Complex Engineering Problem

Submitted by:

Fariaa Faheem 2019-EE-1

Marwa Waseem 2019-EE-7

Hamza Akhtar 2019-EE-12

Filza Shahid 2019-EE-151

Supervised by: Prof. Khalid Butt

Department of Electrical Engineering

University of Engineering and Technology Lahore

Complex Engineering Problem

Submitted to the faculty of the Electrical Engineering Department
of the University of Engineering and Technology Lahore
in partial fulfillment of the requirements for the Degree of

Bachelor of Science

in

Electrical Engineering.

Internal Examiner

External Examiner

Director
Undergraduate Studies

Department of Electrical Engineering

University of Engineering and Technology Lahore

Declaration

I declare that the work contained in this thesis is my own, except where explicitly stated otherwise. In addition this work has not been submitted to obtain another degree or professional qualification.

Signed: _____

Date: _____

Contents

List of Figures

Chapter 1

Problem Statement

The main objectives of this Complex Engineering Problem are:

- To develop programs that store and manage the given data by using four different data structures, that are:
  1. Hash Table (Quadratic Probing)
  2. Array
  3. Linked List
  4. Binary Tree
- Implement the following operations on the given data:
  1. Insert all of the given data in a data structure.
  2. Print data of data structure in sorted order (traverse in sorted order) (numerically or alphabetically).
  3. Find records.
  4. Delete half of the data from the data structure.
- To measure execution time and memory consumption for each operation.
- To compare operations on different data structures depending on their execution time and memory consumption and conclude which data structure is the best for each operation.

Chapter 2

File Input

2.1 Methodology

The workflow for file input routines is follows:

- File is opened in reading mode with **fopen()**
- Reading the data file using **fscanf()** from the first line. Ignore the first string read as it is needed. Next line contains the number of records read and it store as an integer. From the next line we have the data of employees.
- Creating a structure which has fields of "ID"(Integer), "Name"(String), "City"(String) and "Service"(String).
- Allocating memory for an array of "pointers to structures".
- Records are read from the file and are stored in the allocated structures and then pointers to these structures are linked to the array.
- File is closed after storing the data with **fclose()**

So after these processes we have *an array of pointers to structures* containing the data of the files.

Chapter 3

Hash Table Implementation

3.1 Methodology

Quadratic Probing for Hash tables is used to carryout basic operations on the data array. These basic operations and their working are as follows:

3.1.1 Insertion

First of all, a hash table of size 1.5 times the size of data is created and then the data from the array is inserted into hash table by computing hash function which is id % table size. On collision the second hash function is calculated and the data is inserted at the appropriate position.

Time Complexity: **O(1) approaches to O(N)**
Space Complexity: **O(N)**

3.1.2 Finding

In order to find the data, hash function is calculated using the given id. If the required key is found the index of the cell is returned, else by using the formula of quadratic probing the next is cell checked and so on until an empty cell or the required key is reached.

Time Complexity: **O(N)**
Space Complexity: **O(N)**

3.1.3 Sorted Traversal

For sorted traversal 101, all the IDs of hash table are copied in an array, the array is sorted using quick sort algorithm. The sorted array is traversed, and for the given ID, its location is found in hash table and the data is printed

Sorting Algorithm Used: **Quick Sort O(log N)**
Time Complexity: **O(N log N)**
Space Complexity: **O(N)**

For simple sorted traversal, hash table is traversed, the minimum element is found and marked as found, and printed then the other minimum is found and printed and so on.
Time Complexity: $\mathbf{O}(N^2)$
Space Complexity: **O(N)**
Note: Sorted traversal time complexity is dependent upon sorting algorithm used.

3.1.4 Deletion

Deletion is carried out by finding the cell in which the id to be deleted is present, then the cell is marked as deleted.

