MAST20003 – Algorithms and Data Structures

Assignment 2 – 994765 – Dion Tran

INTRODUCTION: SUMMARY OF DATA STRUCTURES AND INPUTS.

In this report, I will be exploring the effectiveness of my 2-d binary tree algorithm in inserting and searching with various types of arranged data and explaining the thought process that came to building the algorithm.

Data structures are used to collect and organize data in ways to insert, sort or search in an effective manner. Depending on the data and what the person wants to do with such data, many different data structures were created each with its boons and banes. It starts down with primitive datatypes, i.e. chars, integers, floats…, which are used to categories words or numbers. For example, a person's name "Simon" would be of string data type and his age, 21, can be of integer data type. We can then organize a person's name and age as a record and then collect and store multiple records into a manmade complex abstract data structure. Some example of Abstract data structures are Linked List, Tree, Graph, Stacks and Queues. Linked lists are made up of nodes which hold data and have a pointer that points to the next node making it dynamic which has an advantage over arrays, which are static and can only take a certain amount of data that's set prior.

However linked lists can only traverse in one direction and thus when searching through the linked list for a specific node, if the specific node is at the end, we may have to traverse the entire linked list thus giving it a time complexity of O(n). Another type of dynamic array is the binary tree, and unlike the Linked Lists which are linear data structures, binary trees are hierarchical data structures.

The head node of a binary tree is called the root of the tree. This head node can point to two children node, left and right. In a normal binary tree, the left child would indicate a node with data smaller then the head nodes with the right child having its data be larger. However in this report, we will be looking at a different type of binary tree, a 2-D binary tree which usually takes in x and y data points. Data is organized in where every other layer is split by either the primary key, x point or y point. Figure 1 shows the head's children being split apart by the head node's x point, and its children's children being split by y point.

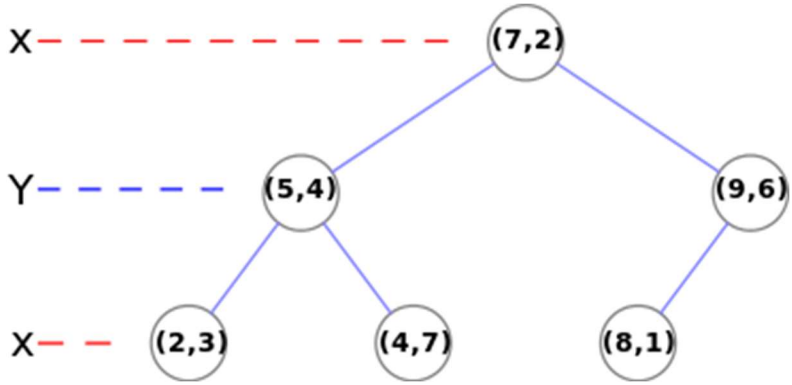

Figure 1. Example 2D Binary Tree

The project was to create an algorithm capable of taking in 2-d location data points and add them into a binary tree. This project has two stages. The first stage was to search for the nearest data point of a given location and the second was to find all data points inside a certain radius of a given location. The data given has been arranged in 3 ways, random, median and sorted by x. We will repot on the number of key comparisons used by the code and compare the results between each stage and each version of the data.

Insert function, “node_t *insertLeaf(node_t *leaf, node_t *par, int count)” in tree1.c and tree2.c, will take in a new node from the data and through recursion will find a suitable place and append itself to it. It will split itself by x or by y by keeping track of the amount of times its recurred, this is stored in int count, and if the count is even, it will split by x, if odd, it will split by y. As the tree is split by x or y and not x and y, there will be times when we will have to search both branches. These moments occur when the difference between the primary key is less than the distance between the node and location. I couldn’t find a way to continuously traverse both branches, and used an alternative where the function will return the node, left or right, which was closest to the location instead.

Stage 1 search function, “node_t *nearestNode(double xinp, double yinp, node_t *par, double sD, int count)” in tree1.c, is similar to the insert function in that it traverses the tree, keeps track of count and is also recursive. Instead it will return the node that is nearest by keeping track of the shortest distance between the nodes, and if the next node is not closer to the shortest distance, the tree will stop and return the node.

Stage 2 search function, “void printNearestNode(double xinp, double yinp, double rad, node_t *par, double sD, int count, FILE *file)” in tree2.c, is a void function that traverses the binary tree and prints out nodes that are in the radius of the location.

Theoretically, a 2D binary tree’s worst case time complexity is O(n) as there is a possibility for the binary tree to imitate a linked list, i.e all the nodes are all pointing to the right or left.

It is expected that the binary tree created the data that is sorted by x, from smallest to largest, will give the highest average time complexity as at least half of the data points should point to the right and as such there is a higher likely chance of increased average key comparisons.

STAGE 1:

- Data (number of key comparisons)

15 key comparisons tested with the 3 data versions.

map1 key comparisons	random	median	sort by x
144.9576 -37.82756	14	12	6
144.9566201 -37.79455742	4	7	6
144.9519423 -37.78462231	4	12	6
144.9733972 -37.8299839	8	12	6
144.9753504 -37.7932694	8	12	6
144.9012645 -37.82998456	10	12	1
144.9013768 -37.83106466	9	12	2
144.9016815 -37.82937965	10	12	3
144.9017936 -37.82997653	10	12	3
144.9900224 -37.81379913	8	12	4
144.990067 -37.81280632	11	12	6
144.9903968 -37.81303809	11	12	6
144.9904547 -37.81058384	11	12	6
144.9904547 -37.81058384	11	12	6
144.9905519 -37.81181089	11	12	6
average	9.3333333	11.66666667	4.8666667

Table 1. Map1 number of key comparisons.

