RAMAIAH UNIVERSITY OF APPLIED SCIENCES

ASSIGNMENT

Course Code	CSC202A
Course Name	Data Structure and Algorithms
Programme	B.Tech
Department	CSE
Faculty	FET

Name of the Student	Satyajit Ghana
Reg. No	17ETCS002159
Semester/Year	03/2018
Course Leader/s	Dr. Pushphavathi T P

Declaration Sheet			
Student Name	Satyajit Ghana		
Reg. No	17ETCS002159		
Programme	B.Tech	Semester/Year	03/2018
Course Code	CSC202A		
Course Title	Data Structures and Algorithms		
Course Date		to	
Course Leader	Dr. Pushphavathi T P		

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Question No. 1

Solution to Question No. 1 Part A:

A 1.1 Introduction to data structures to represent graphs:

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as, A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.

Planar Graph. A graph where all the intersections of two edges are a vertex. Since this graph is located within a plane, its topology is two-dimensional. This is typically the case for power grids, road and railway networks, although great care must be inferred to the definition of nodes (terminals, warehouses, cities).

Non-planar Graph. A graph where there are no vertices at the intersection of at least two edges. Networks that can be considered in a planar fashion, such as roads, can be represented as non-planar networks. This implies a third dimension in the topology of the graph since there is the possibility of having a movement "passing over" another movement such as for air and maritime transport, or an overpass for a road. A non-planar graph has potentially much more links than a planar graph.

Simple graph. A graph that includes only one type of link between its nodes. A road or rail network are simple graphs.

Multigraph. A graph that includes several types of links between its nodes. Some nodes can be connected to one link type while others can be connected to more than one that are running in parallel. A graph depicting a road and a rail network with different links between nodes serviced by either or both modes is a multigraph. **(Dr Jean Paul, Dr Cesar)**

A 1.2 Importance and relevance of graph theoretical concepts and algorithms to the transportation problem:

In solving problems in transportation networks Graph theory in mathematics is a fundamental tool. The term graph in mathematics has two different meaning. One is the graph of a function or the graph of a relation. The second usually related to 'graph theory' is a collection of 'vertices' or 'nodal' and 'links' or 'edges' for purpose of this paper we are concerned with the latter type graph theory has been closely tied to its applications and its use first can be credited to transport ant followed by its application to other fields. In transportation graph theory is most commonly used to study problems.

One way street problem: Robin's Theorem, the first problem we consider has to do with movement of traffic. If traffic where to move more rapidly and with fewer delays in our cities, this would alleviate wasted energy and air pollution. It has sometimes been argued that making certain streets one-way would move traffic more efficiently. We consider the one-way and, if so, how to do it. Of course, it is always possible to make certain streets in a city one-way simply put up a one-way street sign. What is desired is to do in such a way that it is still possible to get from any place to any other place.

A 1.3 Real life examples of transportation problem using a graph data structure:

Graphs are used to model situation in which a commodity is transported from one location to another. A common example is the water supply, where the pipelines are edge, vertices represent water users, pipe

joins and so on. Highway systems can be thought of as transporting cars. In many examples it is natural to interpret some or all edges as directed. A common feature of transportation system is the existence of a capacity associated with each edge, the maximum number of cars that can use a road in an hour. The maximum amount of water that can pass through a pipe and so on.

1. Maritime traffic

Let u_i , $i = 1, 2.., m$ and v_j $j = 1, 2.., n$ are different seaports and some products are ready for shipment at u_i to v_j. Let s_i be the quantity available at a_i and d_j the quantity demanded at v_j. How should the products be shipped? Here also, network serves as a model. That is u_i, $i = 1, 2.., m$ and v_j $j = 1, 2.., n$ are treated as nodes and shipping routes can be represented by arcs of the form (u_i, v_j) with a capacity equal to the shipping capacity between the two seaports. Two new nose s and t are introduced as a source and sink, respectively such that join s to each u_i by an arc with capacity $c(s, u_i) = s_i$ and join each node v_i to t by an arc with capacity $c(v_j, t) = d_j$. A maximum flow for this transportation network yields the quantity of products to ship along each route in order to satisfy all demands, if this is possible.

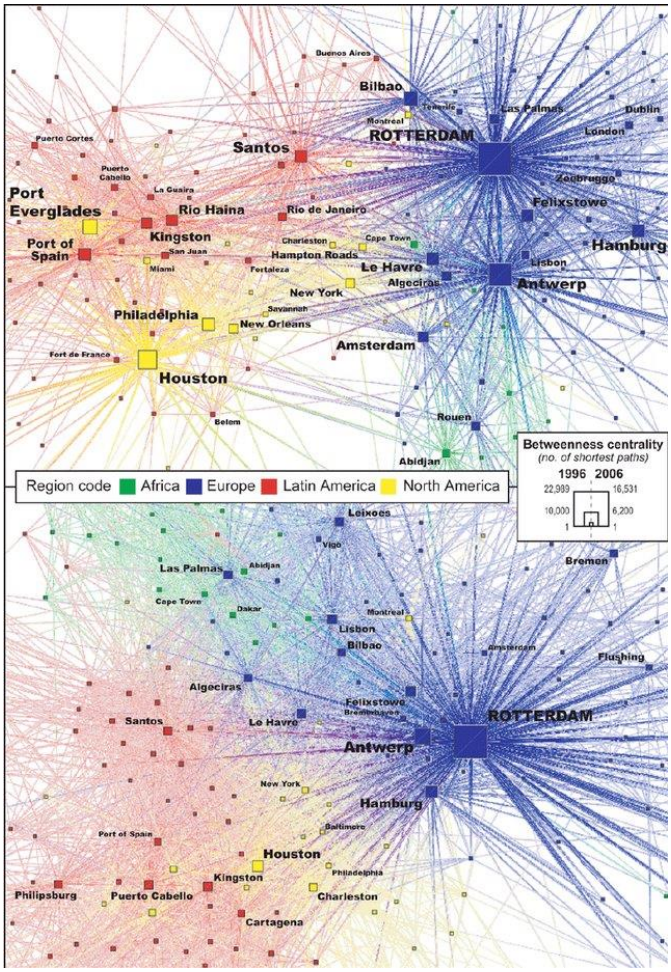
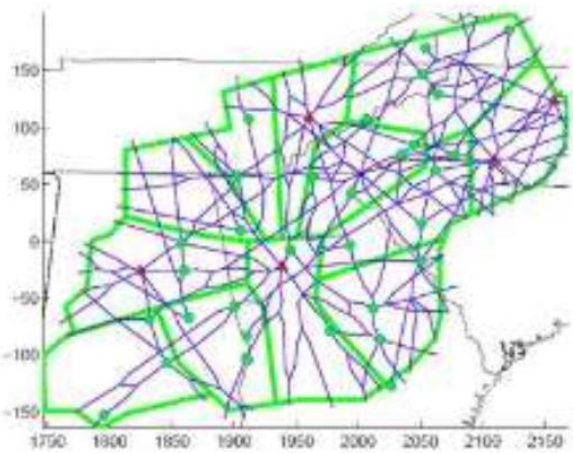


Figure A1.1 Graph visualization of the Atlantic liner shipping network, 1996-2006

2. Air Traffic Control Network

Air traffic control is an essential element of the communication structure which supports air transportation. Two basic for air traffic control (ATC) are safely and efficiency of air traffic movement. ATC organizes the air space to achieve the objective of a safe, expeditious and orderly flow of air traffic. The increasing range of aircraft technology means more attention to the allotment of air space. The problem is future compounded by the fact that busy airports sustain excessive lending and departure rates and airports themselves are invariably situated within busy terminal areas and in close proximity to other airports. Future more, these airports are often sited near the junction of air routers serving other destinations. **(Sanjay kumar Bisen)**



Question No. 2

Solution to Question No. 1 Part B:

B 1.1 Identification of an application that requires sorting:

Introduction:

In computer science, a sorting algorithm is an algorithm that puts numbers or elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for suitable data and for producing human readable output. More formally, the output must satisfy two constraints: first constraint is that output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order). Second constraint is that output is a permutation, or reordering, of the input. Since the dawn of computing, the sorting problem has a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement.

Applications:

- Searching – Binary search tests whether an item is in a dictionary in $O(\log n)$ time, provided the keys are all sorted. Search preprocessing is perhaps the single most important application of sorting.