Time Complexity: **O(1)**
Space Complexity: **O(N)**

3.2 Execution Times and Memory Consumptions

```
Number of Records: 1000
Insert              Execution Time:     0.000054 s      Memory Consumption:     36280 bytes
Find                Execution Time:     0.000015 s      Memory Consumption:     36280 bytes
Sorted Traversal 101 Execution Time:    0.000120 s      Memory Consumption:     40280 bytes
Sorted Traversal    Execution Time:     0.008748 s      Memory Consumption:     40280 bytes
Delete              Execution Time:     0.000039 s      Memory Consumption:     40280 bytes

--------------------------------
Process exited after 0.02986 seconds with return value 0
Press any key to continue . . .
```

FIGURE 3.1: Results for hash implementation with data size 1000.

```
Number of Records: 10000
Insert              Execution Time:     0.000785 s      Memory Consumption:     360328 bytes
Find                Execution Time:     0.000317 s      Memory Consumption:     360328 bytes
Sorted Traversal 101 Execution Time:    0.002094 s      Memory Consumption:     400328 bytes
Sorted Traversal    Execution Time:     0.864026 s      Memory Consumption:     400328 bytes
Delete              Execution Time:     0.000342 s      Memory Consumption:     400328 bytes

--------------------------------
Process exited after 0.8982 seconds with return value 0
Press any key to continue . . .
```

FIGURE 3.2: Results for hash implementation with data size 10000.

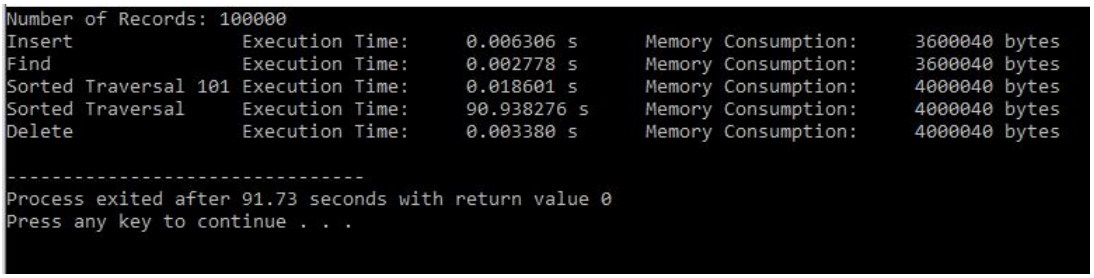

FIGURE 3.3: Results for hash implementation with data size 100000.

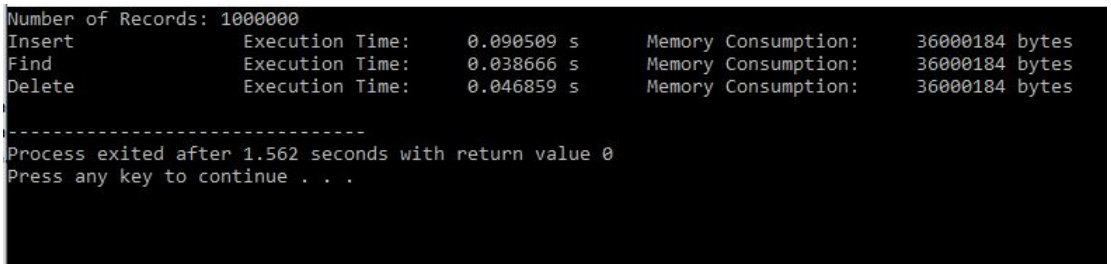

FIGURE 3.4: Results for hash implementation with data size 1000000.

Chapter 4

Array Implementation

4.1 Methodology

Simple arrays are used to carryout basic operations on the data array. These basic operations and their working are as follows:

4.1.1 Insertion

First of all an array of size of data is created. The data is inserted at the given index and that cell is marked as legitimate.

Time Complexity: **O(1)**
Space Complexity: **O(N)**

4.1.2 Finding

In order to find the data, the array is traversed until our required key is found and the index of that cell is returned.

Time Complexity: **O(N)**
Space Complexity: **O(N)**

4.1.3 Sorted Traversal

For sorted traversal, the array is sorted using quick sort algorithm and then it is traversed to print the data.

Sorting Algorithm Used: **Quick Sort O(log N)**
Time Complexity: **O(N log N)**
Space Complexity: **O(N)**

4.1.4 Deletion

Deletion is carried out by finding the cell in which the id to be deleted is present, then the cell is marked as deleted.

