VISVESVARAYA TECHNOLOGICAL UNIVERSITY
JNANA SANGAMA, BELAGAVI – 590 018

A Mini Project Report on

FOOD ORDER MANAGEMENT SYSTEM USING HASHING INDEXING TECHNIQUE

Submitted in partial fulfillment of the requirements as a part of the File Structure Lab for the award of degree of

Bachelor of Engineering
in
Information Science and Engineering

Submitted by

V HARSHITHA
1RN17IS111

VIBHA S NAVALE
1RN17IS115

Faculty Incharge
Mrs. Anusha U A
Assistant Professor
Dept. of ISE, RNSIT

Coordinator
Mrs. Vinutha G K
Assistant Professor
Dept. of ISE, RNSIT

Department of Information Science and Engineering
RNS Institute of Technology
Channasandra, Dr. Vishnuvardhan Road, RR Nagar Post,
Bengaluru – 560 098

2019 – 2020

RNS Institute of Technology

Channasandra, Dr. Vishnuvardhan Road, RR Nagar Post,
Bengaluru – 560 098

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING

CERTIFICATE

This is to certify that the Mini project report entitled ***FOOD ORDER MANAGEMENT SYSTEM USING HASHING INDEXING TECHNIQUE*** has been successfully completed by **V HARSHITHA** bearing USN **1RN17IS111**, and **VIBHA S NAVALE** bearing USN **1RN17IS115,** presently VI semester students of **RNS Institute of Technology** in partial fulfillment of the requirements as a part of the ***FILE STRUCTURES LABORATORY*** for the award of the degree ***Bachelor of Engineering in Information Science and Engineering*** under **Visvesvaraya Technological University, Belagavi** during academic year **2019– 2020**. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the report deposited in the departmental library. The mini project report has been approved as it satisfies the academic requirements as a part of File Structures Laboratory for the said degree.

Mrs. Anusha U A Faculty Incharge	**Mrs. Vinutha G K** Coordinator	**Dr. M V Sudhamani** Professor and HOD

External Viva

Name of the Examiners	**Signature with date**
1. _____	_____
2. _____	_____

ABSTRACT

The Food Order Management System using Hashing Indexing Technique is a stand-alone 16-bit executable program that was developed using Turbo C++. All data is stored in ".txt" files implementing file structure concepts in the same directory as the program.

Food Order Management in most parts of our country have not yet been exposed to technology except for a few up and coming start-ups. To adapt to newer laws like Goods and Service Tax and other digitization reforms these businesses need to embrace and implement a digital way of billing and record keeping. This project will help fill the void in such restaurants. The project will help standardize this process of record keeping and updating of menus according to the current prices can be done with ease. It makes the process of food ordering easy and faster even for the customer who does not need to wait for a waiter or wait at the counter.

Hashing is a form of indexing that has great benefits when it comes to searching for records as the hash computed allows us to quickly reach the location of the record without having to go through the other records in the file.

In hashing, large keys are converted into small keys by using hash functions. The values are then stored in a data structure called hash table. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in O(1) time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

ACKNOWLEDGMENT

The fulfillment and rapture that go with the fruitful finishing of any assignment would be inadequate without the specifying the people who made it conceivable, whose steady direction and support delegated the endeavors with success.

We would like to profoundly thank **Management** of **RNS Institute of Technology** for providing such a healthy environment to carry out this Project work.

We would like to thank our beloved Director **Dr. H N Shivashankar** for his confidence feeling words and support for providing facilities throughout the course.

We would like to express my thanks to our Principal **Dr. M K Venkatesha** for his support and inspired me towards the attainment of knowledge.

We wish to place on record my words of gratitude to **Dr. M V Sudhamani,** Professor and Head of the Department, Information Science and Engineering, for being the enzyme and master mind behind my Project work.

We would like to express our profound and cordial gratitude to our Project Guide **Dr. M V Sudhamani,** Professor and HOD, Department of Information Science and Engineering for her valuable guidance, constructive comments and continuous encouragement throughout the Project work.

We would like to express our sincere gratitude to **Mrs. Vinutha G K**, Assistant Professor, Department of Information Science and Engineering, for his constant support and guidance.

We would like to thank all other teaching and non-teaching staff of Information Science & Engineering who have directly or indirectly helped me to carry out the project work. And lastly, we would hereby acknowledge and thank my parents who have been a source of inspiration and also instrumental in carrying out this Project work.

V HARSHITHA
USN: 1RN17IS111

VIBHA S NAVALE
USN: 1RN17IS115

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

Food Order Management System is based on the concept of ordering food where the restaurant employee as an admin can take food order from the customer by selecting food items, entering a customer name and their details. The system also allows to quickly and easily manage an online menu which users can browse.

It provides a user-friendly, interactive Menu Driven Interface (MDI). All data is stored in files for persistence. It is a stand-alone system. The project uses 1 file for holding the current menu.

1.1 Introduction to File Structure

A file structure is a combination of representations for data in files and of operations for accessing the data. A file structure allows the system to read, write and modify data. It might also support finding the data that matches some search criteria or reading through the data in some particular order. An improvement in file structure design may make a system hundreds of times faster. The details of the representation of the data and the implementation of the operations determine the efficiency of the file structure for particular system. A tremendous variety in the type needs of the file structure design important.

1.1.1 History

Early work with files presumed that files were on tape since most files were access was sequential and the cost of access grew in direct proportion to the size of the file. As files grew intolerably large for unaided sequential access and as storage devices such as hard disks became available indexes were added to files.

The indexes made it possible to keep a list of keys and pointers in a smaller file that could be searched more quickly with key and pointer, the user had direct access to the large primary file. But simple indexes had some of the same sequential flaws as the data file and as the indexes grew they too became difficult to manage especially for dynamic files in which the set of keys changes.

In the early 1960‟s the idea of applying tree structures emerged but trees can grow very unevenly as records are added and deleted resulting in long searches requiring many disk accesses to find a record.

The general goals of the research and development in file structures can be drawn directly from the library. We would like to get the information we need with one access disk .If it is impossible to get what we need in one access we want structures that allow us to find the target information with as few access as possible. We want our file structures to group information so we likely to get everything we need with only one trip to the disk.

In 1963, Researchers developed an elegant, self-adjusting binary tree structure called AVL tree for data in memory. The problem was that even with a balanced binary tree dozens of accesses were required to find a record in even moderate-sized files. A method was needed to keep a tree balanced when each node of thee tree was not a single record as in a binary tree but a file block containing dozens perhaps even hundreds of records. It took 10 years until a solution emerged in the form of a *B-tree*. Whereas AVL trees grow from the top down as records were added B-trees grew from the bottom up. B-trees provided excellent access performance but there was a cost: no longer could a file be accessed sequentially with efficiency. The problem was solved by adding a linked list structure at the bottom level of the B-tree. The combination of a B-tree and a sequential linked list is called a *B+ tree*.

B-trees and B+ trees provide access times that grow in proportion to logkN, where N is the number of entries in the file and k is the number of entries indexed in a single block of the B-tree structure. This means that B-trees can guarantee that you can find one file entry among millions with only 3 or 4 trips to the disk. Further B-trees guarantee that as you add and delete entries performance stays about the same.

