

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

JNANA SANGAMA, BELAGAVI – 590 018



A Mini Project Report on

Prison Management System using Hashing

*Submitted in partial fulfillment of the requirements as a part of the File Structures Lab of VI semester for the award of degree of **Bachelor of Engineering in Information Science and Engineering**, Visvesvaraya Technological University, Belagavi*

Submitted by

JEEVAN N Y
1RN20IS064

MAITHREYA T M
1RN20IS083

Faculty in-charge

Ms. Aruna U
Asst. Professor
Dept of ISE, RNSIT

Project Coordinator

Mrs. Vinutha G K
Asst. 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

2022-2023

RNS Institute of Technology

Channasandra, Dr. Vishnuvardhan Road, RR NagarPost,

Bengaluru – 560 098

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING



CERTIFICATE

This is to certify that the mini project report entitled **PRISON MANAGEMENT SYSTEM USING HASHING** has been successfully completed by **JEEVAN NY** bearing USN **1RN20IS064** and **MAITHREYA TM** bearing USN **1RN20IS083** 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 of *Bachelor of Engineering in Information Science and Engineering* under **Visvesvaraya Technological University, Belagavi** during academic year **2022-2023**. 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. Vinutha G K
Project Coordinator
Assistant Professor

Ms. Aruna U
Faculty In-charge
Assistant Professor

Dr. Suresh L
Professor and HOD

External Viva

Name of the Examiners

Signature with date

1. _____

2. _____

DECLARATION

We, **JEEVAN N Y (1RN20IS064)** and **MAITHREYA TM (1RN20IS083)** students of VI Semester BE, in Information Science and Engineering, RNS Institute of Technology hereby declare that the mini project entitled “**PRISON MANAGEMENT SYSTEM**” has been carried out by us and submitted in partial fulfillment of the requirements for the *VI Semester of degree of Bachelor of Engineering in Information Science and Engineering of Visvesvaraya Technological University, Belagavi* during academic year 2022-2023.

Place: Bengaluru

Date: 07/07/2023

JEEVAN N Y
1RN20IS064

MAITHREYA T M
1RN20IS083

ABSTRACT

The given code is a C++ program that manages prison and inmate details using hash tables. The program allows users to add, search, and delete prisons and inmates. It uses separate hash tables for prisons and inmates, with a hash function to calculate the hash value for each entry. The program defines two structures: "Prison" and "Inmate" which store the details of prisons and inmates, respectively. The program provides functions to save prison and inmate details to files, as well as functions to save hash values and keys to separate files. The main functionality of the program includes adding prisons and inmates, displaying all prisons and inmates, searching for a specific prison or inmate, and deleting prisons and inmates. When adding a prison or inmate, the program checks for unique IDs and maximum limits for linked lists or hash values. The program also handles the deletion of associated inmates when a prison is deleted. It automatically updates the details in the files after any modification to prisons or inmates. The main function initializes the hash table size and creates empty hash tables for prisons and inmates. It prompts the user to enter choices for various operations and executes the corresponding functions based on the input. Overall, this C++ program provides a basic framework for managing prison and inmate details using hash tables, allowing for efficient storage and retrieval of data.

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 Mini Project work.

We would like to express our thanks to our beloved Director **Dr. M K Venkatesh** for his support and inspired us towards the attainment of knowledge.

We would like to express our thanks to our Principal **Dr. Ramesh Babu H S** for his support and inspired us towards the attainment of knowledge.

We wish to place on record our words of gratitude to **Dr. Suresh L** Professor and Head of the Department, Information Science and Engineering, for being the enzyme, master mind behind my mini project work for guiding us and helping us out with the report.

We would like to express our profound and cordial gratitude to our faculty in- charge, **Ms. Aruna U**, Assistant Professor, Department of Information Science and Engineering for her valuable guidance, constructive comments and continuous encouragement throughout the mini project work.

We would like to express our sincere gratitude to **Mrs. Vinutha G K**, Assistant Professor, Department of Information Science and Engineering for her 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 us to carry out the project work.