- Closest pair – Given a set of n numbers, how do you find the pair of numbers that have the smallest difference between them? Once the numbers are sorted, the closest pair of numbers must lie next to each other somewhere in sorted order. Thus, a linear-time scan through them completes the job, for a total of $O(n \log n)$ time including the sorting.

- Element uniqueness – Are there any duplicates in a given set of n items? This is a special case of the closest-pair problem above, where we ask if there is a pair separated by a gap of zero. The most efficient algorithm sorts the numbers and then does a linear scan though checking all adjacent pairs.

- Selection – What is the kth largest item in an array? If the keys are placed in sorted order, the kth largest can be found in constant time by simply looking at the kth position of the array. In particular, the median element appears in the $\frac{n^{nd}}{2}$ position in sorted order.

- Convex hulls – What is the polygon of smallest area that contains a given set of n points in two dimensions? The convex hull is like a rubber band stretched over the points in the plane and then

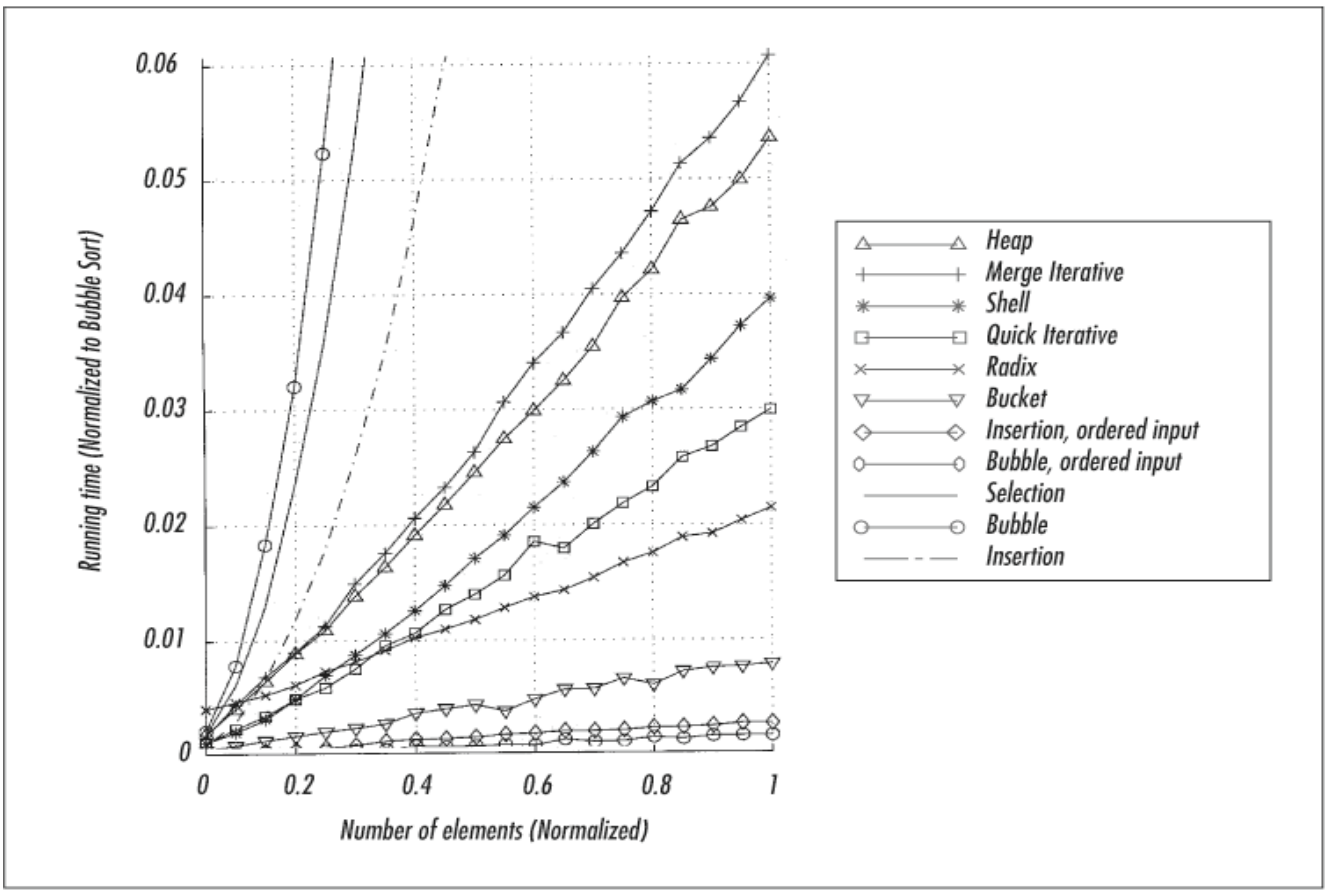
released. It compresses to just cover the points. The convex hull gives a nice representation of the shape of the points and is an important building block for more sophisticated geometric algorithms,

(The Algorithm Design Manual, Steven S. Skiena)

B 1.2 Evaluation of the available sorting algorithms and selection of the most appropriate one for your application:

Table B1.1 Comparison of Sorting Algorithms

Name	Best	Average	Worst	Memory	Stable
Quick Sort	$n \log n$	$n \log n$	n^2	$\log n$	No
Merge Sort	$n \log n$	$n \log n$	$n \log n$	n	Yes
Heap Sort	n	$n \log n$	$n \log n$	1	No
Insertion Sort	$n \log n$	$n \log n$	n^2	$\log n$	Yes
Selection Sort	n^2	n^2	n^2	1	Yes
Bubble Sort	n	n^2	n^2	1	Yes



- Quick sort: When you don't need a stable sort and average case performance matters more than worst case performance. A quick sort is $O(N\ log\ N)$ on average, $O(N^2)$ in the worst case. A good implementation uses $O(log\ N)$ auxiliary storage in the form of stack space for recursion.

- Merge sort: When you need a stable, $O(N\ log\ N)$ sort, this is about your only option. The only downsides to it are that it uses $O(N)$ auxiliary space and has a slightly larger constant than a quick sort.

- Heap sort: When you don't need a stable sort and you care more about worst case performance than average case performance. It's guaranteed to be $O(N\ log\ N)$, and uses $O(1)$ auxiliary space, meaning that you won't unexpectedly run out of heap or stack space on very large inputs.

- **Insertion sort**: When N is guaranteed to be small, including as the base case of a quick sort or merge sort. While this is $O(N^2)$, it has a very small constant and is a stable sort. **It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.**

- Bubble sort, selection sort: When you're doing something quick and dirty and for some reason you can't just use the standard library's sorting algorithm. The only advantage these have over insertion sort is being slightly easier to implement.

Advantages:

Name of Sort	Advantages	Disadvantages
Quick Sort	$\Theta(N \log N)$	Unstable, sensitive, $\Omega(N^2)$
Merge Sort	$\Theta((N \log N)$, insensitive	$O(N)$ temporary workspace
Heap Sort	$\Theta((N \log N)$, insensitive	unstable
Insertion Sort	$\Theta(N)$ for nearly sorted	$\Omega(N^2)$ otherwise
Selection Sort	Stable, Insensitive	$O(n^2)$
Bubble Sort	$\Theta(N)$ for nearly sorted	$\Omega(N^2)$ otherwise

Since our application's data is fed on real time, i.e. the database is population from beginning, the database is hence almost sorted. The data is almost always sorted and needs a few swaps to sort it. A stable sorting algorithm with preferably less memory consumption is required.

Insertion sort is a good candidate for such kind of data, which has low memory consumption of $\log n$ and is a stable sort, moreover it works well with almost sorted data. The time complexity for almost sorted data is $\Theta(N)$ and is a stable sort.