Time Complexity: **O(1)**
Space Complexity: **O(N)**

4.2 Execution Times and Memory Consumptions

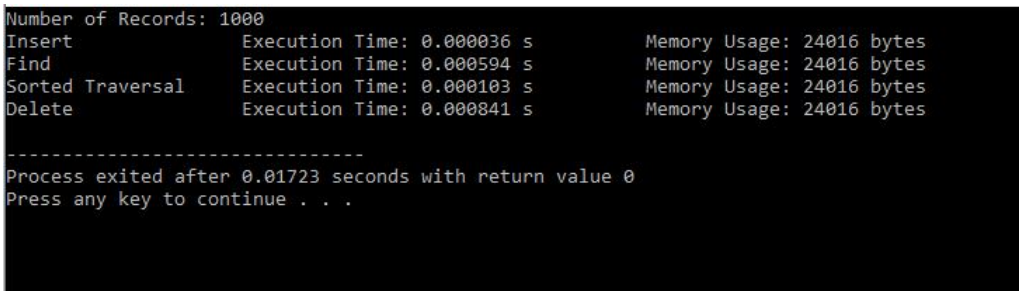

FIGURE 4.1: Results for array implementation with data size 1000.

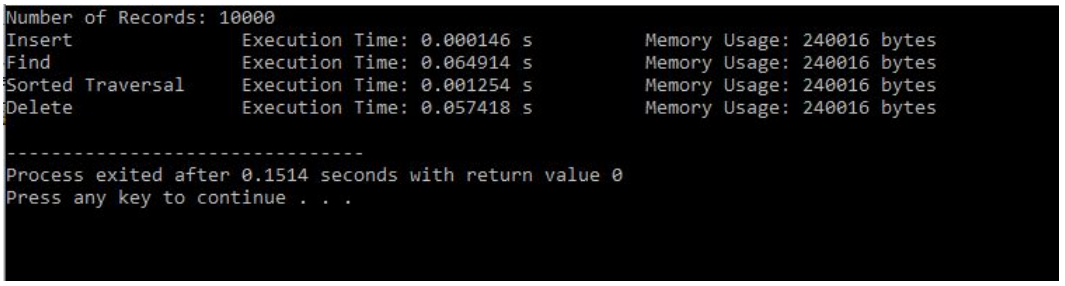

FIGURE 4.2: Results for array implementation with data size 10000.

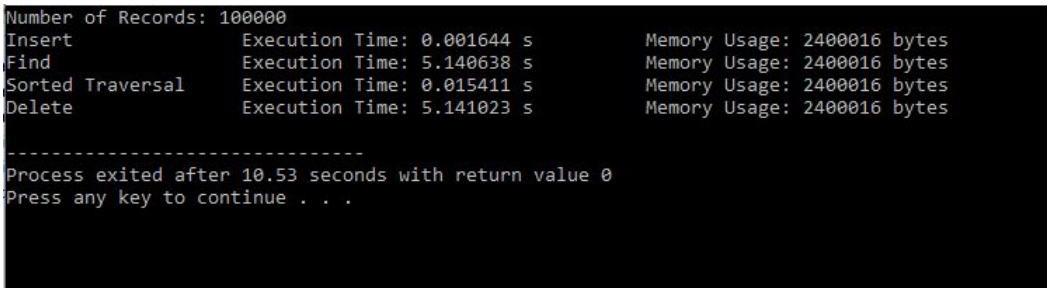

FIGURE 4.3: Results for array implementation with data size 100000.

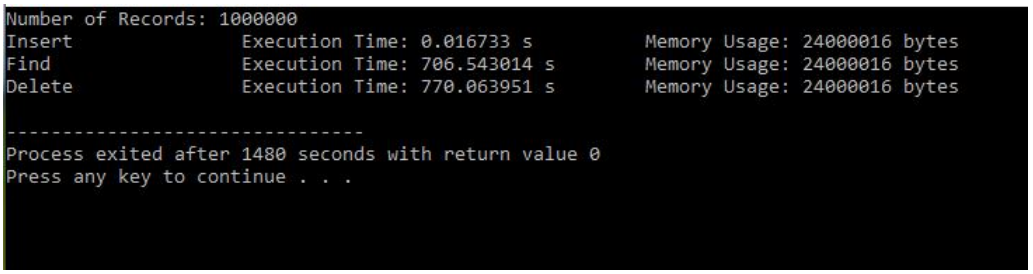

FIGURE 4.4: Results for array implementation with data size 1000000.