Hashing is a good way to get what we want with a single request with files that do not change size greatly over time. Hashed indexes were used to provide fast access to files. But until recently hashing did not work well with volatile, dynamic files. Extendible dynamic hashing can retrieve information with 1 or at most 2 disk accesses no matter how big the file became.

1.1.2 About the File

When we talk about a file on disk or tape, we refer to a particular collection of bytes stored there. A file when the word is used in this sense physically exists. A disk drive may contain hundreds even thousands of these physical files.

From the standpoint of an application program a file is somewhat like a telephone line connection to a telephone network. The program can receive bytes through this phone line or send bytes down it but it knows nothing about where these bytes come from or where

they go. The program knows only about its end of the line. Even though there may be thousands of physical files on a disk a single program is usually limited to the use of only about 20 files. The application program relies on the OS to take care of the details of the telephone switching system. It could be that bytes coming down the line into the program originate from a physical file they come from the keyboard or some other input device. Similarly, bytes the program sends down the line might end up in a file or they could appear on the terminal screen or some other output device.

Although the program doesn't know where the bytes are coming from or where they are going it does know which line it is using. This line is usually referred to as the logical file, to distinguish it from the physical files on the disk or tape.

1.1.3 Various Kinds of storage of Fields and Records

RECORD: A subdivision of a file, containing data related to a single entity.

FIELD: A field is the smallest, logically meaningful, unit of information in a file.

A subdivision of a record containing single attribute of the entity which the record describes.

Field Structures:

Four of the most common methods of adding structure to files to maintain the identity of fields are:

1. Force the fields into a predictable length.
2. Begin each field with a length indicator.
3. Place a delimiter at the end of each field to separate it from the next field.
4. Use a "keyword=value" expression to identify each field and its contents.

Method 1: Fix the Length of Fields

In the above example, each field is a character array that can hold a string value of some maximum size. The size of the array is one larger than the longest string it can hold. Simple arithmetic is sufficient to recover data from the original fields.

The disadvantage of this approach is adding all the padding required to bring the fields up to a fixed length makes the file much larger. We encounter problems when data is too long to fit into the allocated amount of space. We can solve this by fixing all the fields at lengths that are large enough to cover all cases, but this makes the problem of wasted space in files even worse. Hence, this approach isn't used with data with large amount of variability in length of fields but where every field is fixed in length if there is very little variation in field lengths.

Method 2: Begin Each Field with a Length Indicator

We can count to the end of a field by storing the field length just ahead of the field. If the fields are not too long (less than 256 bytes) it is possible to store the length in a single byte at the start of each field. We refer to these fields as *length-based*.

Method 3: Separate the Fields with Delimiters

We can preserve the identity of fields by separating them with delimiters. All we need to do is choose some special character or sequence of characters that will not appear within a field and then *insert* that delimiter into the file after writing each field. White-space characters (blank, new line, tab) or the vertical bar character can be used as delimiters.

Method 4: Use a "Keyword=Value" Expression to Identify Fields

This has an advantage the others don't. It is the first structure in which a field provides information about itself. Such *self-describing structures* can be very useful tools for organizing files in many applications. It is easy to tell which fields are contained in a file.

Even if we don't know ahead of time which fields the file is supposed to contain. It is also a good format for dealing with missing fields. If a field is missing, this format makes it obvious, because the keyword is simply not there. It is helpful to use this in combination with delimiters to show division between each value and the keyword for the following field. But this also wastes a lot of space: 50% or more of the file's space could be taken up by the keywords.

A record can be defined as a set of fields that belong together when the file is viewed in terms of a higher level of organization.

Record Structures:

The five most often used methods for organizing records of a file are:

- Require the records to be predictable number of bytes in length.
- Require the records to be predictable number of fields in length.
- Begin each record with a length indicator consisting of a count of the number of bytes that the record contains.
- Use a second file to keep track of the beginning byte address for each record.
- Place a delimiter at the end of each record to separate it from the next record.

Method 1: Make the Records a Predictable Number of Bytes (Fixed-Length Record)

A *fixed-length record file* is one in which each record contains the same number of bytes. In the field and record structure shown we have a fixed number of fields, each with a predetermined length that combine to make a fixed-length record

Fixing the number of bytes in a record does not imply that the size or number of fields in the record must be fixed. Fixed-length records are often used as containers to hold variable numbers of variable-length fields. It is also possible to mix fixed and variable-length fields within a record.

Method 2: Make Records a Predictable Number of Fields

Rather than specify that each record in a file contains some fixed number of bytes we can specify that it will contain a fixed number of fields. In the figure below we have 6 contiguous fields and we can recognize fields simply by counting the fields.

Method 3: Begin Each Record with a Length Indicator

We can communicate the length of records by beginning each record with a filed containing an integer that indicates how many bytes there are in the rest of the record. This is commonly used to handle variable-length records.

Method 4: Use an Index to Keep Track of Addresses

We can use an index to keep a byte offset for each record in the original file. The byte offset allows us to find the beginning of each successive record and compute the length of each record. We look up the position of a record in the index then seek to the record in the data file.

Method 5: Place a Delimiter at the End of Each Record

It is analogous to keeping the fields distinct. As with fields, the delimiter character must not get in the way of processing. A common choice of a record delimiter for files that contain readable text is the end-of-line character (carriage return/ new-line pair or, on Unix systems, just a new-line character: \n). Here, we use a # character as the record delimiter.

1.1.4 Application of File Structure

Relative to other parts of a computer, disks are slow. One can pack thousands of megabytes on a disk that fits into a notebook computer. The time it takes to get information from even relatively slow electronic Random Access Memory (RAM) is about 120 nanoseconds. Getting the same information from a typical disk takes 30 milliseconds.

So, the disk access is a quarter of a million times longer than a memory access. Hence, disks are *very* slow compared to memory. On the other hand disks provide enormous capacity at much less cost than memory. They also keep the information stored on them when they are turned off.

Chapter 2

SYSTEM ANALYSIS

2.1 Analysis of Application

Food Order Management System greatly simplifies the ordering process for both the customer and the restaurant employees. System presents an interactive and up-to-date menu with all available options in an easy to use manner. Customer can ask the employee to choose one or more items to place an order which will land in the cart. Customer can view all the order details using search option. It maintains the time food was ordered and the restaurant employees can add the delivery time based on the ordered time. It allows restaurant employees to quickly go through the orders as they are received and process all orders efficiently and effectively with minimal delays and confusion.

The mission of our project-

1. Provide people-centered (quality, efficient, integrated and safe) services
2. Commuter responsive service planning and promotion
3. Optimize resources and build capacity
4. Adopt environment-friendly and sustainable practices
5. Strengthen commuter feedback mechanism
6. Modernize and maintain zero breakdown fleet
7. Evolve effective mechanism to monitor service performance
8. Conduct safety training, performance audits and awareness for stakeholders
9. Increase commercial revenue through monetizing land, buildings & buses
10. Increase efficiency in operations and administration
11. Ensure inter-agency coordination and multi-modal integration
12. Formulate and enforce police measures for sustainability of the service provision
13. Implement Intelligent Transport System to improve the quality of service
14. Extend travel concession to the weaker sections of the society
15. Act as an agent for cultural synthesis and national integration
16. Promote research on urban transport

2.2 Structures used to Store Fields and Records

Storing Fields

- **Fixing the Length of Fields**

In the Food Order System, the various fields are character arrays that can hold a string of characters till some maximum size. Simple arithmetic is sufficient to recover data from the original fields.