B 1.3 Implementation of C program and testing of the same:

```
void insertion_sort(void** arr, size_t length, size_t ele_size, int (*comparator)(const
void* data1, const void* data2)) {
    void* key;
    for (int i = 1 ; i < length ; i++) {
        key = arr[i];
        int j = i - 1;

        while (j >= 0 && comparator(arr[j], key) >= 0) {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = key;
    }
}
```

```
/* Sequential Data Methods */
void sort_seq_data(Database* mydatabase) {
    insertion_sort(mydatabase -> seq_data -> data, (size_t)mydatabase -> seq_data ->
length, sizeof *(mydatabase -> seq_data -> data), mydatabase -> seqComparator);
}
```

<u>See Appendix A – Vectors, for the complete implementation of the above.</u>

<u>Testing:</u>

```
D:\Assignment-SEM03-02-2018\CSC202A\InsertionSort\cmake-build-debug\InsertionSort.exe

-------------CITY DATABASE----------------------------
1.     View Database
2.     Sort Database
3.     Exit
Your Choice : 1
          ID            CITY NAME   COORDINATES      ADDRESSES
                                       X      Y
--------------------------------------------------------------------------------
   846930886              Chennai     13     80       Kovalam,  Mahabalipuram
  1681692777                Delhi     29     77       Rajpath Marg,  South Delhi
  1714636915          Bhubaneshwar    20     86       Khandagiri
  1804289383             Bangalore    13     78       Koramangala,  Kalyan Nagar
  1957747793                Bhopal    23     77       Van Vihar,

-------------CITY DATABASE----------------------------
1.     View Database
2.     Sort Database
3.     Exit
Your Choice : 2

-------------Sort DATABASE---------------------------
1.     Sort by City Name
2.     Sort by ID
3.     Back
```

```
Your Choice : 1

Sorting by City Name
SORTED !

-------------CITY DATABASE----------------------------
1.     View Database
2.     Sort Database
3.     Exit
Your Choice : 1
          ID            CITY NAME   COORDINATES      ADDRESSES
                                       X      Y
-------------------------------------------------------------------------------
  1804289383            Bangalore     13     78       Koramangala,  Kalyan Nagar
  1957747793               Bhopal     23     77       Van Vihar,
  1714636915         Bhubaneshwar     20     86       Khandagiri
   846930886              Chennai     13     80       Kovalam,  Mahabalipuram
  1681692777                Delhi     29     77       Rajpath Marg,  South Delhi

-------------CITY DATABASE----------------------------
1.     View Database
2.     Sort Database
3.     Exit
Your Choice : 2

-------------Sort DATABASE--------------------------
1.     Sort by City Name
2.     Sort by ID
3.     Back
Your Choice : 2

Sorting by City ID
SORTED !

-------------CITY DATABASE----------------------------
1.     View Database
2.     Sort Database
3.     Exit
Your Choice : 1
          ID            CITY NAME   COORDINATES      ADDRESSES
                                       X      Y
-------------------------------------------------------------------------------
   846930886              Chennai     13     80       Kovalam,  Mahabalipuram
  1681692777                Delhi     29     77       Rajpath Marg,  South Delhi
  1714636915         Bhubaneshwar     20     86       Khandagiri
  1804289383            Bangalore     13     78       Koramangala,  Kalyan Nagar
  1957747793               Bhopal     23     77       Van Vihar,

-------------CITY DATABASE----------------------------
1.     View Database
2.     Sort Database
3.     Exit
Your Choice : 3

Process finished with exit code 0
```

main.c

```c
#include <stdio.h>
#include "database.h"
#include "binary_tree.h"
#include "city_data.h"
#include "vector.h"
#include "debug_helper.h"
#include "input_helper.h"

void menu();
void sort_menu();
```

```
int main() {

    /* Initialize the database to work with CityData */
    Database* mydatabase = newDatabase(compare_name, compare_id, printCityUtil);

    /* Initial Data */
    mydatabase -> add(mydatabase,
            newCity("Bangalore", 13, 78,
                    newMinimalVectorWithArgs(string_compare, printAddressUtil,
                            new_string("Koramangala,"),
                            new_string("Kalyan Nagar"))));
    mydatabase -> add(mydatabase,
            newCity("Chennai", 13, 80,
                    newMinimalVectorWithArgs(string_compare, printAddressUtil,
                            new_string("Kovalam,"),
                            new_string("Mahabalipuram"))));
    mydatabase -> add(mydatabase,
            newCity("Delhi", 29, 77,
                    newMinimalVectorWithArgs(string_compare, printAddressUtil,
                            new_string("Rajpath Marg,"),
                            new_string("South Delhi"))));
    mydatabase -> add(mydatabase,
            newCity("Bhubaneshwar", 20, 86,
                    newMinimalVectorWithArgs(string_compare, printAddressUtil,
                            new_string("Khandagiri"))));
    mydatabase -> add(mydatabase,
            newCity("Bhopal", 23, 77,
                    newMinimalVectorWithArgs(string_compare, printAddressUtil,
                            new_string("Van Vihar"))));
    int choice;
    while(1) {
        menu();
        choice = next_int();
        switch (choice) {
            case 1: {
                printCityDataHeader();
                mydatabase -> seq_data -> print(mydatabase -> seq_data);
            }
                break;
            case 2: {
                sort_menu();
                choice = next_int();
                switch (choice) {
                    case 1: {
                        printf("\nSorting by City Name");
                        mydatabase -> seqComparator = compare_name;
                        mydatabase -> sortSeqData(mydatabase);
                        printf("\nSORTED !\n");
                    }
                        break;
                    case 2: {
                        printf("\nSorting by City ID");
                        mydatabase -> seqComparator = compare_id;
                        mydatabase -> sortSeqData(mydatabase);
                        printf("\nSORTED !\n");
                    }
                        break;
                    default:
                        break;
                }
            }
                break;
            case 3:
```

```
                return 0;
            default:
                break;
        }
    }
}

void menu() {
    printf("\n-------------CITY DATABASE----------------------------\n"
           "1.\tView Database\n"
           "2.\tSort Database\n"
           "3.\tExit\n"
           "Your Choice : ");
}

void sort_menu() {
    printf("\n-------------Sort DATABASE---------------------------\n"
           "1.\tSort by City Name\n"
           "2.\tSort by ID\n"
           "3.\tBack\n"
           "Your Choice : ");
}
```

Question No. 3

Solution to Question No. 2 Part B:

B 2.1 Introduction to search algorithms:

Searching algorithms form an important part of any program, searching in simple terms is finding an element in a given data structure. Generally, the location of the element in the list is returned from the list of the elements. For example, in searching a_n from $[a_0,\ a_1,\ a_2, \ldots\ a_n]$ the index n is returned, else a negative 1 is returned if the element is not found in the list.

Some commonly used algorithms used are linear search and binary search, linear search works for both sorted and unsorted data and has to iterate over each and every element from the data, hence the worst case is that it will have to visit each and every element from the list, the worst case is $O(n)$. While on the other hand binary search works on only sorted data, and this is more efficient than linear search since every leaf from the tree should not be visited, if you imagine the data structure of a binary tree, which is sorted or more precisely a binary search tree, while searching the elements, after every comparison half of the tree is cut, hence the time complexity decreases drastically, if there are 2^n elements, since every leaf can have 2 branches, and if the height is N, then our search's time complexity is same as the height of the tree, $N = 2^n, n = \log_2 N$, the time complexity of a BST, or Binary Search in a sorted data is $O(\log N)$.

B 2.2 Design of an efficient data structure along with the corresponding algorithm:

A Database Structure would be best suited for such an application, this kind of structure should be able to store any kind of data, including the primitive ones like int, double, char, char*, and also user defined ones like structures, hence we use a generic data structure, BTree and Vector, which have void* as the data defined, hence we can store any kind of data in them.

Database contains a Binary Tree to store the data in a sorted manner, by using the comparator method passed to the database. This helps in improving the time-complexity in searching the database.