And last, we would hereby acknowledge and thank our **Parents** who have been a source of inspiration and also instrumental in carrying out this mini project work.

JEEVAN N Y

1RN20IS064

MAITHREYA T M

1RN20IS083

TABLE OF CONTENTS

CERTIFICATE	
DECLARATION	i
ABSTRACT	ii
ACKNOWLEDGMENT	iii
TABLE OF CONTENTS	iv
LIST OF FIGURES	vi
LIST OF TABLES	vii
1 INTRODUCTION	01
1.1 Introduction to File Structures	01
1.1.1 History	01
1.1.2 About the File	02
1.1.3 Various kinds of storage of fields and records	02
1.1.4 Application of File Structures	05
2 SYSTEM ANALYSIS	06
2.1 Analysis of Application	06
2.2 Structure used to Store the Fields and Records	06
2.3 Operations Performed on a File	06
2.4 Indexing Used	08
3 SYSTEM DESIGN	09
3.1 Design of the Fields and Records	09
3.2 User Interface	10
3.2.1 Adding Inmate data	10
3.2.2 Remove Inmate data	10
3.2.3 Search Inmate data	11
3.2.4 Adding Prison data	11
3.2.5 Display Prison data	11
3.2.6 Remove Prison data	11
3.2.7 Search Prison data	12
4 IMPLEMENTATIONS	13
4.1 About C++	13
4.1.1 Classes and Objects	13
4.1.2 File Handling	13
4.2 Pseudo code	14

4.2.1 Structure Code	14
4.2.2 Hash Function	14
4.2.3 Function to Add Inmate	14
4.2.4 Function to Add Prison	15
4.2.5 Function to delete Inmate	16
4.2.6 Function to delete Prison	17
4.2.7 Function to display all Prison	18
4.2.8 Function to display all Inmate	18
4.2.9 Function to search Prison	18
4.2.10 Function to search Inmate	19
4.3 Testing	20
4.3.1 Unit Testing	20
4.3.2 Integration Testing	21
4.3.3 System Testing	22
4.3.4 Acceptance Testing	23
4.4 Discussion of Results	24
4.4.1 Menu Options	24
4.4.2 Add Prison record data	24
4.4.3 Add Inmate record data	25
4.4.4 Search Prison record	25
4.4.5 Search Inmate record	26
4.4.6 Delete Prison record	26
4.4.7 Delete Inmate record	26
4.4.8 Display all prison records	27
4.4.9 Display all Inmate records	27
4.4.10 Output Prison data file	28
4.4.11 Output Inmate data file	28
4.4.12 Output Prison hash file	28
4.4.13 Output Inmate hash file	28
5 CONCLUSION AND FUTURE ENHANCEMENTS	30
REFERENCES	31

LIST OF FIGURES

Figure No	Descriptions	Page No
Figure 1.1	Four methods for field structures	03
Figure 1.2	Predictable number of Bytes and Fields	05
Figure 1.3	Length Indicator, Index and Record Delimiters	05
Figure 3.1	Class Product	09
Figure 3.2	Output Menu Screen	10
Figure 4.1	Starting Menu Screen	24
Figure 4.2	Add a new Prison record	24
Figure 4.3	Add a new Inmate record	25
Figure 4.4	Search a particular Prison record	25
Figure 4.5	Search a particular Inmate record	26
Figure 4.6	Delete Prison record	26
Figure 4.7	Delete Inmate record	26
Figure 4.8	Display all Prison records	27
Figure 4.9	Display all Inmate records	27
Figure 4.10	Output of Prison data file	28
Figure 4.11	Output of Inmate data file	28
Figure 4.12	Output of Prison hash file	28
Figure 4.13	Output of Inmate hash file	28

LIST OF TABLES

Table No	Descriptions	Page
Table 4.1	Unit Test Case for all modules	20
Table 4.2	Integration Test Case for all modules	21
Table 4.3	System Test Case for all modules	22
Table 4.4	Acceptance Testing	23

Chapter 1

INTRODUCTION

The Prison management system plays a vital role in organizations by efficiently managing Prison and ensuring smooth workflow continuity. One crucial aspect of this system is the use of hashing techniques, which enhance the security and efficiency of the management process. Hashing is a cryptographic method that converts data fixed-length string of characters, known as a hash value.

