22nd March, 2015

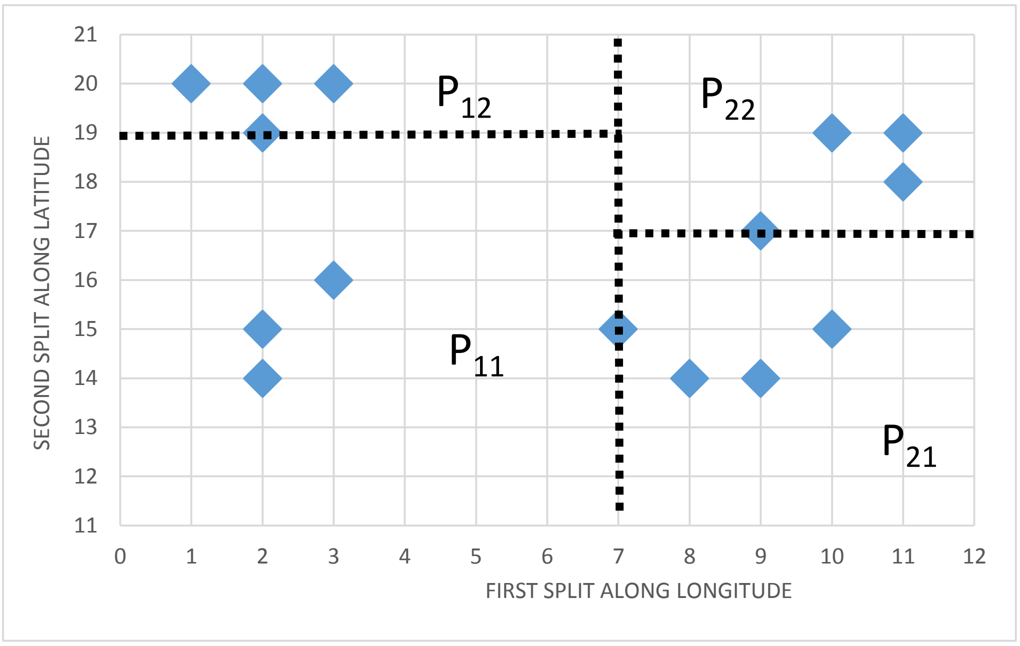
Birla Institute of Technology & Science, Pilani
Data Structures & Algorithms (CS F211)
Online Test # 1

Max. Marks: 30 **Time: 150 min**

Problem:

BITS Pilani wants to maintain Geographical Information Systems across all our campuses so that it can quickly find out the objects which are in physical vicinity of any point in the campus. This will enable the institute to identify whether a given location is free, and if there are other objects within the vicinity of the location, to list them fast.

For ensuring that we can identify the region containing the point, it was decided to use the following way of visualizing points:



The location point of each object is given. Points in campus are first partitioned along median of longitude (7). Then points on left and right of median are partitioned separately using their own median of latitude (19 in left and 17 in right). All objects in the campus gets partitioned into 4 regions P_{11}, P_{12}, P_{21} and P_{22}, as shown in the above figure.

In order to design a data structure which can make such queries execute faster than $O(n^2)$ (calculating distance of all n points with each other, whenever a query is launched), following strategy has been thought of:

- Represent each point on Map as an array of values of its dimensions. For example – Clock Tower might be stored as an array of [longitude, latitude, sea level, pointId].
- Represent all points on campus i.e. whole dataset, as an array of points.
- Partition the dataset into two set of equal number of points on the basis of value of a pivot point in a chosen dimension. For example – If we consider partitioning on the basis of longitudes; one partition should contain half of total number of points whose longitude is

smaller than longitude of the pivot point and the other partition should contain other half of the points whose longitude is larger than or equal to longitude of pivot point.
- Pick the next dimension and split both the partitions again.
- Return an array storing starting index of each partition.

This way, at the end of this partitioning, all points which are in close vicinity of each other will belong to the same partition.