- **Separating the Fields with Delimiters**

We preserve the identity of fields by separating them with delimiters. We have chosen the vertical bar character ('|'), as the delimiter here.

Storing Records

- **Making Records a Predictable Number of Fields**

In this system, we have a fixed number of fields, each with a maximum length, that combine to make a data record. The records hold variable-length fields within them.

- **Using an Index to Keep Track of Addresses**

We use Hashing to find byte offsets for each record in the original file. They allow us to find the beginning of each successive record. We look up the position of a record using the hash function, and then seek to the record using the address generated by the hash function.

- **Using Buckets to increase the capacity at each hash location**

We can place records continuously in buckets without a delimiter as we use fixed length records and we only place a delimiter for the hash location at the end-of-line (\n).

Operations performed on File

Insertion

The system is initially used to add menu items having attributes like order id, restaurant id, customer name, phone number, order time and delivery time are entered by the restaurant employees. Validation is performed to stop duplication of ids.

Display

The system can then be used to display the entire menu. The records are displayed based on the sequence of their entry. Deleted dishes are completely erased from the file and won't be displayed.

Search

The system can then be used to search for existing records of all customers. The employee of a restaurant is prompted for an order ID, which is used as the key in searching for records in the Hash Function. The hash function uses the key to generate the byte address, which is then used to seek to the desired data record the product file. The details of the requested record, if found, are displayed, with suitable headings on the screen. If absent, a "record not found" message is displayed to the user.

Delete

The system can then be used to delete existing records from all customers. The deleted record is removed from the data file. The requested record, if found, is marked for deletion, and a "record deleted" message is displayed. If absent, a "record not found" message is displayed to the user.

Modify

If selected for modify option, the system will ask to enter the order id which is to be modified the corresponding details of the customer are also displayed and then option to modify the fields is asked to be entered. The user can then give all the related information and press enter. And the updated or modified values will be reflected in the file.

2.3 Indexing Used

Hashing

Hashing is the transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value. It is also used in many encryption algorithms. A hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0. For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on. Initially, the hash table contains no items, so every slot is empty.

Bucket − A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.

Hash Function − A hash function, h, is a mapping function that maps all the set of search keys K to the address where actual records are placed. It is a function from search keys to bucket addresses.

Hashing enables faster data retrieval. There are 2 types of hashing, these are static hashing and dynamic hashing. In this project static hashing has been implemented. In static hashing, when a search key value is provided, the hash function always computes the same address. For example, if mod-11 hash function is used, then it shall generate only 11 values. The output address shall always be same for that function. The number of buckets for the program remains unchanged at all the times.

Advantages of hashing

Records need not be sorted after any of the transaction. Hence the effort of sorting is reduced in this method. Since block address is known by hash function, accessing any record is very faster. Similarly updating or deleting a record is also very quick. This method can handle multiple transactions as each record is independent of other. i.e.; since there is no dependency on storage location for each record, multiple records can be accessed at the same time. Even if collisions occur it is still quick to go reach the correct location using hashing than with linear searching. It is suitable for online systems like net-banking, ticket booking etc. Some of the applications are:

- In cryptography hash functions can be used to hide information. One such application is in digital signatures where a so-called secure hash is used to describe a large document with a small number of bits.
- A one-way hash function is used to hide a string, for example for password protection. Instead of storing passwords in plain text, only the hash of the password is stored. To verify whether a password entered is correct, the hash of the password is compared to the stored value. These signatures can be used to authenticate the source of the document, ensure the integrity of the document as any change to the document invalidates the signature, and prevent repudiation where the sender denies signing the document.
- String commitment protocols use hash functions to hide to what string a sender has committed so that the receiver gets no information. Later, the sender sends a key that allows the receiver to reveal the string. In this way, the sender cannot change the string once it is committed, and the receiver can verify that the revealed string is the committed string. Such protocols might be used to flip a coin across the internet: The sender flips a coin and commits the result. In the meantime, the receiver calls heads or tails, and the sender then sends the key, so the receiver can reveal the coin flip.

- Hashing can be used to approximately match documents, or even parts of documents. Fuzzy matching hashes overlapping parts of a document and if enough of the hashes match, then it concludes that two documents are approximately the same. Big search engines look for similar documents so that on search result pages they don't show the many slight variations of the same document. It is also used in spam detection, as spammers make slight changes to email to bypass spam filters or to push up a document's content rank on a search results page. Geneticists use it to compare sequences of genes fragments with a known sequence of a related organism to assemble the fragments into a reasonably accurate genome.
- Hashing is used to implement hash tables. In hash tables one is given a set of keys K α and needs to map them to a range of integers so they can have stored at those locations in an array. The goal is to spread the keys out across the integers as well as possible to minimize the number of keys that collide in the array. You should not confuse the terms hash function and hash table. They are two separate ideas, and the latter uses the former.
- The hashing algorithm is called the hash function. The hash function is used to index the original value or key and then used later each time the data associated with the value or key is to be retrieved. Thus, hashing is always a one-way operation. If a hash function produces the same hash for 2 different values, it is known as collision. A hash function that offers an extremely low risk of collision may be considered acceptable.
- **Examples:** Division-remainder method, Summation and reminder method, Radix transformation method

The hash function that has been used in this project is Summation and reminder method. If the table size is 10 and all of the keys sum to five. In this case, the choice of hash function and table size needs to be carefully considered. The best table sizes are prime numbers. One problem though is that keys are not always numeric as its common for them to be strings. A possible solution: add up the ASCII values of the characters in the string to get a numeric value and then perform the Summation and reminder method.

Summation and reminder method: The sum of all the digits of the key in its numeric form is first calculated. Then the sum is then shrunk to the table size by performing a modulus operation on it with the maximum size of the hash table. This technique of summing up and bringing down the key to the smaller hash table size format is called Summation and reminder.

Linear Probing: The technique used to resolve collision is linear probing. Linear probing is a scheme in computer programming for resolving collisions in hash tables, data structures for maintaining a collection of key–value pairs and looking up the value associated with a given key. Along with quadratic probing and double hashing, linear probing is a form of open addressing. When the hash function causes a collision by mapping a new key to a cell of the hash table that is already occupied by another key, linear probing searches the table for the closest following free location and inserts the new key there. Lookups are performed in the same way, by searching the table sequentially starting at the position given by the hash function, until finding a cell with a matching key or an empty cell. In linear probing, to insert a key k, it first checks h (k) and then checks each following location in succession, wrapping around as necessary, until it finds an empty location. That is, the probe is h (k, i) = (h (k) + i) mod m. Each position determines a single probe sequence, so there are only m possible probe sequences.

The problem with linear probing is that keys tend to cluster. It suffers from primary clustering: Any key that hashes to any position in a cluster (not just collisions), must probe beyond the cluster and adds to the cluster size. Worse yet, primary clustering not only makes the probe sequence longer, it also makes it more likely that it will be lengthen further.