Chapter 5

Linked List Implementation

5.1 Methodology

Singley linked lists are used to carryout basic operations on the data array. These basic operations and their working are as follows:

5.1.1 Insertion

Insertion is done by dynamically allocating nodes. Keys i.e., ID's of employees and data is linked with these node. Finally, nodes are inserted at the head of the list.

Time Complexity: **O(1)**
Space Complexity: **O(N)**

5.1.2 Finding

There is no order in the linked list data like trees so find operation is carried out by simply traversing the list until the required key is found or tail of the list is reached.

Time Complexity: **O(N)**
Space Complexity: **O(N)**

5.1.3 Sorted Traversal

For sorted traversal, first of all list should be sorted by any convenient sorting algorithm and then traversed from head to tail.

Sorting Algrithm Used: **Quick Sort O(log N)**
Time Complexity: **O(N log N)**
Space Complexity: **O(N)**
Note: Sorted traversal time complexity is dependent upon sorting algorithm used.

5.1.4 Deletion

Deletion is carried out by finding the node to be deleted. This step involves traversing the list. After finding, the node is bypassed by link adjusment and is deleted.

Time Complexity: **O(N)**
Space Complexity: **O(N)**

5.2 Execution Times and Memory Consumptions

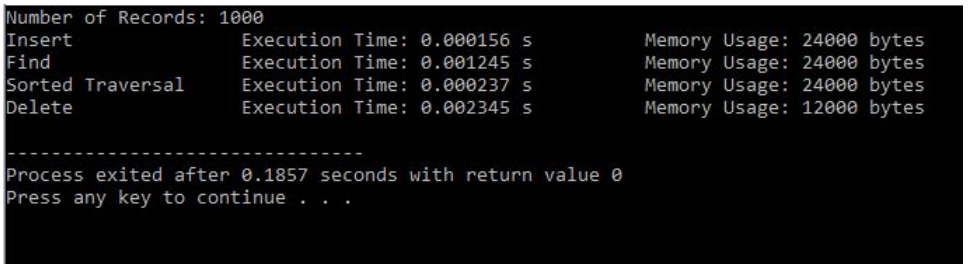

FIGURE 5.1: Results for linked list implementation with data size 1000.

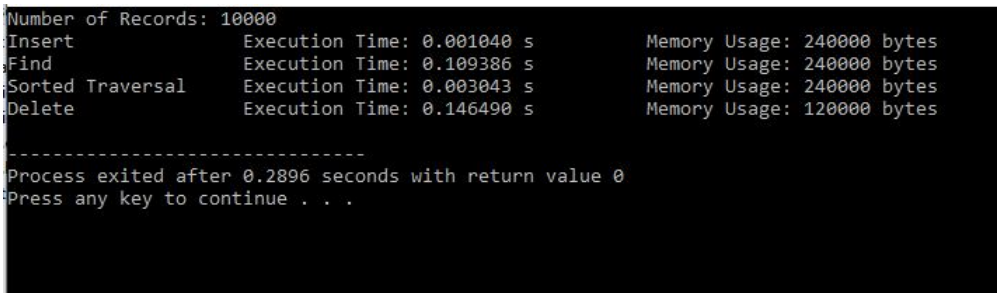

FIGURE 5.2: Results for linked list implementation with data size 10000.

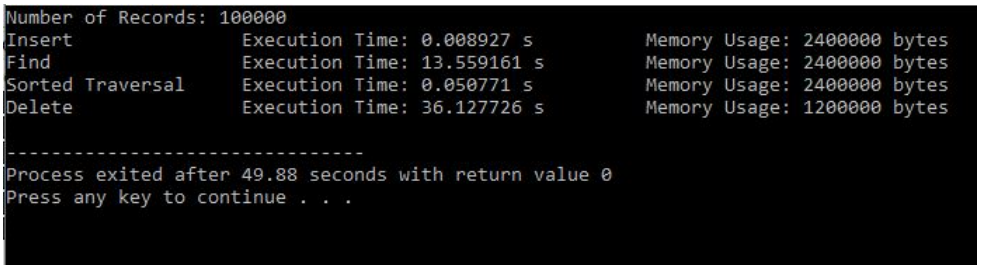

FIGURE 5.3: Results for linked list implementation with data size 100000.