The Database has more generic methods such as

(i) add : to add any form of data to the database
(ii) delete : to delete a data from the database
(iii) print : to print the database
(iv) printOne : to print one data from the database
(v) search : to search data from the database, requires the comparator method for the data stored in the database
(vi) sortSeqData : to sort the sequential data stored in the vector.

```
struct Database {
    BTree* data_root;
    Vector* seq_data;
    int (*add)(struct Database*, void* data);
    int (*delete)(struct Database*, char* id);
```

```
    int (*search)(struct Database*,
            void* search_term,
            int (*comparator)(const void* data1, const void* data2));
    void (*print)(struct Database*);

    /* Sequential Data Methods */
    void (*sortSeqData)(struct Database*);
    int (*seqComparator)(const void*, const void*);

    /* DataType specific methods */
    int (*comparator)(const void* data1, const void* data2);
    void (*printOne)(void* data);
};
```

The Data Structure used for the City Data is pretty straight forward, it contains the String name and the coordinates of the city as integers X, and Y, since there are going to be a lot of addresses, and we do not know the length of that list, we use a Vector, which can arbitrarily take any length of elements. We also generate an id for each of the city in order to make sure that there are no duplicates in the database, this number is generated in sequence so there are no collisions that take place. There's another method "print" which is a utility method to print the city data.

```
struct CityData {
    char* id;
    char* name;
    int X;
    int Y;
    Vector* address;
    void (*print)(struct CityData*);
};

struct Coordinates {
    int X;
    int Y;
};
```

B 2.3 A C program:

<u>See Appendix A for the complete implementation of the DatabaseRecords in C.</u>

<u>OUTPUT:</u>

```
D:\Assignment-SEM03-02-2018\CSC202A\DatabaseRecords\cmake-build-debug\DatabaseRecords.exe

-------------CITY DATABASE----------------------------
1.     View Database
2.     Add City to Database
3.     Search in Database
4.     Delete Records
5.     View Sequential Data
6.     Exit
Your Choice : 1
          ID            CITY NAME    COORDINATES       ADDRESSES
                                        X      Y
-------------------------------------------------------------------------------
  1804289383            Bangalore     13     78        Koramangala,  Kalyan Nagar,  Peenya
Industrial Layout#  Sanjay Nagar  Yeshwanthpur
```

```
  1957747793              Bhopal     23     77       Van Vihar,  Lower Lake,  Peer Gate
  1714636915         Bhubaneshwar     20     86       Khandagiri,        Anand    Bazar,
Madhusudan Nagar
   846930886             Chennai     13     80       Kovalam,  Mahabalipuram,  Mylapore
  1681692777               Delhi     29     77       Rajpath Marg,  South Delhi,  Netaji
Subhash Marg,  Shambhu Dayal Bagh

-------------CITY DATABASE----------------------------
1.     View Database
2.     Add City to Database
3.     Search in Database
4.     Delete Records
5.     View Sequential Data
6.     Exit
Your Choice : 2

Enter Name of the City : Mumbai
Enter Coordinate X and Y : 19 72
Enter the Addresses :
Prabhadevi,
Goregaon East,
Chowpatty Beach,
Marine Drive,
Juhu Beach


-------------CITY DATABASE----------------------------
1.     View Database
2.     Add City to Database
3.     Search in Database
4.     Delete Records
5.     View Sequential Data
6.     Exit
Your Choice : 1
          ID            CITY NAME   COORDINATES      ADDRESSES
                                       X      Y
-------------------------------------------------------------------------------
  1804289383            Bangalore     13     78       Koramangala,  Kalyan Nagar,  Peenya
Industrial Layout#  Sanjay Nagar  Yeshwanthpur
  1957747793              Bhopal     23     77       Van Vihar,  Lower Lake,  Peer Gate
  1714636915         Bhubaneshwar     20     86       Khandagiri,        Anand    Bazar,
Madhusudan Nagar
   846930886             Chennai     13     80       Kovalam,  Mahabalipuram,  Mylapore
  1681692777               Delhi     29     77       Rajpath Marg,  South Delhi,  Netaji
Subhash Marg,  Shambhu Dayal Bagh
   424238335              Mumbai     19     72       Prabhadevi,       Goregaon   East,
Chowpatty Beach,  Marine Drive,  Juhu Beach

-------------CITY DATABASE----------------------------
1.     View Database
2.     Add City to Database
3.     Search in Database
4.     Delete Records
5.     View Sequential Data
6.     Exit
Your Choice : 3

-------------SEARCH DATABASE--------------------------
1.     Search by City Name
2.     Search by Coordinates
3.     Back
Your Choice : 1

Enter City Name : Bhubaneshwar
          ID            CITY NAME   COORDINATES      ADDRESSES
                                       X      Y
-------------------------------------------------------------------------------
```

```
  1714636915         Bhubaneshwar     20     86       Khandagiri,         Anand    Bazar,
Madhusudan Nagar

-------------CITY DATABASE----------------------------
1.     View Database
2.     Add City to Database
3.     Search in Database
4.     Delete Records
5.     View Sequential Data
6.     Exit
Your Choice : 4

-------------DELETE RECORD----------------------------
1.     Delete by City Name
2.     Delete by Coordinates
3.     Back
Your Choice : 1

Enter City Name : Delhi
  1681692777                Delhi     29     77       Rajpath Marg,  South Delhi,  Netaji
Subhash Marg,  Shambhu Dayal Bagh

-------------CITY DATABASE----------------------------
1.     View Database
2.     Add City to Database
3.     Search in Database
4.     Delete Records
5.     View Sequential Data
6.     Exit
Your Choice : 1
          ID            CITY NAME   COORDINATES      ADDRESSES
                                       X      Y
--------------------------------------------------------------------------------
  1804289383            Bangalore     13     78       Koramangala,  Kalyan Nagar,  Peenya
Industrial Layout#  Sanjay Nagar  Yeshwanthpur
  1957747793               Bhopal     23     77       Van Vihar,  Lower Lake,  Peer Gate
  1714636915         Bhubaneshwar     20     86       Khandagiri,         Anand    Bazar,
Madhusudan Nagar
   846930886              Chennai     13     80       Kovalam,  Mahabalipuram,  Mylapore
   424238335               Mumbai     19     72       Prabhadevi,       Goregaon    East,
Chowpatty Beach,  Marine Drive,  Juhu Beach

-------------CITY DATABASE----------------------------
1.     View Database
2.     Add City to Database
3.     Search in Database
4.     Delete Records
5.     View Sequential Data
6.     Exit
Your Choice : 6

Process finished with exit code 0
```

B 2.4 Computation of time and space complexity:

The various operations that can be performed such as add, delete, search, print are the operations for which the time and space complexity is to be calculated.

1. Add

```
int add_to_database(Database* mydatabase, void* data) {
    mydatabase -> seq_data -> add(mydatabase -> seq_data, data);
    insertion_sort(mydatabase -> seq_data -> data, (size_t)mydatabase -> seq_data ->
length, sizeof *(mydatabase -> seq_data -> data), compare_id);
    BTree* newNode = newLeaf(data, mydatabase -> printOne);
    add_to_tree(&mydatabase -> data_root, newNode, mydatabase -> comparator);
```

```
}
```

$$Time\ Complexity\ for\ Add$$
$$= Time\ Complexity\ for\ (Add\ to\ Vector + Insertion\ Sort + Add\ to\ Tree\ )$$

Add to Vector is adding to an Array of Data and the time complexity for that is $O(1)$.

Insertion Sort has a time complexity of n^2 in the worst case, but since the data is sorted at every step and is almost always nearly sorted it can be taken as $O(n \log n)$.

Add to Tree is addition to a Binary Search Tree. The worst-case time complexity of search and insert operations is $O(h)$ where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become $O(n)$.

Hence,

$$TAdd(n) = O(1) + O(n \log n) + O(n)$$
$$TAdd(n) = O(n \log n)$$

The Space Complexity for Insertion Sort is $O(\log N)$

$$Space\ Complexity\ for\ Add$$
$$= Space\ Complexity\ for\ (Add\ to\ Vector + Insertion\ Sort + Add\ to\ Tree\ )$$

$$SAdd(n) = O(1) + O(\log N) + O(1)$$
$$SAdd(n) = O(\log N)$$

2. Delete

```
int index = linear_search_database(mydatabase, input, compare_data_and_name);
/* delete from the tree and then from the seq_data */
CityData* toDelete = mydatabase -> seq_data -> data[index];
/* Since the BST was made with name, use name as param to delete from tree */
mydatabase -> data_root = delete_from_tree(mydatabase -> data_root, toDelete -> name,
compare_data_and_name);
mydatabase -> seq_data -> remove(mydatabase -> seq_data, index);
```

$$Time\ Complexity\ for\ Delete$$
$$= Time\ Complexity\ for\ Search + Time\ Complexity\ for\ Delete\ from\ BST$$
$$+ Time\ Complexity\ for\ Delete\ from\ Vector$$

The time complexity for Linear Search is $O(n)$

The worst-case time complexity of delete operation is $O(h)$ where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of delete operation may become $O(n)$.