Chapter 3

SYSTEM DESIGN

3.1 Design of the Fields and Records

The order ID is declared as a character array that can hold a maximum of 8 characters. All the fields, restaurant code, customer name, phone number, order time, deliver time are declared as character arrays, quantity has a predetermined range, only within which it is accepted during input. The class declaration of a typical menu file record and member functions is as shown in Figure 3.1.

```
class Order
{
public:
  char order_no[12],restaurant_code[10],cust_name[12],phone[12],order_time[7],delivery_time[7],order[60], amount_paid[5];
public:
        void initial();
        void read();
        void pack();
        void unpack();
        int  retrieve(int addr,char k[],int i);
        void datadisp();
        int  remove(int addr,char k[]);
        void modify(int addr ,char k[]);
};
```

Figure 3.1 Menu Class Declaration

3.2 User Interface

The User Interface or UI refers to the interface between the system and the user. Here, the UI is menu-driven, that is, a list of options (menu) is displayed to the user and the user is prompted to enter an integer corresponding to the choice that represents the desired operation to be performed.

3.2.1 Insertion of a Record

If the operation desired is addition of a new customer, the user is required to enter 1 as his/her choice, from the menu, after which a new screen is displayed. Next, the user is prompted to enter the order ID, restaurant ID, customer name, phone number, orders that can be selected from a menu. Then the user is prompted to enter the order time and delivery time. After all the values are accepted, a “record inserted” message is displayed.

3.2.2 Display of a Record

For display of records, the user is required to enter 3 as his/her choice from the menu displayed. If there are no records in any file, “no records found” message is displayed. Suitable headings are given and for each dish displayed, all the field details are displayed.

3.2.3 Deletion of a Record

If the operation desired is deletion, the user is required to enter 4 as his/her choice, from the menu displayed. The admin is prompted to enter the order ID, of whose record is to be deleted. If there are no records that match then, a “no records to found” message is displayed. If the record is found, details of the particular customer is displayed followed by a “record deleted” message. In each case, the user is then prompted to press any key to return back to the menu screen.

3.2.4 Search for a Record

If the operation desired is search, the user is required to enter 2 as his/her choice, from the menu displayed, after which a new screen is displayed. Next, the user is prompted to enter the order ID, whose record is to be searched for. If there are no records matching, a no records found message is displayed. If the record is found, the details of the record, with suitable headings, are displayed. In each case, the user is then prompted to press any key to return back to the menu screen.

3.2.5 Modification of a Record

For the modify operation, the user is required to enter 5 as his/her choice, from the menu displayed. Next, the user is prompted to enter the order ID, whose matching record is to be updated.

The order ID entered has no matching record, a “record not found” message is displayed, and the user is then prompted to press any key to return back to the menu screen. If it is found, the record data is displayed with relevant headings and the user is then prompted to enter the modified order ID, restaurant ID, customer name, phone number and the food menu is displayed to select a new dish, along with new order and delivery time After all the values are accepted, a “record updated” message is displayed, and the admin is prompted to press any key to return back to the menu screen.

Chapter 4

IMPLEMENTATION

Implementation is the process of defining how the system should be built, ensuring its working and meets quality standards. It is a systematic and structured approach for effectively integrating a software-based service or component into the requirements of end users.

4.1 About C++

4.1.1 Classes and Objects

An order class is created with order number, customer name, location, phone number, order time, delivery time and amount as its data members. An object of this class type is used to store the values entered by the user, for the fields represented by the above data members, to be written to the data file. Class objects are created and allocated memory only at runtime.

4.1.2 File Handling

Files form the core of this project and are used to provide persistent storage for user entered information on disk. The open () and close () methods, as the names suggest, are defined in the C++ Stream header file fstream.h, to provide mechanisms to open and close files. The physical file handles used to refer to the logical filenames, are also used to ensure files exist before use and that existing files aren't overwritten unintentionally. The 2 types of files used are data files and index files. open () and close () are invoked on the file handle of the file to be opened/closed. open () takes 2 parameters- the filename and the mode of access. close () takes no parameters.

4.1.3 Character Arrays and Character functions

Character arrays are used to store the record fields to be written to data files. They are also used to store the ASCII representations of the integer that are written to the data file. Character functions are defined in the header file ctype.h and also in string.h. Some of the functions used include:

- toupper() – used to convert lowercase characters to uppercase characters.
- itoa() – to store ASCII representations of integers in character arrays
- atoi() – to convert an ASCII value into an integer.
- strlen() – returns the length of the character array
- strcpy() – copies one string to another.
- strcat() – concatenates one string to the end of the next.

4.2 Pseudocode

4.2.1 Insertion Module Pseudocode

```
void Order::read()
{
        int addr,len,len1,len2,len3,len4,len5;
        char ord_no[12],drestaurant_code[10],dorder[8];

   LABEL:
        cout<<"\n\t\t\tENTER THE ORDER ID: ";
        gets(ord_no);
        len=strlen(ord_no);
        addr=hash(ord_no);
        if(len!=8)
        {
                cout<<"\n\t  ENTER THE VALID ORDER ID(8 CHARACTERS): ";
                goto LABEL;
        }

        if(retrieve(addr,ord_no,0)>0)
        {
                cout<<"!!*** ORDER ID IS ALREADY PRESENT, WE CANNOT ADD IT TO THE HASH FILE ***!!\n";
                goto LABEL;
        }
        file.close();
        strcpy(order_no,ord_no);

      DESC:
        cout<<"\n\t\t\tENTER RESTAURANT CODE: ";
        gets(drestaurant_code);
        len1=strlen(drestaurant_code);
        if(len1!=8||drestaurant_code[0]>63||drestaurant_code[1]<63||drestaurant_code[2]<63||drestaurant_code[3]<63 ||drestaurant_code[4]<63)
        {
                cout<<"\n\t  ENTER THE VALID RESTAURANT CODE(8 CHARACTERS): ";
                goto DESC;
        }
        file.close();
        strcpy(restaurant_code,drestaurant_code);
}
```

Figure 4.1 Read Function Code

The read() function (Figure. 4.1) takes a new record as input to the data file if the key number entered is not a duplicate. Values are inserted after computing the hash byte offset address and linear probing is done to find the actual position of placement of record.

4.2.2 Display Module Pseudocode

The function shown in Figure 4.2 retrieves all the records in the file by checking if the first five characters are not equal to hash (#) which is used as the tombstone here. The first 5 characters not being hash implies there is a record stored in that record position which is then traversed through from top to bottom of the file, so the data display is not entry sequenced or hash sequenced.

```
file.seekg(i*recsize*bucketsize,ios::beg);
file.getline(dummy,5,'\n');
if(strcmp(dummy,"####")!=0 && strcmp(dummy,"$$$$")!=0)
{
        file.seekg(i*recsize*bucketsize,ios::beg);
        file.getline(order_no,10,'|');
        unpack();
        clrscr();
      //drawdoublelinebox(17,7,60,30);
      textcolor(15);
      textbackground(RED);
   cout<<endl<<endl<<endl;
   cout<<setw(25)<<" "<<"DISPLAYING FOOD ORDER RECORDS"<<endl<<endl;
   for(int i=0;i<80;i++)   cout<<'*';   cout<<endl;
   cout<<setw(20)<<" "<<" ORDER ID               :  "<<setw(10)<<order_no<<endl<<endl;
   cout<<setw(20)<<" "<<" RESTAURANT CODE        :  "<<setw(9)<<restaurant_code<<endl<<endl;
   cout<<setw(20)<<" "<<" CUSTOMER NAME          :  "<<setw(10)<<cust_name<<endl<<endl;
   cout<<setw(20)<<" "<<" PHONE NUMBER           :  "<<setw(10)<<phone<<endl<<endl;
   cout<<setw(20)<<" "<<" YOUR ORDER             :  "<<setw(6)<<order<<endl<<endl;
   cout<<setw(20)<<" "<<" ORDER TIME             :  "<<setw(5)<<order_time<<endl<<endl;
   cout<<setw(20)<<" "<<" DELIVERY TIME          :  "<<setw(5)<<delivery_time<<endl<<endl;
   cout<<setw(20)<<" "<<" AMOUNT PAID            :  "<<setw(5)<<amount_paid<<endl<<endl;
   for(int k=0;k<80;k++) cout<<'*';
   cout<<endl<<endl<<setw(20)<<" ";
   getch();

}
```