Chapter 6

Tree Implementation

6.1 Methodology

Balanced trees are used to carryout basic operation on the data array. These basic operations and their working are as follows:

6.1.1 Insertion

Id's of employees are used to populate the self-balancing binary search tree i.e., *AVL trees* and then data is linked with the corresponding nodes. Tree is balanced by the phenomenon of left, right, left-right and right-left rotations.

Time Complexity: **O(log N)**
Space Complexity: **O(N)**

6.1.2 Finding

In AVL trees the nodes are arranged in specfic order. Left child node always have key less than the root node and right child will have key greater than the root node. So finding a tree node involves comparing the "key to be found" at each node if its less then only traverse the left subtree and if its larger then traverse the right subtree. In our case we found the even indexed records from data array in tree and measured its execution time and memory consumption.

Time Complexity: **O(log N)**
Space Complexity: **O(N)**

6.1.3 Sorted Traversal

Due to the order propety of AVL trees sorting traversal can be done simply by *in-order traversal* of the tree. In order traversal involves first traversing the left sub-tree recursively then visiting the root node and finally right sub-tree is traversed recursively.

Time Complexity: **O(N)** *Note: This time complexity is only for traversal*
Space Complexity: **O(N)**

6.1.4 Deletion

Deletion is carried out by going to the tree node to be deleted and then finding the minimum key or element in its right subtree and replacing the node's key with this minimum key. In this way, the order of AVL tree is maintained. In this engineering problem, we deleted all the odd indexed records from the tree.

Time Complexity: **O(log N)**
Space Complexity: **O(N)**

6.2 Execution Times and Memory Consumptions

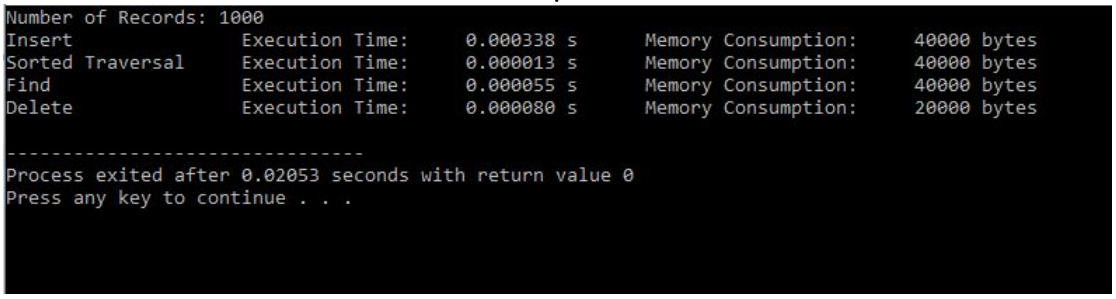

FIGURE 6.1: Results for tree implementation with data size 1000.

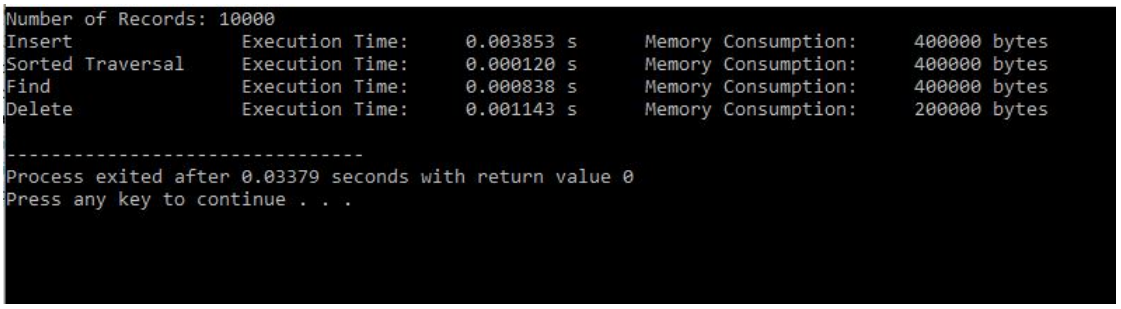

FIGURE 6.2: Results for tree implementation with data size 10000.