The time complexity for deleting from the vector, since the index is known is $O(1)$

$$TDelete(n) = O(n) + O(n) + O(1)$$
$$TDelete(n) = O(n)$$

$$Space\ Complexity\ for\ Delete$$
$$= Space\ Complexity\ for\ Search + Space\ Complexity\ for\ Delete\ from\ BST$$
$$+ Space\ Complexity\ for\ Delete\ from\ Vector$$

$$SDelete = O(1) + O(1) + O(1)$$
$$SDelete = O(1)$$

3. Searching

The time complexity for Searching in BST

$$T(n) = T\left(\frac{N}{2}\right) + O(1)$$

Applying Master's Theorem for computing Run-Time Complexity of the recurrence relation

$$T(n) = aT\left(\frac{N}{b}\right) + f(N)$$

Here a = 1, and b = 2, hence

$$f(n) = n^c \log^k n\,, k = 0\ c = \log_b a$$

Hence,

$$T(n) = O\left(N^c \log^{k+1} N\right) = O(\log_2 N)$$

$$TSearch(n) = O(\log N)$$

Since we are only searching for one element at a time, the time complexity for Searching in a BST is $O(1)$.

Bibliography

1. Graph Theory: Definition and Properties, Dr. Jean-Paul Rodrigue and Dr. Cesar Ducruet.
2. Application of Graph Theory in Transportation Networks, Sanjay kumar Bisen
3. Graph Theory use in Transportation Problems and Railway Networks, Sanjay Kumar Bisen.
4. The Algorithm Design Manual by Steven S. Skiena, 2nd Edition, pp 105

database.h

```
//
// Created by shadowleaf on 25-Sep-18.
//

#ifndef DATABASERECORDS_DATABASE_H
#define DATABASERECORDS_DATABASE_H

#include "vector.h"
#include "binary_tree.h"

struct Database {
    BTree* data_root;
    Vector* seq_data;
    int (*add)(struct Database*, void* data);
    int (*delete)(struct Database*, char* id);
    int (*search)(struct Database*,
            void* search_term,
            int (*comparator)(const void* data1, const void* data2));
    void (*print)(struct Database*);

    /* Sequential Data Methods */
    void (*sortSeqData)(struct Database*);
    int (*seqComparator)(const void*, const void*);

    /* DataType specific methods */
    int (*comparator)(const void* data1, const void* data2);
    void (*printOne)(void* data);
};

typedef struct Database Database;

Database* newDatabase(
        int (*)(const void*, const void*),
        int (*)(const void*, const void*),
        void (*)(void*));

int add_to_database(Database* mydatabase, void* data);

void printDatabase(Database* mydatabase);

void sort_seq_data(Database* mydatabase);

int search_database(Database*, void* param, int (*)(const void*, const void*));
int linear_search_database(Database*, void* param, int (*)(const void*, const void*));

#endif //DATABASERECORDS_DATABASE_H
```

database.c

```
//
```

```c
// Created by shadowleaf on 25-Sep-18.
//

#include <stdlib.h>
#include "database.h"
#include "city_data.h"
#include "vector.h"
#include "debug_helper.h"

/* When you create an instance of my Database, tell me how can i compare two of your data
inputs,
 * tell me how can i print one of those data, that's it, i'll take care of things from
here*/
Database* newDatabase(
        int (*comparator)(const void* data1, const void* data2),
        int (*seq_comparator)(const void* data1, const void* data2),
        void (*printOne)(void* data)) {
    Database* mydatabase = malloc(sizeof *mydatabase);
    mydatabase -> data_root = NULL;
    mydatabase -> seq_data = newMinimalVector(comparator, printOne);
    mydatabase -> add = add_to_database;
    mydatabase -> print = printDatabase;
    mydatabase -> search = search_database;

    /* Sequential Data Method */
    mydatabase -> sortSeqData = sort_seq_data;
    mydatabase -> seqComparator = seq_comparator;

    /* DataType specific methods */
    mydatabase -> comparator = comparator;
    mydatabase -> printOne = printOne;

    return mydatabase;
}

/* To Search the database */
// Comparator must check between your data and your parameter of your data
int search_database(Database* mydatabase, void* param, int (*comparator)(const void*,
const void*)) {
    BTree* res = search_tree(mydatabase -> data_root, param, comparator);
    if (res != NULL) {
        res -> printData(res -> data);
        return 1;
    } else {
        printf("\n NO SUCH RECORD EXIST!\n");
        return -1;
    }
}

/* Return the position of data in Sequential List of data, then do whatever you want with
it */
int linear_search_database(Database* mydatabase, void* param, int (*comparator)(const
void*, const void*)) {
    return (mydatabase -> seq_data -> search(mydatabase -> seq_data, param, comparator));
}
```

```
/* To add data to the database */
int add_to_database(Database* mydatabase, void* data) {
    mydatabase -> seq_data -> add(mydatabase -> seq_data, data);
    insertion_sort(mydatabase -> seq_data -> data, (size_t)mydatabase -> seq_data ->
length, sizeof *(mydatabase -> seq_data -> data), compare_id);
    BTree* newNode = newLeaf(data, mydatabase -> printOne);
    add_to_tree(&mydatabase -> data_root, newNode, mydatabase -> comparator);
}

/* To Print the complete database */
void printDatabase(Database* mydatabase) {
    printBTree(mydatabase -> data_root);
}

/* Sequential Data Methods */
void sort_seq_data(Database* mydatabase) {
    insertion_sort(mydatabase -> seq_data -> data, (size_t)mydatabase -> seq_data ->
length, sizeof *(mydatabase -> seq_data -> data), mydatabase -> seqComparator);
}
```

citydata.h

```
//
// Created by shadowleaf on 25-Sep-18.
//

#ifndef DATABASERECORDS_CITY_DATA_H
#define DATABASERECORDS_CITY_DATA_H

#include <string.h>
#include <stdio.h>
#include "vector.h"

struct CityData {
    char* id;
    char* name;
    int X;
    int Y;
    Vector* address;
    void (*print)(struct CityData*);
};

struct Coordinates {
    int X;
    int Y;
};

typedef struct Coordinates Coordinates;

typedef struct CityData CityData;

/* Constructors */
CityData* newCity(char*, int, int, Vector*);
Coordinates* newCoordinates(int, int);

/* Print Utils */
void printCityDataHeader();
void printCityData(CityData*);
void printCityUtil(void* _mycity);
void printAddressUtil(void* address);

/* Take Input */
```

```c
CityData* readInputCity();

/* Comparator Methods */
int compare_name(const void* data1, const void* data2);
int compare_data_and_name(const void* data, const void* name);
int compare_data_and_coordinates(const void* data, const void* coordinates);
int compare_id(const void* data1, const void* data2);

#endif //DATABASERECORDS_CITY_DATA_H
```