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 applications 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 an application 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 applications.

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.

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 could 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

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

Field Structures

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

- Force the fields into a predictable length.
- Begin each field with a length indicator.
- Place a delimiter at the end of each field to separate it from the next field
- 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 1 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

The end of the field can be stored by counting 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. The fields are referred as length-based.

Method 3: Separate the Fields with Delimiters

The identity of fields can be preserved by separating them with delimiters. It is needed to 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.

a)	<table><tr><td>Ames</td><td>Mary</td><td>123 Maple</td><td>Stillwater</td><td>OK74075</td></tr><tr><td>Mason</td><td>Alan</td><td>90 Eastgate</td><td>Ada</td><td>OK74820</td></tr></table>	Ames	Mary	123 Maple	Stillwater	OK74075	Mason	Alan	90 Eastgate	Ada	OK74820
Ames	Mary	123 Maple	Stillwater	OK74075							
Mason	Alan	90 Eastgate	Ada	OK74820							
b)	<table><tr><td>04Ames04Mary09123 Maple10Stillwater02OK0574075</td></tr><tr><td>05Mason04Alan1190 Eastgate03Ada02OK0574820</td></tr></table>	04Ames04Mary09123 Maple10Stillwater02OK0574075	05Mason04Alan1190 Eastgate03Ada02OK0574820								
04Ames04Mary09123 Maple10Stillwater02OK0574075											
05Mason04Alan1190 Eastgate03Ada02OK0574820											
c)	<table><tr><td>Ames Mary 123 Maple Stillwater OK74075 </td></tr><tr><td>Mason Alan 90 Eastgate Ada OK74820 </td></tr></table>	Ames Mary 123 Maple Stillwater OK74075	Mason Alan 90 Eastgate Ada OK74820								
Ames Mary 123 Maple Stillwater OK74075											
Mason Alan 90 Eastgate Ada OK74820											
d)	<table><tr><td>Last=Ames;first=Mary;address=123 Maple;city=Stillwater;state=OK;zip=74075;</td></tr></table>	Last=Ames;first=Mary;address=123 Maple;city=Stillwater;state=OK;zip=74075;									
Last=Ames;first=Mary;address=123 Maple;city=Stillwater;state=OK;zip=74075;											

Figure 1.1 Four methods for field structures

Record Structures

The five most often used methods for organizing records are:

- Require the records to be predictable number of bytes in length.
- Require the records to be predictable number of fields in length.
- 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 modulo 6.

Method 3: Begin Each Record with a Length Indicator

Communication of the length of records by beginning each record with a field 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

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, use a # character as the record delimiter.

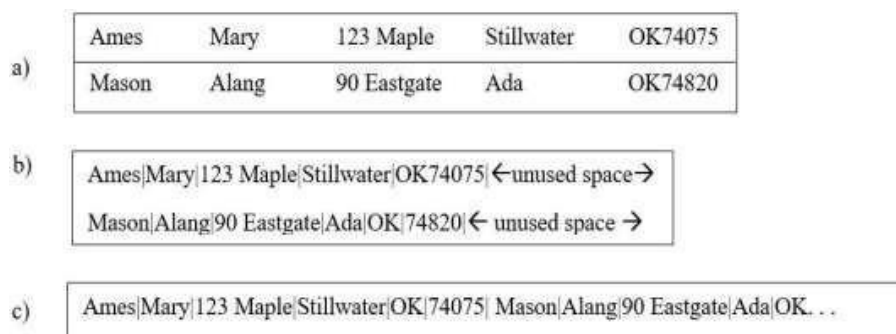


Figure 1.2 Making Records Predictable number of Bytes and Fields



Figure 1.3 Using Length Indicator, Index and Record Delimiters

1.1.4 Application of File Structure

Relative to other parts of a computer, disks are slow. 1 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. Tension between a disk's relatively slow access time and its enormous, non-volatile capacity.