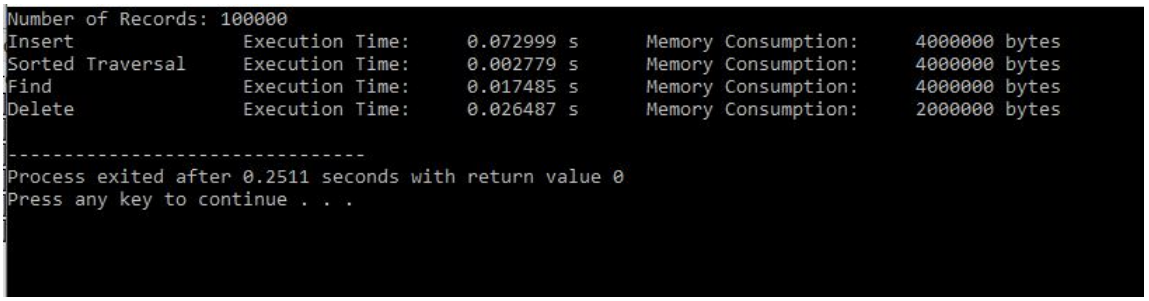

FIGURE 6.3: Results for tree implementation with data size 100000.

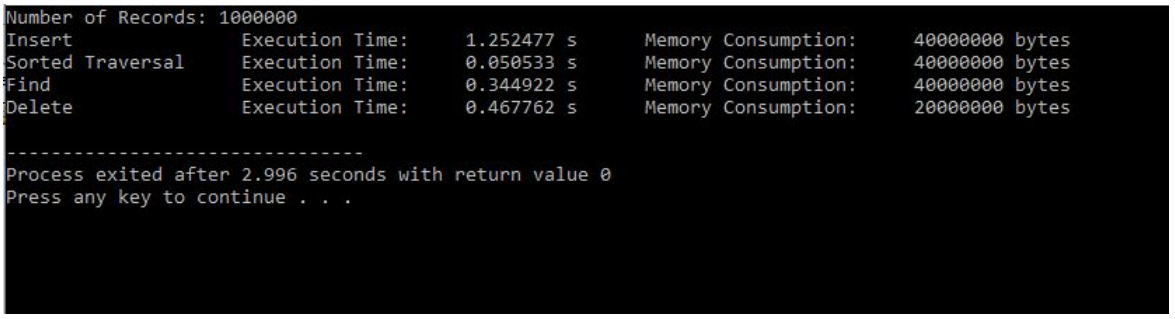

FIGURE 6.4: Results for tree implementation with data size 1000000.

Chapter 7

Results

No. of Records	Data Structure	Execution Time (s)					Memory Consumption (bytes)				
		Insert	Find	Sorted		Delete	Insert	Find	Sorted		Delete
1000	Hash Table	0.000054	0.000015	0.000120	0.008748	0.000039	36280	36280	40280	36280	36280
	Array	0.000036	0.000594	0.000103		0.000841	24016	24016	24016		24016
	Linked List	0.000156	0.001245	0.000237		0.002345	24000	24000	24000		12000
	Tree	0.000338	0.000055	0.000013		0.000080	40000	40000	40000		20000
10000	Hash Table	0.000785	0.000317	0.002094	0.864086	0.000342	360328	360328	400328	360328	360328
	Array	0.000146	0.064914	0.001254		0.057418	240016	240016	240016		240016
	Linked List	0.001040	0.109386	0.003043		0.146490	240000	240000	240000		120000
	Tree	0.003853	0.000838	0.000120		0.001143	400000	400000	400000		200000
100000	Hash Table	0.006306	0.002778	0.018601	90.938276	0.003380	3600040	3600040	4000040	3600040	3600040
	Array	0.001644	5.140638	0.015411		5.141023	2400016	2400016	2400016		2400016
	Linked List	0.008927	13.559161	0.050771		36.127726	2400000	2400000	2400000		1200000
	Tree	0.072999	0.017485	0.002779		0.026487	4000000	4000000	4000000		2000000
1000000	Hash Table	0.090509	0.038666	-	-	0.046859	36000184	36000184	-	-	36000184
	Array	0.016733	706.543014	-		770.063951	24000016	24000016	-		24000016
	Linked List	0.077202	2100	-		6200	24000000	24000000	-		12000000
	Tree	1.252477	0.344922	0.050533		0.467762	40000000	40000000	40000000		20000000

FIGURE 7.1: Combined results for all data structures and operations.

