to the center.

==
Performance Analysis Section
==

I tested the fetch times of the following queries 3 times each:

All times in seconds.

```
./search-map 64.15 -21.95 .3 food data/ice.txt >data/markers.js
```

Check for food around 64.15 -21.95 in a .3 mile radius

Linked List Speed	KD Tree Speed
0.004149	0.000988
0.005126	0.000284
0.005989	0.001023

```
./search-map 65.15 -18.95 5 e data/ice.txt >data/markers.js
```

Check for anything with the letter e in it for 5 miles around 65.15 -18.95 (deserted)

Linked List Speed	KD Tree Speed
0.004833	0.000144
0.004323	5.1e-05
0.00565	6.3e-05

```
./search-map 65 -19.95 30 inn data/ice.txt >data/markers.js
```

Check for “inn” within a 30 mile radius

Linked List Speed	KD Tree Speed
0.005467	0.000248
0.003751	0.000196

0.00459	0.000364

From the data table we can conclude that our tree runs in O(log n) time (true for all binary trees) for the best case scenario. Run time for the linked list version remains constant, since it must always check every element in the list. The KD tree version excels when the radius is smaller, because it can isolate the surrounding area due to the way it is ordered. Compare this to the linked list where the runtime is always O(n) time.

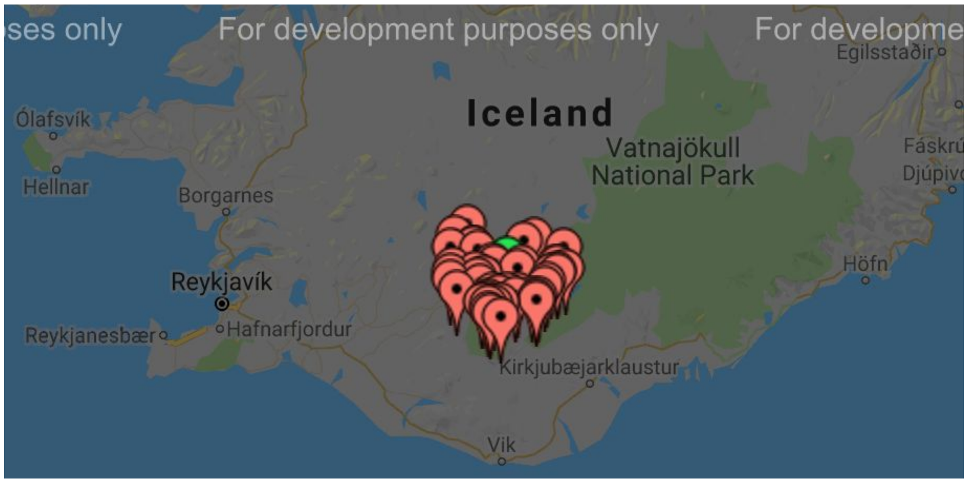
==

Picture Analysis

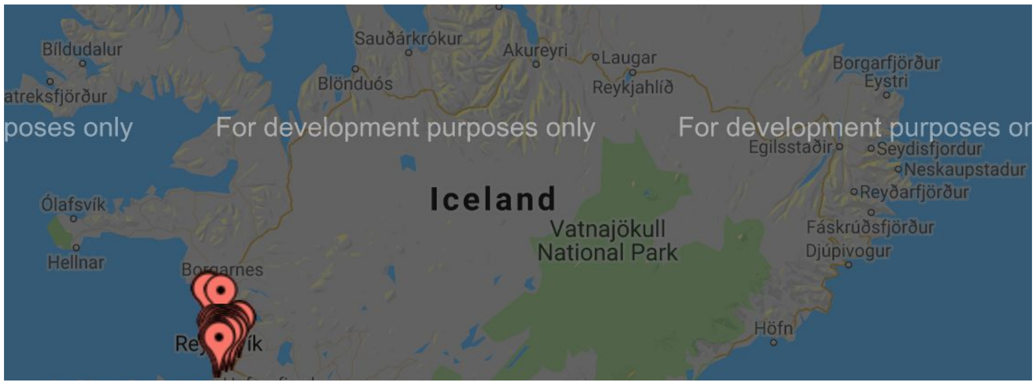
==

(15 points) At least three pictures on different queries on the OSM file of your preference.