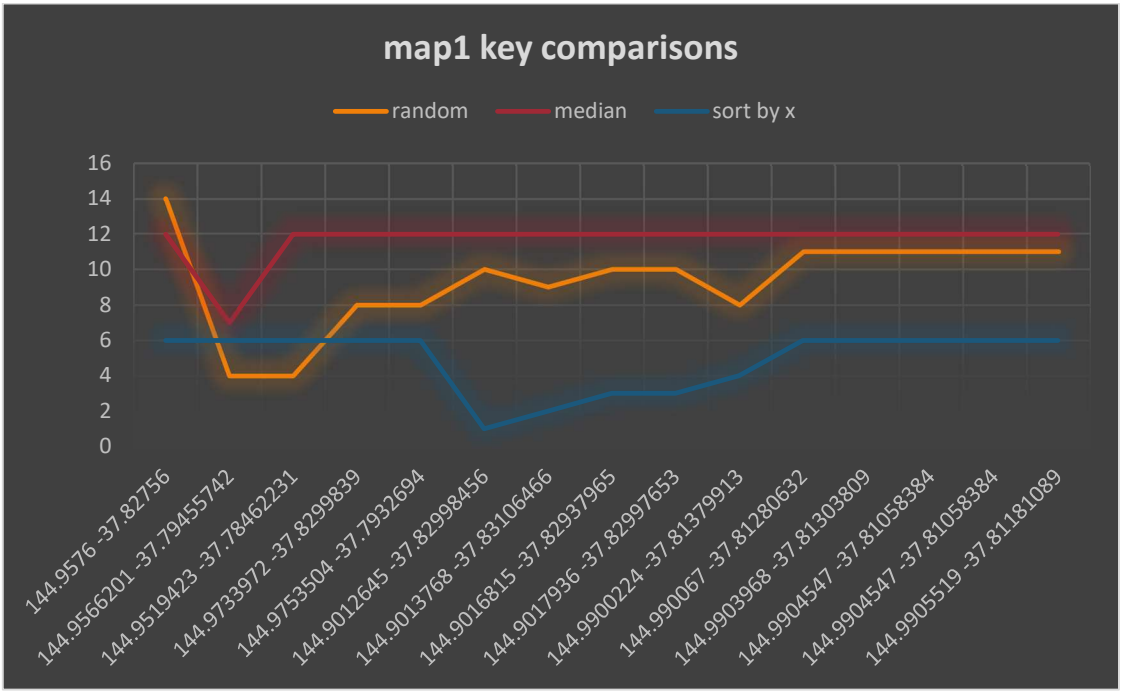

Figure 2. Line graph of map1 key comparisons.

Table 1. Shows median having the most average comparisons at 11.667, it seems like it is consistently at 12 with random having an average comparison of 9.33 and the shortest amount of key comparisons is the sort by x with an average of 4.8667. Figure 2. Shows median and sort by x having consistency in key comparisons whereas median is quite sporadic.

- Comparison with theory

This goes directly against my theory. The test data shows the complete contradictory in where it was predicted that sort by x will have higher key comparisons however it was the lowest. Although there is a possibility for the code to have errors.

STAGE 2:

- Data (number of key comparisons)

map2 -no radius key comparisons	random	median	sort by x
144.9576 -37.82756 0	0	0	0
144.9566201 -37.79455742 0	0	7	0
144.9519423 -37.78462231 0	4	0	0
144.9733972 -37.8299839 0	0	0	6
144.9753504 -37.7932694 0	0	12	0

map2 -0.005 radius key comparisons	random	median	sort by x
144.9576 -37.82756 0.005	0	7|8	0
144.9566201 -37.79455742 0.005	0	0	0
144.9519423 -37.78462231 0.005	4	0	0
144.9733972 -37.8299839 0.005	0	7|8|9|10|11|12	6
144.9753504 -37.7932694 0.005	3|7|8	0	0

map2 -0.01 radius key comparisons	random	median	sort by x
144.9576 -37.82756 0.01	2|3	1	0
144.9566201 -37.79455742 0.01	3	5|6|7|8|9	0
144.9519423 -37.78462231 0.01	3|4	0	0
144.9733972 -37.8299839 0.01	0	0	5|6
144.9753504 -37.7932694 0.01	3|7|8	7|8|9|10|11|12	0

Tables with multiple numbers in data points have key comparisons of different nodes separated by a "|"

Smaller radius data points key points are included in the larger data points. Sort by x has one 6 across all 3 radius. These 5 points are the same as the first 5 points in stage 1 testing.

- Comparison with stage 1.

Both stage 1 and stage 2 show the same trend with raw data, for example, a binary search tree made using data sorted by x in both stages have it taking less key comparisons to reach optimal node, with both numbers showing around 6. On the other hand, data sorted by median showed that it took at utmost double the amount of key comparisons sorting by x had.

- Comparison with theory.

This goes against my theory, though there is not enough data, median shows more higher numbers than sort by x and random. Both sort by x and random are similar in average key comparisons.