PS: First column of sorted for hash tables is for sorted traversal 101.

No. of Records	Data Structure	Execution Time (s)					Memory Consumption (bytes)				
		Insert	Find	Sorted		Delete	Insert	Find	Sorted		Delete
1000	Hash Table	0.000054	0.000015	0.00012	0.008748	0.000039	36280	36280	40280	36280	36280
	Array	0.000036	0.000594	0.000103		0.000841	24016	24016	24016		24016
	Linked List	0.000156	0.001245	0.000237		0.002345	24000	24000	24000		12000
	Tree	0.000338	0.000055	0.000013		0.00008	40000	40000	40000		20000
10000	Hash Table	0.000785	0.000317	0.002094	0.864086	0.000342	360328	360328	400328	360328	360328
	Array	0.000146	0.064914	0.001254		0.057418	240016	240016	240016		240016
	Linked List	0.00104	0.109386	0.003043		0.14649	240000	240000	240000		120000
	Tree	0.003853	0.000838	0.00012		0.001143	400000	400000	400000		200000
100000	Hash Table	0.006306	0.002778	0.018601	90.93828	0.00338	3600040	3600040	4000040	3600040	3600040
	Array	0.001644	5.140638	0.015411		5.141023	2400016	2400016	2400016		2400016
	Linked List	0.008927	13.55916	0.050771		36.12773	2400000	2400000	2400000		1200000
	Tree	0.072999	0.017485	0.002779		0.026487	4000000	4000000	4000000		2000000
1000000	Hash Table	0.090509	0.038666	-	-	0.046859	36000184	36000184	-	-	36000184
	Array	0.016733	706.543	-		770.064	24000016	24000016	-		24000016
	Linked List	0.077202	2100	-		6200	24000000	24000000	-		12000000
	Tree	1.252477	0.344922	0.050533		0.467762	40000000	40000000	40000000		20000000

FIGURE 7.2: Heat map.

PS: Red (minimum) Orange Yellow Green(maximum)

7.1 Conclusions

From the table above we can draw following conlcusions for the basic operations of data structures:

7.1.1 Arrays

- Execution time shows that arrays are pretty efficient in insertion and maintaining a contagious memory but this comes at the cost of high memory consumption then other data structures.
- Random access of data is a constant time operation but for specific time its linear time operation.
- Sorting is rather efficient in arrays as compared to other linear data structures. However, order property is not present in arrays like in trees.
- Deletion doesn't free memory.

7.1.2 Linked Lists

- Insertion is done in constant complexity in linked lists.
- Searching is a linear operations in linked lists and involves traversing the list node by node. Memory is non-contagious in linked lists and random access is not possible.

- Sorting depends upon sorting algorithm used but it invloves pointer manipulation which slows down the process.
- Deletion in linked lists frees up memory so its a plus point for linked lists

7.1.3 HashTables

- Insertion in starting is constant time operation but it approaches to O(N) with increasing data in Hastable.
- Searching is the strongest ability of hashtable it provides a constant time access of data when used properly.
- Sorting is rather a tedious task in hastable due to its random in nature. Sorted-Traversal101 (as shown in second coloumn in the above table) uses no memory and have a complexityO(N^2) while with the use of excess memory it can be reduced to O(N log N)
- Deletion in hastable doesn't frees up memory.

7.1.4 Tree

- Insertion is done recursively so it is O(log N) process.
- Searching is also O(log N) due to its inherent order property.
- Sorting is inherent in trees because data is already stored in an ordered manner so it only requires inorder traversal to access them
- Deletion in trees frees up memory and is O(log N) process.

7.2 System Properties

- RAM: 8GB
- CPU Base Clock: 2.70 GHz
- Architecture: x64