citydata.c

```c
//
// Created by shadowleaf on 25-Sep-18.
//

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <memory.h>
#include <limits.h>
#include "city_data.h"
#include "input_helper.h"

/* Constructors */
CityData* newCity(char* name, int X, int Y, Vector* address) {
    CityData* mycity = malloc(sizeof *mycity);
    mycity -> name = name;
    mycity -> X = X;
    mycity -> Y = Y;
    mycity -> print = printCityData;
    mycity -> address = address;

    /* Generate a random UID */
    srand((unsigned) time(NULL));
    mycity -> id = malloc(32 * sizeof *mycity -> id);
    sprintf(mycity -> id, "%lu", random());

    return mycity;
}

Coordinates* newCoordinates(int X, int Y) {
    Coordinates* mycoordinates = malloc(sizeof *mycoordinates);
    mycoordinates -> X = X;
    mycoordinates -> Y = Y;

    return mycoordinates;
}

/* Print Utils */

void printCityDataHeader(){
    printf("%12s %20s %13s\t%s\n",
            "ID", "CITY NAME", "COORDINATES", "ADDRESSES");
    printf("%12s %20s %6s %6s\n",
            "","","X","Y");
    for (int i = 0 ; i < 80 ; i++)
        printf("-");
    printf("\n");
}

void printCityUtil(void* _mycity) {
    CityData* mycity = (CityData*) _mycity;
```

```
    printCityData(mycity);
}

void printAddressUtil(void* address) {
    char* _address = (char*) address;
    printf(" %s ", _address);
}

void printCityData(CityData* mycity) {
//    printf("\nId\t\t: %s"
//           "\nName\t: %s"
//           "\nX\t\t: %d"
//           "\nY\t\t: %d"
//           "\n", mycity -> id, mycity -> name, mycity -> X, mycity -> Y);
    printf("%12s %20s %6d %6d\t",
           mycity -> id, mycity -> name, mycity -> X, mycity -> Y);
    // printf("Addresses : ");
    mycity -> address -> print(mycity -> address);
    printf("\n");
}

CityData* readInputCity() {
    char* buffer;
    char* name;
    int X, Y;
    Vector* addressVector;
    printf("\nEnter Name of the City : ");
    get_word(&name);
    printf("Enter Coordinate X and Y : ");
    X = next_int();
    Y = next_int();

    addressVector = newMinimalVector(string_compare, printAddressUtil);
    char* address;
    printf("Enter the Addresses : \n");
    while (get_line(&address) > 1) {
        addressVector -> add(addressVector, new_string(address));
    }

    return newCity(name, X, Y, addressVector);
}

/* Comparator Methods */
int compare_name(const void* data1, const void* data2) {
    return strcmp(((CityData*)data1) -> name, ((CityData*)data2) -> name);
}

int compare_id(const void* data1, const void* data2) {
    return atoi(((CityData*)data1) -> id) - atoi(((CityData*)data2) -> id);
}

int compare_data_and_name(const void* data, const void* name) {
    return strcmp(((CityData*)data) -> name, (char*)name);
}

int compare_data_and_coordinates(const void* data, const void* coordinates) {
    if (((CityData*)data) -> X == ((Coordinates*)coordinates) -> X
        && ((CityData*)data) -> Y == ((Coordinates*)coordinates) -> Y)
        return 0;

    return -1;
}
```

binary_tree.h

```
//
// Created by shadowleaf on 25-Sep-18.
//

#ifndef DATABASERECORDS_BINARY_TREE_H
#define DATABASERECORDS_BINARY_TREE_H

struct BTree {
    struct BTree* left;
    void* data;
    struct BTree* right;

    void (*add)(struct BTree** root, struct BTree* data,
            int (*comparator)(void* data1, void* data2));
    void (*printData)(void* leaf);
};

typedef struct BTree BTree;

BTree* newLeaf(void* data, void (*printData)(void* leaf));

BTree* search_tree(BTree* root, void* param, int (*comparator)(const void* data1, const
void* data2));

void add_to_tree(BTree** root, BTree* data, int (*comparator)(const void* data1, const
void* data2));
BTree* delete_from_tree(BTree* root, void* param, int (*comparator)(const void* data1,
const void* data2));

/* Helper Method */
BTree* min_leaf(BTree*);

void printBTree(BTree* node);
#endif //DATABASERECORDS_BINARY_TREE_H
```

binary_tree.c

```
//
// Created by shadowleaf on 25-Sep-18.
//

#include <stdlib.h>
#include <stdio.h>
#include "binary_tree.h"
#include "debug_helper.h"
#include "city_data.h"

/* Creates a New Leaf, initializes it and return it */
BTree* newLeaf(void* data, void (*printData)(void* data)) {
    BTree* myleaf = malloc(sizeof *myleaf);

    myleaf -> left = NULL;
    myleaf -> data = data;
    myleaf -> right = NULL;
```

```
    myleaf -> add = add_to_tree;
    myleaf -> printData = printData;

    return myleaf;
}
/* Compares the Leaf with the parameter using comparator */
BTree* search_tree(BTree* root, void* param, int (*comparator)(const void* data1, const
void* data2)){
    if (root == NULL || comparator(root -> data, param) == 0) {
        return root;
    }

    if (comparator(root -> data, param) >= 0) {
        return search_tree(root -> right, param, comparator);
    } else {
        return search_tree(root -> left, param, comparator);
    }

}

/* Compares two Leaves with Comparator that can compare them */
void add_to_tree(BTree** root, BTree* data, int (*comparator)(const void* data1, const
void* data2)) {
    if (*root == NULL) {
        *root = data;
        return;
    }

    if (comparator((*root) -> data, data -> data) >= 0) {
        add_to_tree(&((*root) -> right), data, comparator);
    } else {
        add_to_tree(&((*root) -> left), data, comparator);
    }
}

/* Deletes a leaf from the BTree if it exists, uses comparator to check
 * between the leaf and data of leaf*/
BTree* delete_from_tree(BTree* root, void* param, int (*comparator)(const void* data1,
const void* data2)) {
    if (root == NULL)
        return root;

    if (comparator(root -> data, param) > 0) {
        root -> right = delete_from_tree(root -> right, param, comparator);
    } else if (comparator(root -> data, param) < 0) {
        root -> left = delete_from_tree(root -> left, param, comparator);
    } else {
        /* Found you dammit, Delete this goddamn leaf! */

        // case 1 and 2: leaf with only one kid or no kids
        if (root -> left == NULL) {
            BTree* rightKid = root -> right;
            free(root);
            return rightKid;
        } else if (root -> right == NULL) {
```

```
            BTree* leftKid = root -> left;
            free(root);
            return leftKid;
        }

        // case 3: leaf with two kids
        BTree* temp = min_leaf(root -> right);
        free(root -> data);
        root -> data = temp -> data;

        /* Delete the in-order successor */
        root -> right = delete_from_tree(root -> right, ((CityData*)(temp -> data)) ->
name, comparator);

    }
    return root;
}

/* Helper Method */
/* Returns the left-most leaf, or the minimum value leaf */
BTree* min_leaf(BTree* leaf) {
    BTree* current_leaf = leaf;

    while (current_leaf -> left != NULL)
        current_leaf = current_leaf -> left;

    return current_leaf;
}

/* Prints the Tree In-Order */
void printBTree(BTree* node) {
    if (node == NULL) {
        return;
    }

    printBTree(node -> right);
    node -> printData(node -> data);
    printBTree(node -> left);
}
```

vector.h

```
#ifndef VECTOR_H
#define VECTOR_H

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "vector.h"

/* For vargs without a list count */
#define PP_NARG(...) \
         PP_NARG_(__VA_ARGS__,PP_RSEQ_N())
#define PP_NARG_(...) \
         PP_ARG_N(__VA_ARGS__)
#define PP_ARG_N( \
          _1, _2, _3, _4, _5, _6, _7, _8, _9,_10, \
         _11,_12,_13,_14,_15,_16,_17,_18,_19,_20, \
         _21,_22,_23,_24,_25,_26,_27,_28,_29,_30, \
```

```
        _31,_32,_33,_34,_35,_36,_37,_38,_39,_40, \
        _41,_42,_43,_44,_45,_46,_47,_48,_49,_50, \
        _51,_52,_53,_54,_55,_56,_57,_58,_59,_60, \
        _61,_62,_63,_64,_65,_66,_67,_68,_69,_70, \
        _71,_72,_73,_74,_75,_76,_77,_78,_79,_80, \
        _81,_82,_83,_84,_85,_86,_87,_88,_89,_90, \
        _91,_92,_93,_94,_95,_96,_97,_98,_99,_100, \
        _101,_102,_103,_104,_105,_106,_107,_108,_109,_110, \
        _111,_112,_113,_114,_115,_116,_117,_118,_119,_120, \
        _121,_122,_123,_124,_125,_126,_127,N,...) N
#define PP_RSEQ_N() \
        127,126,125,124,123,122,121,120, \
        119,118,117,116,115,114,113,112,111,110, \
        109,108,107,106,105,104,103,102,101,100, \
        99,98,97,96,95,94,93,92,91,90, \
        89,88,87,86,85,84,83,82,81,80, \
        79,78,77,76,75,74,73,72,71,70, \
        69,68,67,66,65,64,63,62,61,60, \
        59,58,57,56,55,54,53,52,51,50, \
        49,48,47,46,45,44,43,42,41,40, \
        39,38,37,36,35,34,33,32,31,30, \
        29,28,27,26,25,24,23,22,21,20, \
        19,18,17,16,15,14,13,12,11,10, \
        9,8,7,6,5,4,3,2,1,0

typedef int (*Comparator)(const void*, const void*);

/* The Vector structure */
struct Vector {
    int     length;
    void    **data;
    void    (*print)(struct Vector*);
    void    (*add)(struct Vector*, void *);
    void    (*remove)(struct Vector*, int);
    int     (*comparator)(const void *, const void *);
    int     (*comparator_r)(const void *, const void *);
    void    (*sort)(struct Vector*, bool);
    int     (*search)(struct Vector*, void*, int (*)(const void*, const void*));
    void    (*printOne)(void* data);
};

typedef struct Vector Vector;

#define _COMPARATOR_ int (*)(const void*, const void*)
#define _PRINTONE_ void (*)(void*)

Vector* _newMinimalVectorWithArgs(_COMPARATOR_, _PRINTONE_, size_t argc, ...);
#define newMinimalVectorWithArgs(_COMPARATOR_, _PRINTONE_, ...) \
_newMinimalVectorWithArgs(_COMPARATOR_, _PRINTONE_,PP_NARG(__VA_ARGS__), __VA_ARGS__)

/* Generic methods */
Vector* newVector(void (*)(Vector*), _COMPARATOR_, _COMPARATOR_);
Vector* newMinimalVector(_COMPARATOR_, _PRINTONE_);
void add(Vector *, void *);
void init(Vector *, void (*)(Vector *), _COMPARATOR_, _COMPARATOR_);
void del(Vector*, int);
void sort(Vector*, bool descending);
void print_vector(Vector* list);

/* DataType specific methods */
void*   new_int(int);
void*   new_double(double);
void*   new_string(char*);
void    print_int(Vector *);
```

```c
void    print_string(Vector *);

/* Comparator methods */
int     int_compare(const void *, const void *);
int     int_compare_r(const void *, const void *);
int     string_compare(const void *, const void *);
int     string_compare_r(const void *, const void *);

/* Sorting Methods */
void insertion_sort(void** arr, size_t length, size_t ele_size, _COMPARATOR_);

/* Searching Algorithms */
int linear_search_vector(Vector*, void* param, _COMPARATOR_);

#endif
```