Chapter 2

SYSTEM ANALYSIS

2.1 Analysis of Application

The application for a Prison Management System utilizes hashing techniques to enhance its efficiency and security. Hashing is a crucial component of this system, enabling the efficient storage and retrieval of the record.

2.2 Structure used to Store the Fields and Records Storing Fields

2.2.1 Fixing the Length of Fields:

In Prison Management system, it's crucial to ensure the integrity and consistency of data stored within the system. One important aspect is fixing the length of fields to maintain uniformity and prevent data corruption or discrepancies. The hashing technique can be employed to achieve this goal. Separating the Fields with Delimiters. We preserve the identity of fields by separating them with delimiters. We have chosen the comma (,) as the delimiter here.

2.2.2 Storing Records

Making Records a Predictable Number of Fields: In this system, have a fixed number of fields, each with a maximum length, that combine to make a data record. Fixing the number of fields in a record does not imply that the size of fields in the record is fixed.

2.3 Operations Performed on a File

2.3.1 Insertion based on inmate id

Compute the hash value for the inmate ID. The hash function should map the inmate ID to a unique index in the hash table. Check if the computed hash value is already occupied in the hash table. If the index is empty, proceed to step 3. Otherwise, handle the collision using an appropriate collision resolution technique.

2.3.2 Searching based on Inmate id

Create a hash table to store the Music records. Each record will consist of an Inmate ID and associated Inmate details. Initialize the hash table with an appropriate size to accommodate the expected number of Inmate. The size should be chosen based on the anticipated load factor to ensure efficient performance. Define a hash function that takes an Inmate ID as input and returns the hash value. The hash function should distribute the keys uniformly across the hash table. When adding Inmate to the system, calculate the hash value using the hash function and insert the record at the corresponding index in the hash table. To search for an inmate based on their ID, calculate the hash value of the ID using the same hash function. Use the calculated hash value to access the corresponding index in the hash table. If the record at the index matches the Inmate ID being searched for, then the Inmate has been found. In this case, an appropriate collision resolution technique, such as chaining, to handle collisions and search for the Music in the collided entries.

2.3.3 Deletion based on Inmate id

Calculate the hash value of the given Inmate ID. Use the hash value to determine the index or bucket in the hash table where the Inmate record may be stored. Traverse the hash Map or any other data structure associated with the bucket to find the Inmate record with the matching ID. If the Inmate record is found, remove it from the data structure. If the data structure becomes empty after deletion, update the reference in the hash table to indicate an empty file. If the Inmate record is not found, it means that the Inmate does not exist in the system. Handle any necessary error or notification if the Inmate ID is not found.

2.4 Indexing Used

Hashing

Hashing with collision avoidance techniques involves several steps to ensure that collisions are minimized or handled effectively. Here are the general steps involved:

Define the hash function, choose or design a hash function that converts input data into a fixed-size hash value. The hash function should have good distribution properties to minimize collisions. Now determine the hash table size, decide on the size of the hash table, which is typically a prime number. The size should be large enough to accommodate the expected number of elements without causing excessive collisions.

Initialize the hash table, create an empty hash table with the determined size. This table will store the hashed values and associated data. Hashing with chaining (Separate chaining), In this collision avoidance technique, each slot of the hash table contains a linked list or any other data structure to store multiple values that hash to the same location. When inserting an element, compute its hash value and place it in the corresponding slot. If there is already an element present, add it to the linked list or other data structure at that location.

Handle retrieval and deletion, when retrieving or deleting an element, apply the same hash function to locate the slot in the hash table. In chaining, search the linked list at that location for the desired element. Similarly in open addressing, the probing sequence is used to locate the element. Resize the hash table, if necessary, if the number of elements in the hash table exceeds a certain threshold (e.g., load factor), consider resizing the hash table to maintain an efficient balance between space and time complexity.