Iceland - “e”, 20 miles from 64.15 -18.95



Iceland - “food”, 20 miles from 64.15 -21.95



Iceland - “inn”, 60 miles from 65.14 -20.992

Ísafjörður
Bíldudalur
Patreksfjörður
Ólafsvík
Hellnar
Borgarnes
Reykjavík
Reykjanesbær
Hafnarfjordur
Sauðárkrókur
Blönduós
Dalvík
Akureyri
Húsavík
Laugar
Reykjahlíð
Iceland
Vatnajökull National Park
Borgarfjörður Eystri
Egilsstaðir
Seydisfjordur
Neskaupstadur
Reyðarfjörður
Fáskrúðsfjörður
Djúpivogur
Höfn
Kirkjubæjarklaustur
Vík
For development purposes only

```
==================================================================
```

Source Code

```
==================================================================
```

KDTree.h

```
==================================================================
#ifndef __kdtree__
#define __kdtree__
#include <string>

class KDNode {
    private:
        double latitude;
        double longitude;
        std::string description;
        KDNode *left;
        KDNode *right;
// perhaps you want to include `depth` as well
        unsigned int depth;
// this function returns the distance in miles from lat, lon
// to this object
        double distance(double lat, double lon);

    public:
```

```
//constructor has added depth parameter, for making traversal
//more efficient
        KDNode(double lat, double lon, const char *desc,
unsigned int dep);
        ~KDNode();

    friend class KDTree;
};

class KDTree {
    private:
        unsigned int size;
        KDNode *root;

        void destroy(KDNode *p);
//print is a method used for debugging
        void print(KDNode *p);
//helpPrint is the recursive function that finds the points
//within the map, which is called in printNeighbors
        unsigned int helpPrint(double lat, double lon, double
rad, const char *filter, KDNode *q);
    public:
        KDTree();
        ~KDTree();

        void print();

        unsigned int getSize();
        void insert(double lat, double lon, const char *desc);

        unsigned int printNeighbors(double lat, double lon,
double rad, const char *filter);
};

#endif
```