vector.c

```c
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include "vector.h"
#include "debug_helper.h"
#include "city_data.h"

typedef int (*Comparator)(const void*, const void*);

Vector* newVector(void (*print_util)(Vector*),
        int (*compare)(const void*, const void*),
        int (*compare_r)(const void*, const void*)) {
    Vector* myVector = malloc(sizeof *myVector);
    init(myVector, print_util, compare, compare_r);
    return myVector;
}

Vector* newMinimalVector(Comparator comparator,
        void (*printOne)(void*)) {
    Vector* myVector = malloc(sizeof *myVector);
    myVector -> length = 0;
    myVector -> data = NULL;
    myVector -> add = add;
    myVector -> remove = del;
    myVector -> print = print_vector;
    myVector -> printOne = printOne;
    myVector -> search = linear_search_vector;
    myVector -> comparator = comparator;

    return myVector;
}

Vector* _newMinimalVectorWithArgs(Comparator comparator, void (*printOne)(void*), size_t
argc, ...) {
    Vector* thisVector = newMinimalVector(comparator, printOne);

    va_list valist;

    // va_start takes the valist and the last parameter before the "..."
    va_start(valist, argc);
    for (int i = 0 ; i < argc ; i++) {
        // printf("*%d*", va_arg(valist, int));
        thisVector -> add (thisVector, va_arg(valist, void*));
    }
    va_end(valist);
```

```
    return thisVector;
}

void init(Vector *list,
        void (*print_util)(Vector*),
        int (*compare)(const void*, const void*),
        int (*compare_r)(const void*, const void*)) {
    list -> length  = 0;
    list -> data    = NULL;
    list -> print   = print_util;
    list -> add     = add;
    list -> remove  = del;
    list -> comparator = compare;
    list -> comparator_r = compare_r;
    list -> sort = sort;
    list -> search = linear_search_vector;
}

void add(Vector *list, void * DATA) {
    list -> data = realloc(list -> data, sizeof *(list -> data) * (list -> length + 1));
    (list -> data)[(list -> length)] = DATA;
    list -> length++;
}

void del(Vector *list, int index) {
    for (int i = index ; i < list -> length - 1 ; i++) {
        (list -> data)[i] = (list -> data)[i+1];
    }
    list -> length --;
    list -> data = realloc(list -> data, sizeof *(list -> data) * (list -> length));
}

void sort(Vector *list, bool descending) {
    if (descending)
        qsort(list -> data, (size_t)list -> length,
            sizeof *(list -> data),
            list -> comparator_r);
    else
        qsort(list -> data, (size_t)list -> length,
            sizeof *(list -> data),
            list -> comparator);
}

/* Searching Algorithms */
int linear_search_vector(Vector* list, void* param, int (*comparator)(const void*, const
void*)) {
    for (int i = 0 ; i < list -> length ; i++) {
        if (comparator(list -> data[i], param) == 0) {
            list->printOne(list->data[i]);
            return i;
        }
    }

    return -1;
}

/* Comparator Functions */
int int_compare(const void * a1, const void * a2) {
    // di(**((const int**)a1));
    if (**((const int**) a1) > **((const int**) a2)) return 1;
    if (**((const int**) a1) < **((const int**) a2)) return -1;
    return 0;
}
```

```
int int_compare_r(const void * a1, const void * a2) {
    // di(**((const int**)a1));
    if (**((const int**) a1) > **((const int**) a2)) return -1;
    if (**((const int**) a1) < **((const int**) a2)) return 1;
    return 0;
}

int string_compare(const void * s1, const void * s2) {
    return strcmp(*(const char**)s1, *(const char**)s2);
}

int string_compare_r(const void * s1, const void * s2) {
    return (-1)*strcmp(*(const char**)s1, *(const char**)s2);
}

/* Generic Print Utility */
void print_vector(Vector* list) {
    for (int i = 0 ; i < list -> length ; i++) {
        list -> printOne(list -> data[i]);
    }
}

/*Print Utils*/

void print_int(Vector *list) {
    for (int i = 0 ; i < list -> length ; i++) {
        di(*(int *)((list -> data)[i]));
    }
}

void print_string(Vector *list) {
    for (int i = 0 ; i < list -> length ; i++) {
        ds((char *)((list -> data)[i]))
    }
}

/* New Data of Specific DataType Methods */

void* new_int(int data) {
    int* new_data = malloc(sizeof *new_data);
    *new_data = data;
    return new_data;
}

void* new_double(double data) {
    double* new_data = malloc(sizeof *new_data);
    *new_data = data;
    return new_data;
}

void* new_string(char* data) {
    char* new_data = malloc(strlen(data) * sizeof *new_data);
    strcpy(new_data, data);
    return new_data;
}

/* Other Sorting Techniques */
            void insertion_sort(void** arr, size_t length, size_t ele_size, int
(*comparator)(const void* data1, const void* data2)) {
                void* key;
                for (int i = 1 ; i < length ; i++) {
                    key = arr[i];
                    int j = i - 1;
```

```
                    while (j >= 0 && comparator(arr[j], key) >= 0) {
                        arr[j+1] = arr[j];
                        j--;
        }
        arr[j+1] = key;
    }
}
```

input_helper.h

```
#ifndef INPUT_HELPER_H
#define INPUT_HELPER_H

int get_word(char**);

int get_sentence(char**);

int get_line(char** line);

int next_int();

char* sanitize_string(char*);
char* strlwr(char*);

#endif
```

input_helper.c

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include "input_helper.h"
#include "debug_helper.h"

/* get_word: takes a pointer to a pointer to char
 * stops reading the input if a space is encountered
 * or EOF or new line */
int get_word(char** word) {
    int c, len = 0, maxlen = 1;

    /* Let's have a minimum of some memory
     * I know the OS will put some padded memory, but still
     * mann ki shaanti ke liye kar leta hun */
    *word = malloc((len+1) * sizeof **word);

    /* Skip the whitespaces */
    while((c=getchar()) == ' ')
        ;
    if (c != EOF)
        ungetc(c, stdin);

    while ((c = getchar()) != ' ' && c != '\n' && c != EOF) {
        if (len == maxlen) {
            maxlen *= 2;
            *word = realloc(*word, maxlen * sizeof **word);
```

