Implementation:

Given the nature of k-d trees we decided the best approach would be to first define the goals of our algorithm.  After conducting our initial research on the topic, we ultimately chose to implement a nearest neighbor algorithm. We then further narrowed in on a spatial search that would search through location data to return the nearest member of the given data to a given set of coordinates.

Our primary objective was to develop a system capable of efficiently finding the nearest neighbor in a spatial dataset. We found several examples of location data containing points of interest (e.g., schools, national parks) and their location coordinates. Our aim was to develop an algorithm that allows the user to input a specific location and have the code return the nearest point based on the provided dataset.

The first step towards this goal was to create a main file (‘main.cpp’) that would be capable of reading the user’s input from command line arguments. The main file takes in two arguments, the first is the dataset to search over and the second is a file containing user commands.

Once the input and goals were defined, we created a header file (‘kdtree.h’) that would contain the classes and structures needed to implement our design. To this goal we created two structures, ‘Point’ to store a coordinate pair, and ‘NationalPark’ to store a name and a ‘Point’. The primary focus of our implementation is the ‘KDTree’ class, which stores all the given data into a kd-tree structure.

The ‘buildTree’ function works using recursion to select the median point along the axes, alternating from latitude to longitude for each new level of the tree. This function ensures that the tree is balanced. The ‘destroy’ function recursively destroys a selected node. The function begins by destroying the left and right subtrees before deleting the current node. The ‘findNearest’ function works using a helper function called ‘findNearestHelper’. The ‘findNearest’ function sets up a variable called ‘best’ to store the closest neighbor and ‘bestDist’ to store the distance of this point. The helper function ‘findNearestHelper’ recursively searches for the nearest neighbor of the user inputted point. It calculates the distance from the current node's park to the target point and updates the best distance and best park if the current distance is smaller. It then determines which subtree to search first based on the target's coordinates and the current splitting axis. It searches the near subtree first and only searches the far subtree if the distance to the splitting plane is less than the best distance found so far. The ‘distance’ function plays a big role in the ‘findNearestHelper’ function. It calculates the distance between points by calculating the difference in latitudes and longitudes, squaring these differences, and then returns the square root of their sum. The ‘insert’ method works in a similar manner to the ‘findNearest’ method as it utilizes a helper function that performs the recursive search along the tree to determine the correct location for the new data.

There were several setbacks we encountered while developing this code. One issue we repeatedly came across was memory allocation. We originally had delete commands in the destructor method of the ‘KDTNode’ class which was causing segmentation faults. By removing these commands, we were able to correctly handle dynamic memory allocation. Another challenge we encountered was creating the logic for the ‘findNearestHelper’. To switch between latitude and longitude during the search, we came up with the method of using the modulo operator on ‘depth’ since it will always return a binary value, in our case a 0 being latitude and a 1 referring to longitude. We also wanted to use an efficient code so we created variables ‘diff’, ‘near’, and ‘far’ to perform the depth-first search on the mostly like subtree first and only search the other subtree if the distance from the splitting plain was less than the current value stored in ‘bestDist’.