Figure 4.2 The display module

4.2.3 Deletion Module Pseudocode

The delete option shown in Figure 4.3 uses the same code as the retrieve function and once the record is found and displayed similar to the search option, the entire space allocated for the record (78 bytes) is replaced with hash which acts as the tombstone. This allows any new record to be stored in this record position. If the record is not found, the retrieve function displays a “Record not found” message similar to the search option in the menu.

```
if(strcmp(order_no,k)==0)
{
        found=1;
        unpack();
        clrscr();
        textcolor(15);
      textbackground(1);
   cout<<endl<<endl;
   cout<<setw(26)<<" "<<"FOOD ORDER RECORD DISPLAYED"<<endl;
   for(int l=0;l<80;l++)   cout<<'*';   cout<<endl;
   cout<<setw(20)<<" "<<" ORDER ID              :  "<<setw(10)<<order_no<<endl<<endl;
   cout<<setw(20)<<" "<<" RESTAURANT CODE       :  "<<setw(9)<<restaurant_code<<endl<<endl;
   cout<<setw(20)<<" "<<" CUSTOMER NAME         :  "<<setw(10)<<cust_name<<endl<<endl;
   cout<<setw(20)<<" "<<" PHONE NUMBER          :  "<<setw(10)<<phone<<endl<<endl;
   cout<<setw(20)<<" "<<" YOUR ORDER            :  "<<setw(6)<<order<<endl<<endl;
   cout<<setw(20)<<" "<<" ORDER TIME            :  "<<setw(5)<<order_time<<endl<<endl;
   cout<<setw(20)<<" "<<" DELIVERY TIME         :  "<<setw(5)<<delivery_time<<endl<<endl;
   cout<<setw(20)<<" "<<" AMOUNT PAID           :  "<<setw(5)<<amount_paid<<endl<<endl;
   for(int z=0;z<80;z++) cout<<'*';
   cout<<endl<<setw(20)<<" ";

        textcolor(15);
      textbackground(1);
        cout<<"FOOD ORDER RECORD TO BE DELETED IS FOUND"<<endl;
        file.seekp(i*recsize*bucketsize,ios::beg);
      for(s=0;s<recsize;s++)
           file.put('$');
        cout<<setw(25)<<" "<<"  RECORD IS DELETED... !!"<<endl;
        getch();
        file.close();
        break;
}
```

Figure 4.3 The deletion module

4.2.4 Modify Module Pseudocode

The modify option shown in Figure 4.4 is implemented by calling the remove() function and then the read() and store() functions after that. The call to the remove function along with few other statements work similar to the delete option. This is followed by a call to the read and store functions, thus reading a new record contents from the user and storing it in the file.

```
void Order::modify(int addr, char k[])
{
int found=0,i;
char dummy[10];
i=addr;
int s;
int x;
char confirm;

file.open(datafile,ios::in|ios::out);
cout<<setiosflags(ios::left);
do
{
        file.seekg(i*recsize*bucketsize,ios::beg);
        file.getline(dummy,5,'\n');
        if(strcmp(dummy,"####")==0)
                break;
       file.seekg(i*recsize*bucketsize,ios::beg);
        file.getline(order_no,10,'|');
        if(strcmp(order_no,k)==0)
        {
                found=1;
                unpack();
                clrscr();
                textcolor(YELLOW);
               textbackground(0);
           cout<<endl;
           cout<<setw(18)<<" "<<"FOOD ORDER RECORD TO BE MODIFIED IS DISPLAYED"<<endl<<endl;
           for(int c=0;c<80;c++)   cout<<'*';   cout<<endl;
           cout<<setw(20)<<" "<<" ORDER ID              :  "<<setw(10)<<order_no<<endl<<endl;
           cout<<setw(20)<<" "<<" RESTAURANT CODE       :  "<<setw(9)<<restaurant_code<<endl<<endl;
           cout<<setw(20)<<" "<<" CUSTOMER NAME         :  "<<setw(10)<<cust_name<<endl<<endl;
           cout<<setw(20)<<" "<<" PHONE NUMBER          :  "<<setw(10)<<phone<<endl<<endl;
           cout<<setw(20)<<" "<<" YOUR ORDER            :  "<<setw(6)<<order<<endl<<endl;
           cout<<setw(20)<<" "<<" ORDER TIME            :  "<<setw(5)<<order_time<<endl<<endl;
           cout<<setw(20)<<" "<<" DELIVERY TIME         :  "<<setw(5)<<delivery_time<<endl<<endl;
           cout<<setw(20)<<" "<<" AMOUNT PAID           :  "<<setw(5)<<amount_paid<<endl<<endl;
           for(int v=0;v<80;v++) cout<<'*';
           cout<<endl<<setw(20)<<" ";

                textcolor(YELLOW);
              textbackground(0);
                cout<<"FOOD ORDER RECORD TO BE MODIFIED IS FOUND"<<endl;
                cout<<"ARE YOU SURE YOU WANT TO MODIFY THE RECORD?(Y/N)";
                cin>>confirm;
                if(confirm=='Y' || confirm=='y')
                {
                getch();
                file.seekp(i*recsize*bucketsize+9,ios::beg);
                for(s=0;s<recsize-9;s++)
                   file.put('$');
                  cout<<"\t\t\tRECORD DELETED...!!"<<endl;
                  getch();
                   file.close();
```

Figure 4.4 The modification module

4.2.5 Search Module Pseudocode

The retrieve function shown in Figure 4.5 works similar to display function, it checks if the first five characters are not equal to hash(#).The first 5 characters not being hash implies there is a record stored in that record position whereas being hash implies the particular record space is empty. It further checks if the key value entered by the user to search is equal to the key value of the record the file pointer is pointing to, if it is equal the particular record contents are displayed. This checking is done till the end of the file and then it loops back to the top of the file. The searching ends when we reach the point we started from. If no record is matched with the user entered key, then a "Record not found" message is displayed.

```
if(strcmp(order_no,k)==0)
{
        found=1;
        if(l==1)
        {
        unpack();
        clrscr();
        textcolor(15);
      textbackground(YELLOW);
   cout<<endl;
   cout<<setw(18)<<" "<<"THE SEARCHED FOOD ORDER RECORD IS DISPLAYED"<<endl<<endl;
   for(int f=0;f<80;f++)   cout<<'*';   cout<<endl;
   cout<<setw(20)<<" "<<" ORDER ID              :  "<<setw(10)<<order_no<<endl<<endl;
   cout<<setw(20)<<" "<<" RESTAURANT CODE       :  "<<setw(9)<<restaurant_code<<endl<<endl;
   cout<<setw(20)<<" "<<" CUSTOMER NAME         :  "<<setw(10)<<cust_name<<endl<<endl;
   cout<<setw(20)<<" "<<" PHONE NUMBER          :  "<<setw(10)<<phone<<endl<<endl;
   cout<<setw(20)<<" "<<" YOUR ORDER            :  "<<setw(6)<<order<<endl<<endl;
   cout<<setw(20)<<" "<<" ORDER TIME            :  "<<setw(5)<<order_time<<endl<<endl;
   cout<<setw(20)<<" "<<" DELIVERY TIME         :  "<<setw(5)<<delivery_time<<endl<<endl;
   cout<<setw(20)<<" "<<" AMOUNT PAID           :  "<<setw(5)<<amount_paid<<endl<<endl;
   for(int g=0;g<80;g++) cout<<'*';
   cout<<endl<<setw(25)<<" ";

   cout<<"FOOD ORDER RECORD FOUND...!!"<<endl;
        }
        break;
}
```