KDTree.cc

==

```
#include "KDTree.h"
#include <math.h>
#include <iostream>

//sets the starting values of the tree and the lat/lon
KDNode::KDNode(double lat, double lon, const char *desc,
unsigned int dep) {
    left = NULL;
    right = NULL;
    depth = dep;
    description = desc;
    latitude = lat;
    longitude = lon;
}


KDNode::~KDNode() {
}

// returns the distance (in miles) between this node and a given
// lat and lon
double KDNode::distance(double lat, double lon) {
    double param = M_PI / 180.0;
// required for conversion from degrees to radians
    double rad = 3956.0;  // radius of earth in miles
    double d_lat = (lat - latitude) * param;
    double d_lon = (lon - longitude) * param;
    double dist = sin(d_lat/2) * sin(d_lat/2) +
cos(latitude*param) * cos(lat*param) * sin(d_lon/2) *
sin(d_lon/2);
    dist = 2.0 * atan2(sqrt(dist), sqrt(1-dist));
    return rad * dist;
}

//the tree constructor for a KD tree
KDTree::KDTree() {
    root = NULL;
    size = 0;
}
//KD tree destructor that deletes all nodes from the tree
KDTree::~KDTree() {
    destroy(root);
}

void KDTree::destroy(KDNode *p) {
    // "Destroy" should work just like the destroy method of any
//tree: delete all children, then move to parent.
    if(p) {
        destroy(p->left);
        destroy(p->right);
        delete p;
    }
}

void KDTree::print(KDNode *p) {
// recursive - prints all nodes  in the tree. Works like the
//destroy function by starting at the leaves and works its way
//up to the root
    if(p) {
        print(p->left);
        print(p->right);
        std::cout << "\t[\"" << "CENTER" << "\", " <<
p->latitude << ", " << p->longitude << "],\n";
        std::cout << "\t[\"" << p->description << "\", " <<
p->latitude << ", " << p->longitude << "],\n";
    }
```

```
}


void KDTree::insert(double lat, double lon, const char *desc) {
//The insertion method for a KD tree. Checks the value of the x
//compared to the parent, then to the y if the x’s are equal
//reference is a boolean that determines which dimension should
//be checked at each level - toggles with each loop
//d keeps track of the depth of the node for the constructor to
//use after the loop
    KDNode *parent = NULL;
    bool reference = true;
    unsigned int d = 0;
    if (root == NULL){
        root = new KDNode(lat, lon, desc, d);
        size++;
        return;
    }
    KDNode *q = root;
    while(q){
        d++;
        if(reference){
            reference = false;
            if(lon < q->longitude){
                parent = q;
                q = q->left;
            }else{
                parent = q;
                q = q->right;
            }
        }else{
            reference = true;
            if(lat < q->latitude){
                parent = q;
                q = q->left;
            }else{
                parent = q;
                q = q->right;
            }
        }
    }
    if(reference){
        if(lat < parent->latitude){
            parent->left = new KDNode(lat, lon, desc, d);
            size++;
        }else{
            parent->right = new KDNode(lat, lon, desc, d);
            size++;
        }
    }else{
        if(lon < parent->longitude){
            parent->left = new KDNode(lat, lon, desc, d);
            size++;
        }else{
            parent->right = new KDNode(lat, lon, desc, d);
            size++;
        }
    }
}

unsigned int KDTree::printNeighbors(double lat, double lon,
double rad, const char *filter) {
//printNeighbors only sets up the markers to be printed. The
//call to helpPrint actually adds the markers to be printed
    std::cout << "var markers = [\n";
    std::cout << "\t[\"" << "CENTER" << "\", " << lat << ", " <<
lon << "],\n";
    int nombre = helpPrint(lat, lon, rad, filter, root);
    std::cout << "];\n";
    return nombre;
}

unsigned int KDTree::helpPrint(double lat, double lon, double
rad, const char *filter, KDNode *q){
//a recursive helper function to help print the tree.
//the “else” part is useful for eliminating subtrees where
//either the latitude or the longitude is too extreme.
```

```
//This is where the depth parameter is used, in order to decide
//to compare latitude or longitude.
    if(!q){
        return 0;
    }
    if(q->distance(lat, lon) < rad ){
        if(q->description.find(filter) != std::string::npos){
            std::cout << "\t[\"" << q->description << "\", " <<
q->latitude << ", " << q->longitude << "],\n";
            return helpPrint(lat, lon, rad, filter, q->left) +
helpPrint(lat, lon, rad, filter, q->right) + 1;
        }else{
            return helpPrint(lat, lon, rad, filter, q->right) +
helpPrint(lat, lon, rad, filter, q->left);
        }

    }else{
        if(q->distance(lat,q->longitude) > rad && q->depth % 2
!= 0) {
            if(q->latitude > lat){
                return helpPrint(lat, lon, rad, filter,
q->left);
            }else{
                return helpPrint(lat, lon, rad, filter,
q->right);
            }
        }
        else if (q->distance(q->latitude,lon) > rad && q->depth
% 2 == 0){
            if(q->longitude > lon){
                return helpPrint(lat, lon, rad, filter,
q->left);
            }else{
                return helpPrint(lat, lon, rad, filter,
q->right);
            }
        }else{
            return helpPrint(lat, lon, rad, filter, q->right) +
helpPrint(lat, lon, rad, filter, q->left);
```

```
        }
    }
}
//prints the tree starting at the root
void KDTree::print(){
    std::cout << "var markers = [\n";
    print(root);
    std::cout << "];\n";
}
//returns the size of the tree
unsigned int KDTree::getSize() {
    return size;
}
```