```
        }
        (*word)[len++] = c;
    }

    if (len != 0)
        *word = realloc(*word, len * sizeof **word);

    (*word)[len] = '\0';

    return 0;
}

/* get_sentence: takes a pointer to a pointer to char
 * stops reading the input if a newline is encountered
 * or EOF */
int get_sentence(char** sentence) {
    int c, len = 0, maxlen = 1;

    /* Let's have a minimum of some memory
     * I know the OS will put some padded memory, but still
     * mann ki shaanti ke liye kar leta hun */
    *sentence = malloc((len+1) * sizeof **sentence);

    while ((c = getchar()) && c != '\n' && c != EOF) {
        if (len == maxlen) {
            maxlen *= 2;
            *sentence = realloc(*sentence, maxlen * sizeof **sentence);
        }
        (*sentence)[len++] = c;
    }

    if (len != 0)
        *sentence = realloc(*sentence, len * sizeof **sentence);

    (*sentence)[len] = '\0';

    return 0;
}

/* Takes a complete line of input and returns the length
 * because if you have len = 0 you can stop reading */
int get_line(char** line) {
    int c, len = 0, maxlen = 1;

    *line = malloc((len+1) * sizeof **line);

    while ((c = getchar()) && c != '\n' && c != EOF) {
        if (len == maxlen) {
            maxlen *= 2;
            *line = realloc(*line, maxlen * sizeof **line);
        }
        (*line)[len++] = c;
    }

    if (len != 0)
        *line = realloc(*line, len * sizeof *line);
```

```c
    /* removes the new line, if present */
    if ((*line)[len] == '\n' && len != 0) --len;
    (*line)[len] = '\0';

    return len;
}

/* DataTypes input */
int next_int() {
    char *read;
    get_word(&read);
    int integer = atoi(read);
    free(read);
    return integer;
}


/* Removes the leading and trailing spaces from the string */
char* sanitize_string(char* s) {
    size_t size;
    char* end;

    size = strlen(s);

    if (!size) return s;

    end = s + size - 1;

    while (end >= s && isspace(*end))
        end--;
    *(end + 1) = '\0';

    while (*s && isspace(*s))
        s++;

    return s;
}

char* strlwr(char* s) {
    for (int i = 0 ; i < strlen(s) ; i++)
        s[i] = tolower(s[i]);

    return s;
}
```

debug_helper.h

```c
#ifndef DEBUG_HELPER_H
#define DEBUG_HELPER_H

#define ds(s) printf("\nDEBUG--*"#s " : %s*\n", s);
#define dc(c) printf("\nDEBUG--%"#c " : %c%\n", c);
#define di(i) printf("\nDEBUG--#"#i " : %d#\n", i);
//#define dd(d) printf("\nDEBUG--$"#d " : %.15f\n", *(double*)d);
#define dd(d) printf("\nDEBUG--$"#d " : %.15f\n", d);
```

```
#endif
```

main.c

```
#include <stdio.h>
#include "database.h"
#include "binary_tree.h"
#include "city_data.h"
#include "vector.h"
#include "debug_helper.h"
#include "input_helper.h"

void menu();
void search_menu();
void delete_menu();

int main() {

    /* Initialize the database to work with CityData */
    Database* mydatabase = newDatabase(compare_name, compare_id, printCityUtil);

    /* Initial Data */
    mydatabase -> add(mydatabase,
            newCity("Bangalore", 13, 78,
                    newMinimalVectorWithArgs(string_compare, printAddressUtil,
                            new_string("Koramangala,"),
                            new_string("Kalyan Nagar,"),
                            new_string("Peenya Industrial Layout"),
                            new_string("Sanjay Nagar"),
                            new_string("Yeshwanthpur"))));
    mydatabase -> add(mydatabase,
            newCity("Chennai", 13, 80,
                    newMinimalVectorWithArgs(string_compare, printAddressUtil,
                            new_string("Kovalam,"),
                            new_string("Mahabalipuram,"),
                            new_string("Mylapore"))));
    mydatabase -> add(mydatabase,
            newCity("Delhi", 29, 77,
                    newMinimalVectorWithArgs(string_compare, printAddressUtil,
                            new_string("Rajpath Marg,"),
                            new_string("South Delhi,"),
                            new_string("Netaji Subhash Marg,"),
                            new_string("Shambhu Dayal Bagh"))));
    mydatabase -> add(mydatabase,
            newCity("Bhubaneshwar", 20, 86,
                    newMinimalVectorWithArgs(string_compare, printAddressUtil,
                            new_string("Khandagiri,"),
                            new_string("Anand Bazar,"),
                            new_string("Madhusudan Nagar"))));
    mydatabase -> add(mydatabase,
            newCity("Bhopal", 23, 77,
                    newMinimalVectorWithArgs(string_compare, printAddressUtil,
                            new_string("Van Vihar,"),
                            new_string("Lower Lake,"),
                            new_string("Peer Gate"))));
    int choice;
    while(1) {
        menu();
        choice = next_int();
        switch (choice) {
            case 1: {
                printCityDataHeader();
                mydatabase->print(mydatabase);
```

```
            }
                break;
            case 2: {
                mydatabase->add(mydatabase, readInputCity());
            }
                break;
            case 3: {
                search_menu();
                char* input;
                choice = next_int();
                switch (choice) {
                    case 1: {
                        printf("\nEnter City Name : ");
                        get_word(&input);
                        printCityDataHeader();
                        mydatabase -> search(mydatabase, input, compare_data_and_name);
                        free(input);
                    }
                        break;
                    case 2: {
                        int X, Y;
                        printf("\nEnter Coordinates : ");
                        X = next_int(); Y = next_int();
                        Coordinates* coordinates = newCoordinates(X, Y);
                        linear_search_database(mydatabase, coordinates,
compare_data_and_coordinates);
                    }
                        break;
                    default:
                        break;
                }
            }
                break;
            case 4: {
                delete_menu();
                char* input;
                choice = next_int();
                switch (choice) {
                    case 1: {
                        printf("\nEnter City Name : ");
                        get_word(&input);
                        int index = linear_search_database(mydatabase, input,
compare_data_and_name);
                        /* delete from the tree and then from the seq_data */
                        CityData* toDelete = mydatabase -> seq_data -> data[index];
                        /* Since the BST was made with name, use name as param to delete
from tree */
                        mydatabase -> data_root = delete_from_tree(mydatabase ->
data_root, toDelete -> name, compare_data_and_name);
                        mydatabase -> seq_data -> remove(mydatabase -> seq_data, index);
                        free(input);
                    }
                        break;
                    case 2: {
                        int X, Y;
                        printf("\nEnter Coordinates : ");
                        X = next_int(); Y = next_int();
                        Coordinates* coordinates = newCoordinates(X, Y);
                        int index = linear_search_database(mydatabase, coordinates,
compare_data_and_coordinates);
                        /* delete from the tree and then from the seq_data */
                        CityData* toDelete = mydatabase -> seq_data -> data[index];
                        mydatabase -> data_root = delete_from_tree(mydatabase ->
data_root, toDelete -> name, compare_data_and_name);
```

```
                        mydatabase -> seq_data -> remove(mydatabase -> seq_data, index);
                    }
                        break;
                    default:
                        break;
                }
            }
                break;
            case 5: {
                printCityDataHeader();
                mydatabase -> seq_data -> print(mydatabase -> seq_data);
            }
                break;
            case 6:
                return 0;
            default:
                break;
        }
    }
}

void menu() {
    printf("\n-------------CITY DATABASE-----------------------------\n"
           "1.\tView Database\n"
           "2.\tAdd City to Database\n"
           "3.\tSearch in Database\n"
           "4.\tDelete Records\n"
           "5.\tView Sequential Data\n"
           "6.\tExit\n"
           "Your Choice : ");
}

void search_menu() {
    printf("\n-------------SEARCH DATABASE---------------------------\n"
           "1.\tSearch by City Name\n"
           "2.\tSearch by Coordinates\n"
           "3.\tBack\n"
           "Your Choice : ");
}

void delete_menu() {
    printf("\n-------------DELETE RECORD-----------------------------\n"
           "1.\tDelete by City Name\n"
           "2.\tDelete by Coordinates\n"
           "3.\tBack\n"
           "Your Choice : ");
}
```