Figure 4.5 Retrieve() function

4.2.6 Bill Generation Module Pseudocode

The Bill Generation option shown in Figure 4.6 is implemented by calling the bill() function which is called when option 6 is selected in the main menu. In this function the user is asked to enter the Order ID which is then used to generate the bill which has the order ID, Name of the customer the ordered items and the amount that is to be paid. The bill is generated automatically making it easier for both customer and owner of the restaurant.

```
void Order::bill(int addr,char k[])
{
int found=0,i;
char dummy[10];
i=addr;
file.open(datafile,ios::in);
cout<<setiosflags(ios::left);
//if(l==1)
do
{
        file.seekg(i*recsize*bucketsize,ios::beg);
        file.getline(dummy,5,'\n');
        if(strcmp(dummy,"####")==0)
                break;
        file.seekg(i*recsize*bucketsize,ios::beg);
        file.getline(order_no,10,'|');
        if(strcmp(order_no,k)==0)
        {
                found=1;

                unpack();
                clrscr();
                textcolor(15);
              textbackground(YELLOW);
           cout<<endl;
           cout<<setw(18)<<" "<<"THE BILL GENERATED FOR YOUR FOOD ORDER RECORD IS DISPLAYED"<<endl<<endl;
           for(int f=0;f<80;f++)   cout<<'*';   cout<<endl;
           cout<<setw(20)<<" "<<" ORDER ID               :  "<<setw(10)<<order_no<<endl<<endl;
           cout<<setw(20)<<" "<<" CUSTOMER NAME          :  "<<setw(10)<<cust_name<<endl<<endl;
           cout<<setw(20)<<" "<<" YOUR ORDER             :  "<<setw(6)<<order<<endl<<endl;
           cout<<setw(20)<<" "<<" AMOUNT PAID            :  "<<setw(5)<<amount_paid<<endl<<endl;
           for(int g=0;g<80;g++) cout<<'*';
           cout<<endl<<setw(25)<<" ";

           cout<<"FOOD ORDER RECORD FOUND...!!"<<endl;
                getch();
                break;
        }
```

Figure 4.6 Bill() Function

4.2.7 Hashing Pseudocode

There are many methods to compute the hash value of the key. These methods are called hash functions. The hash function used in this project is Summation and reminder method.

Summation and reminder method: The key digits are summed up to form an intermediate result. This result is then passed to the modulus function with the maximum size of the hash as 8. Thus, we can fold a key of any size into a smaller number.

In the hash function defined in Figure 4.7 we can see that we extract each character and then subtract 48 from it as the number is represented in the ASCII format.

```
int hash(char order_no[])
{
int i=0,sum=0,c,len;
len=strlen(order_no);
if(len%2==1)
    len++;
for(i=0;i<len;i+=2)
    sum=(sum+((order_no[i]+order_no[i+1])-96));
c=sum%max;
cout<<"\n\t\t    HASH KEY GENERATED IS: "<<c<<endl;
return c;
}
```

Figure 4.7 Hash() function

4.2.8 Collision Handling module

Collision occurs when the address corresponding to the computed hash value has already been occupied. The technique used to resolve collision is Linear probing. It is shown in Figure 4.8. If faced with a collision situation, we look for free buckets in the same hash locations or in the following hash locations until the first free space is found. This traversal of it going element by element at a time, is called linear probing. Once the end of hash table is reached (which is end of 2nd address because the total number of buckets considered is 3), wrap around concept is used to further check for free space from the beginning of the table till the first computed hashed value.

```
if(strcmp(dummy,"####")==0||strcmp(dummy,"$$$$")==0)
 {
        clrscr();
        textcolor(YELLOW);
        textbackground(BLACK);
        cout<<"\n\n\n\n";
        for(int t=0;t<80;t++) cout<<'~';
        for(int y=0;y<80;y++) cout<<'*';
        cout<<"\n\n\t\t\tCOLLISION HAS OCCURED....!!\n\n";
        cout<<"\t      HOME ADDRESS IS: "<<addr<<" & ACTUAL ADDRESS IS: "<<i<<"\n";
        file.seekp(i*recsize*bucketsize+(2*recsize),ios::beg);
        file<<buffer;
        textcolor(YELLOW);
        textbackground(BLACK);
        cout<<"\n\n\t\t   FOOD ORDER RECORD INSERTED SUCCESSFULLY....!!\n\n";
        for(int u=0;u<80;u++) cout<<'*';
        cout<<"\n";
        for(int f=0;f<80;f++) cout<<'~';
        getch();
        flag=1;
        break;
 }
```

Figure 4.8 Code for collision handling

4.3 Testing

Software Testing is the process used to help identify the correctness, completeness, security and quality of the developed computer software. Testing is the process of technical investigation and includes the process of executing a program with the intent of finding errors.

Unit Testing

The unit testing is the process of testing the part of the program to verify whether the program is working correct or not. In this part the main intention is to check the each and every inputs which we are inserting to our file. Here the validation concepts are used to check whether the program is taking the inputs in the correct format or not.

Functions (or features) are tested by feeding them input and examining the output. Functional testing ensures that the requirements are properly satisfied by the project. This type of testing is not concerned with how processing occurs, but rather, with the results of processing. During functional testing, Black Box Testing technique is used in which the internal logic of the system being tested is not known to the tester.

Table 4.1 Unit Testing Test Cases for menu

Case id	Description	Input data	Expected o/p	Actual o/p	Status
1	Opening a file to insert the data	Insert option is selected	File should be opened in append mode without any error messages	File opened in append mode	Pass
2	Validating order id	1	Accept the order ID and prompt for Customer name should be displayed	“Enter the customer name:”	Pass
3	Validating order id	1	Invalid order id and the message “Enter the order ID:” will be displayed	“Enter the order ID:”	Pass
4	Record Storing	-	It should pack all the fields and will be placed correctly to the data file	Data file will be updated	Pass
5	Closing a file	-	File should be closed without any error message	File is closed	Pass

Integration Testing

Integration testing is also taken as integration and testing this is the major testing process where the units are combined and tested. Its main objective is to verify whether the major parts of the program is working fine or not. This testing can be done by choosing the options in the program and by giving suitable inputs it is test.