For finding points within ***vicinity*** (+/- 2 range) of a given coordinate, say at (longitude, latitude, s), within a partition; it is now sufficient to search within partition containing (longitude, latitude, s) and in boundary cases, an additional area of longitude longitude-2 to longitude+2 and latitude latitude-2 to latitude+2.

To further speed up the lookup, following strategy has been thought of:

- Each point in the partition containing (longitude, latitude, s) should be placed on a hash table, with longitude as the key and chaining as the collision resolution technique.
- To search for longitudes starting from longitude-2 to longitude+2, it is now sufficient to perform hash table lookups.
- To identify points with latitudes within range latitude-2 to latitude+2, it would be sufficient to lookup within the chain corresponding to each longitude.
- Points within (+/- 2) vicinity of given point can be printed to file.

Data Structures used: (available in `data.h`)

1. `PointList`: list of points stored in an array, with each row having <longitude, latitude, sealevel, pointID>
2. `hData`: a structure for use in the hashtable to maintain separate chaining, mentioned below. It contains the fields: `longi`, `index`, `next`. Here, `index` is index of a point in pointList.
3. `LongitudeHash`: This hashtable stores longitude information using separate chaining for collision resolution. For chaining, the structure `hData` must be used. The size of hashtable is 41. The hash function to be used: h(x)=x mod 41

Partially implemented code is already provided to you. All functions provided are already present in **`driver.c`**.

The driver takes the following as command-line parameter: longitude, latitude and sea level of a point. The code for reading the data from file, making calls to the partitioning, making lookup for determining vicinity, and printing the partition results to files are already available.

Functions to implement (in the files mentioned)

Complete the given code with following functions for providing vicinity lookup on the geographical data. Please note that all coordinates are integer.

1. `int* partitionInto4(PointList plist, int psize) (to be written in partitionInto4.c)`

   This function partitions the given point list **in-place (**i.e. no extra memory should be allocated for storing the partitions) into four partitions in the order -> P_{11}, P_{12}, P_{21} and P_{22}. It takes pointList and the size of the list as input parameters; and returns an integer array of size 4 containing the starting indices of the four resultant partitions.

This function uses "Quick Select Algorithm" (code to be written in `findMid()` function) to find a pivot point which splits the data into two set of equal number of points, depending upon either longitude or latitude. This pivot point should be placed at the beginning second half of partition (where all value are equal or greater than pivot's value). Longitude should be used for first split and then latitude should be used for remaining splits.

2. `int findMid(PointList plist, int start, int end, int col) (to be written in findMid.c)`

   This function performs quick select algorithm on either longitude (column 0) or latitude (column 1) of point list `plist` . It partitions the list starting from `start` till `end` into two equal halves based on the column type. It returns the starting index of the latter partition after splitting into two. [Hint: you must use a *swap* function to swap contents of rows for analysis.]

3. `populateHash(LongitudeHash H, PointList plist, int start, int end) (to be written in populateHash.c)`
   This function reads content from `plist` from the index `start` to the index `end` and inserts the entry to the hashtable `H`.

4. `printVicinty(LongitudeHash H, int longi, int lati, int sealvl) (to be written in printVicinity.c)`

   This function takes the coordinates of a point in map given by `longi, lati, sealvl.` It then searches in the hash table for all points within +/- 2 latitude and +/- 2 longitude of this point; prints lontitude, latitude, sealevel and pointId of such points to the file named `output3.txt`.

Tasks to perform:

- Write the function `partitionInto4` in `partitionInto4.c`
- Write the function `findMid` in `findMid.c`
- Write the function `populateHash` in `populateHash.c`
- Write the function `printVicinity` in `printVicinity.c`
- Compile and execute the code. Write each of the 4 partitions into a separate file, named as "partition11.txt", "partition12.txt", "partition21.txt" and "partition22.txt". Also generate output3.txt.
- Profile the code and report following things in separate output fiels (to be created manually):
  - Total number of swaps required → output1.txt
  - Cumulative-time of function `partitionInto4()` → output2.txt

Support files: `driver.c, data.h, inputData.txt`

Files to be delivered:

`partitionInto4`.c, `findMid`.c, `populateHash.c,` `printVicinity`.c, output1.txt, output2.txt, output3.txt