Table 4.2 Integration testing for all modules

Case id	Description	Input data	Expected o/p	Actual o/p	Status
1.	To display the entered records of the data file	Enter the option 3 in the menu	Display the record one after the other	Display the record one after the other	pass
2.	To add the new records into the data file	Enter the option 1 in the menu	Display the record entry form	Display the record entry form	Pass
3.	To search for a particular record in the file	Enter the option 2 in the menu enter order ID:12343452	Record not found.	Record not found.	pass
4.	To search for a particular record in the file	Enter the option 2 in the menu and should enter the order ID:12345678	'Record found' and display contents of the searched record.	'Record found' and display contents of the searched record.	pass
5.	To delete a particular record in the file	Enter option 4 in the menu and enter order ID:12343452	Record not found.	Record not found.	pass
6.	To delete a particular record in the file	Enter the option 4 in menu and enter order ID:12345678	Delete the record and data file will be updated.	Delete the record and data file will be updated.	Pass
7.	To update or modify a particular record in the file	Enter the option 5 in the menu enter order ID:12343452	Record not found	Record not found	Pass
8.	To update a particular record in the file	Enter the option 5 in menu and enter order ID:12345678	'Record found' and delete it and data file will be updated.	'Record found' and delete it and data file will be updated	Pass
9.	To generate the bill	Enter the option 6 in menu and enter order ID:12345678	'Record found' and bill will be displayed on the screen.	'Record found' and bill will be displayed on the screen.	Pass
10.	To go back to previous screen	Enter the option 7 in the menu	Back to previous screen.	Back to previous screen.	pass

System Testing

System testing is defined as testing of a complete and fully integrated software product. This testing falls in black-box testing wherein knowledge of the inner design of the code is not a pre-requisite and is done by the testing team. System testing is done after integration testing is complete. System testing should test functional and non-functional requirements of the software.

Table 4.3 System Testing for FOMS

Case id	Description	Input data	Expected output	Actual output	Status
1	To display all records	Option 3 in the user menu	Display all records found on the file	Display all records found on the file	Pass
2	To search the record	Valid order ID = 12345678 (present) and invalid order ID = 12343445 (not present)	Record is displayed for valid and record not found for invalid	Record is displayed for valid and record not found for invalid	Pass
3	To delete the records	Valid order ID = 12345678 (present) and invalid order ID = 12343445 (not present)	Record is deleted for valid and record not found for invalid	Record is deleted for valid and record not found for invalid	Pass
4	To update or modify the records	Valid order ID = 12345678 (present) and invalid order ID = 12343445(not present)	Record is updated for valid and record not found for invalid	Record is updated for valid and record not found for invalid	Pass
5	To generate the bill	Valid order ID = 12345678 (present) and invalid order ID = 12343445(not present)	Bill Generated for valid and record not found for invalid	Bill Generated for valid and record not found for invalid	Pass

4.4 Discussion of Results

All the features provided in the project and its operations have been presented as screenshots.

4.4.1 Menu Options

The below Figure 4.9 is the first screen that is displayed to the user. It contains 2 options.

Figure 4.9 Menu Screen 1

Figure 4.10 is displayed after selecting "Food Order Record" option from the Menu Screen 1. It contains 7 different options as shown below.

Figure 4.10 Menu Screen 2

4.4.2 Displaying all records

The Figure 4.11 displays all the records entered.

Figure 4.11 Display of Records Screen

4.4.3 Record Insertion

The Figure 4.12 shows the page to insert a new record along with key validation.

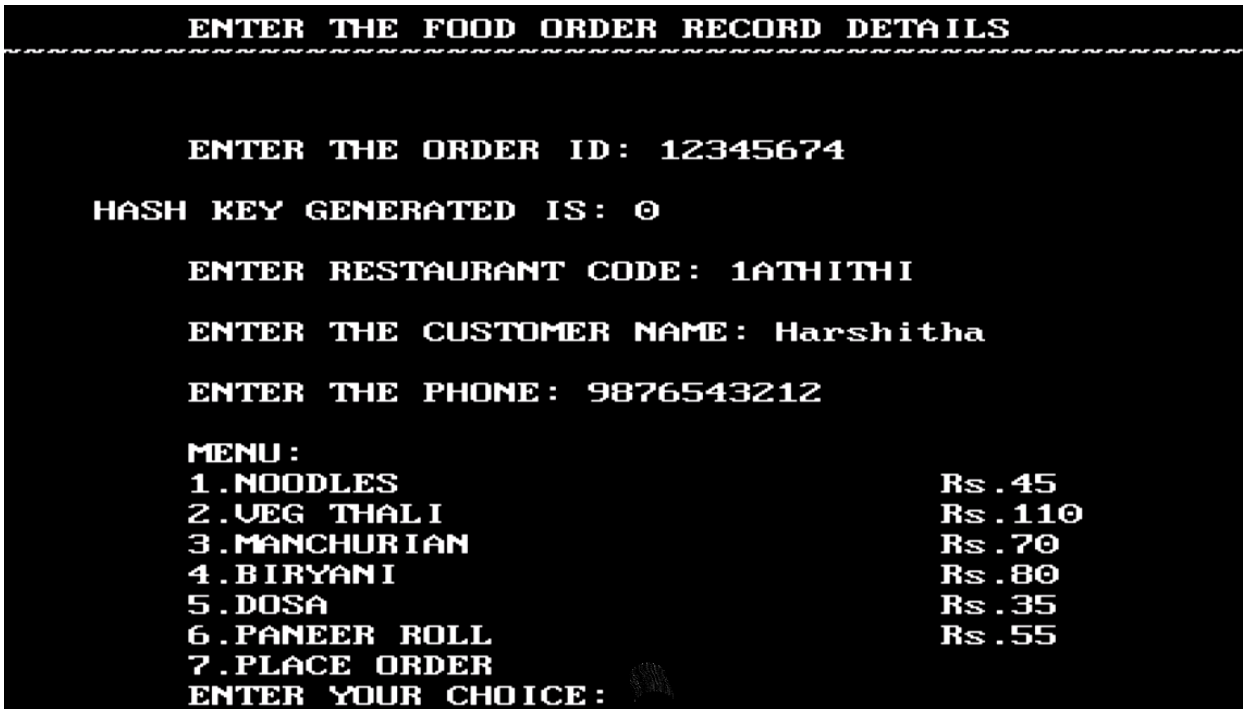

Figure 4.12 Insertion of a Record

4.4.4 Record Deletion

The Figure 4.13 displays the deletion feature where we enter the key to find and then delete the specific records.

Figure 4.13 Deletion of a Record

4.4.5 Searching for a Record

The Figure 4.14 depicts the page to search and retrieve the details of a specific record by Order ID.

Figure 4.14 Search for a Record

4.4.6 Record Modification

The Figure 4.15 displays record modification function in which we delete the existing record and then ask the user to enter the new order details.

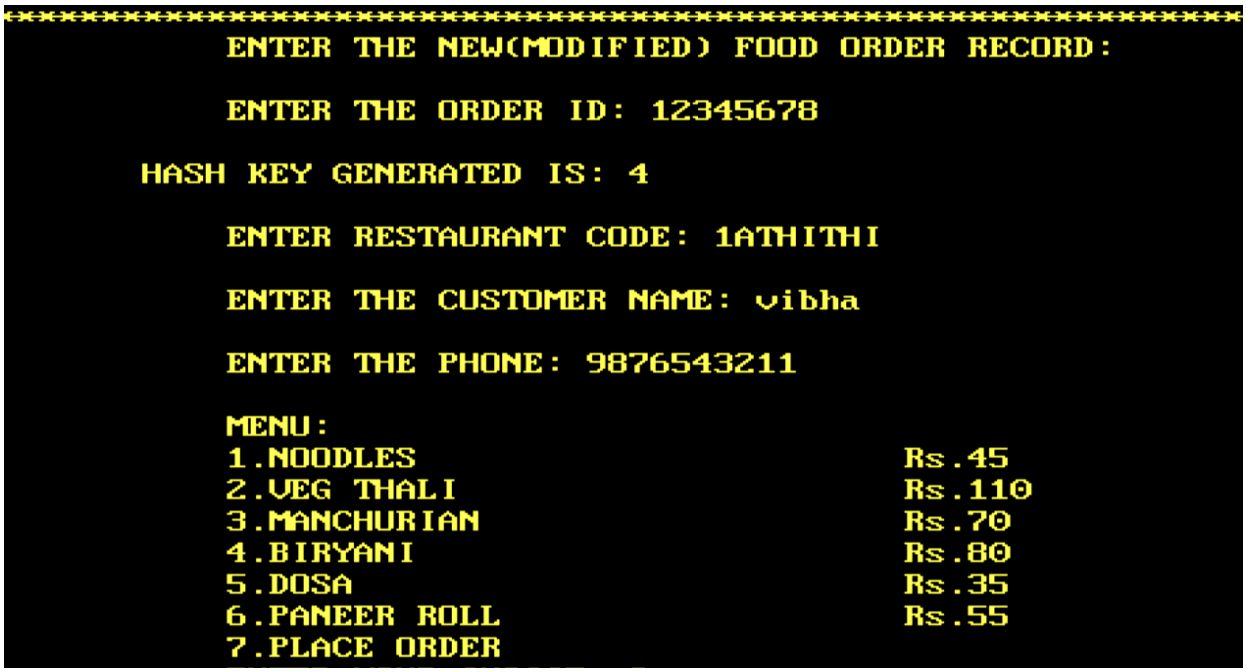

Figure 4.15 Record Modification

4.4.7 Bill Generation

The Figure 4.16 shows bill generation function in which the customer can see the bill by inserting Order ID.

Figure 4.16 Bill generation

4.4.8 File Contents

Data File

The Figure 4.17 shows the data file. It stores all the order details as entered by the user of the system. Here the records are stored in different buckets which can be visualized like columns and each row representing a hash address. All the fields are separated by a delimiter ('|'), but records don't have delimiters as they of fixed length already.

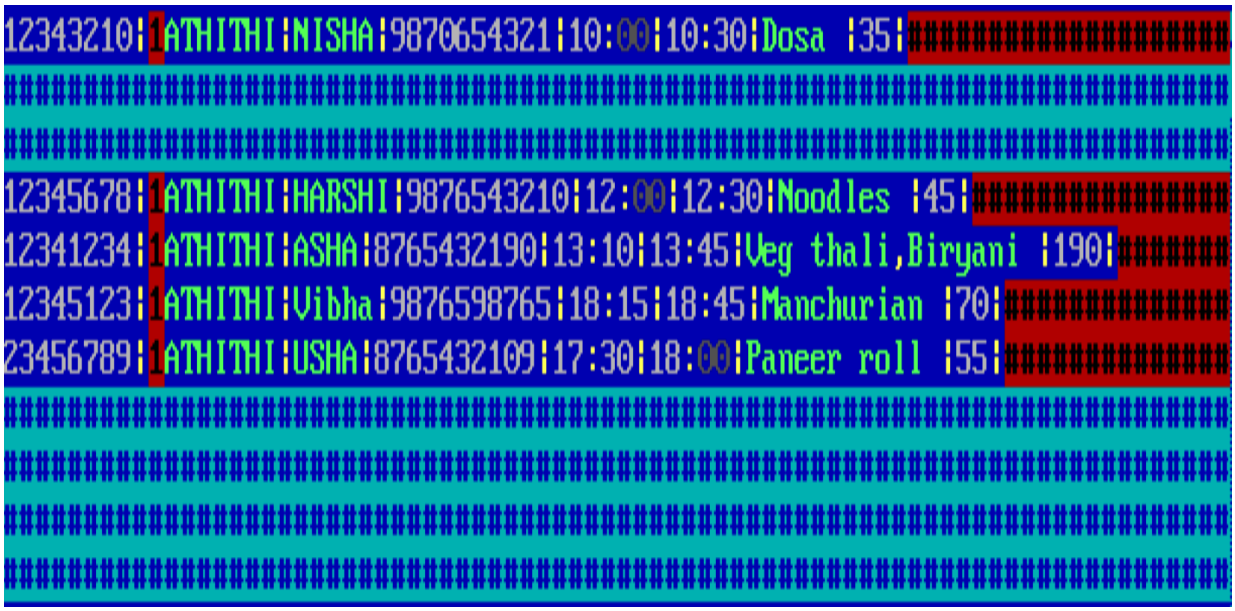

Figure 4.17 Data File Contents

Chapter 5

CONCLUSION AND FUTURE ENHANCEMENTS

The Food Order System has automated the existing manual system by the help of computerized equipment, so that the valuable data/information can be stored in a computer disk for a longer period with easy accessing and manipulation of the same. It focuses on providing the employees of a restaurant, the ability to adapt to the current trends with great flexibility in aspect of orders up keeping. The system has successfully been designed and implemented using hashing and buckets to access the records in a faster manner.

The system can also be used to search, delete, modify and display existing orders. The hashing implementation has provided faster and direct access to records, utilizing memory storage efficiently. The system is tested and re-tested with varying constraints to ensure its effectiveness and provide error free functionality to the end user.

Possible Future Enhancements are

- The system can be made cross platform to make it work online or on mobile devices in order to make the customer experience easier and quicker.
- The system can be made to calculate the total amount and display it as soon as the order is placed.
- The system can be made to add the order and delivery time from the network automatically.
- The project can use a database to store user accounts and passwords and provide a login screen at the beginning for user authentication.
- The project can be implemented using different file structures and indexing techniques to improve the performance.
- The project can also have options for pre-ordering before you reach the restaurant so that waiting time is reduced.
- Tracking orders helps tax filing and other administrative decisions easy.

REFERENCES

[1] File Structures: An Object-Oriented Approach in C++, PEARSON, 3rd Edition.

[2] www.geeksforgeeks.org

[3] K.R. Venugopal, K.G. Srinivas, P.M. Krishnaraj: File Structures Using C++, Tata McGraw-Hill, 2008.

[4] Scot Robert Ladd: C++ A/Cs and Algorithms, BPB Publications, 1993

[5] Raghu Ramakrishan and Johannes Gehrke: Database Management Systems, 3rd Edition, McGraw Hill, 2003

[6] https://www.geeksforgeeks.org/hashing-set-2-separate-chaining/

[7] https://www.tutorialspoint.com/cplusplus/cpp_files_streams.htm

[8] http://www.cppforschool.com/tutorial/Files1.html

[9] https://www.tutorialspoint.com/dbms/dbms_hashing.htm

[10] https://www.site.uottawa.ca/~nat/Courses/DFS-Course/DFS-Lecture-1/tsld012.htm