This typically involves creating a new larger hash table and rehashing all the elements into it. These steps provide a general outline of the process involved in hashing with collision avoidance techniques. The specific implementation details may vary depending on the chosen technique and programming language.

Chapter 3

SYSTEM DESIGN

3.1 Design of the Fields and Records

In the design that has been implemented in the project, we have used two structures called Music and Artists respectively. As our project uses a hash table to store the records, we have used a linked list at each index of the hash table.

Currently we have used 10 indices to the hash table so each index is associated with a linked list, and each data record which will be stored will be stored in the form of a node of the linked list associated to the hash table. So, the structure Music will have the Music details that is first name, last name and the count. Struct Artist is used to implement the artist record data.

```
// Structure for prison details
struct Prison
{
    int id;
    std::string name;
    std::string location;
};

// Structure for inmate details
struct Inmate
{
    int id;
    std::string name;
    int prisonId;
};
```

Figure 3.1 Structure of Prison and Inmate

3.2 User Interface

The User Interface or UI refers to the interface between the application 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.

```
Enter the hash table size: 30
*****
          WELCOME TO PRISON MANAGEMENT
*****
          MAIN MENU
*****
1 . Add Prison
2 . Add Inmate
3 . Search Prison
4 . Search Inmate
5 . Delete Prison
6 . Delete Inmate
7 . Display Prisons
8 . Display Inmates
0 . Exit
-----
Enter your choice: █
```

Figure 3.2 Menu Screen

3.2.1 Adding Inmate data

The user interface for adding Inmate Record data in a Prison Management System using hashing provides a straightforward and efficient way for administrators or authorized personnel to input and store information related to Inmate data. The interface typically includes a form or fields where users can enter essential details about the inmate's name, Inmate's ID and Location where the Inmate was kept.

3.2.2 Remove Inmate data

The user interface for removing Inmate record data in a Prison management system that utilizes hashing is designed to provide a streamlined and efficient experience for the system administrator or authorized personnel. Overall, it aims to simplify and streamline the process, ensuring that authorized users can efficiently manage and maintain the Inmate database by securely removing relevant entries when necessary.

3.2.3 Search Inmate data

The user interface for searching Inmate data in a Prison Management System that utilizes hashing involves a streamlined and efficient process. The interface is designed to provide users with a user-friendly and intuitive experience while retrieving the Inmate data for specific Music. Over all it is designed to provide a seamless and efficient experience, enabling users to quickly retrieve and access the desired information while maintaining data integrity.

3.2.4 Adding Prison data

The user interface for adding Prison Record data in a Prison Management System using hashing provides a straightforward and efficient way for administrators or authorized personnel to input and store information related to Inmate data. The interface typically includes a form or fields where users can enter essential details about then artist name, Prison 's ID and Inmate ID are linked to the Prison.

3.2.5 Display Prison data

The user interface for the Displays the Prison data of a Prison management system using hashing is designed to provide a clear and organized representation of the stored data. The interface typically consists of a table-like structure with columns and rows. Each row represents a bucket or slot in the hash table, while the columns display relevant information about that particular Prison.

3.2.6 Remove Prison data

The user interface for removing Prison record data in a Prison management system that utilizes hashing is designed to provide a streamlined and efficient experience for the system administrator or authorized personnel. Overall, it aims to simplify and streamline the process, ensuring that authorized users can efficiently manage and maintain the Prison database by securely removing relevant entries when necessary.

3.2.7 Search Prison data

The user interface for searching Prison Artist data in a Prison Management System that utilizes hashing involves a streamlined and efficient process. The interface is designed to provide users with a user-friendly and intuitive experience while retrieving that particular Prison data for specific Prison ID. Over all it is designed to provide a seamless and efficient experience, enabling users to quickly retrieve and access the desired information while maintaining data integrity.

Chapter 4

IMPLEMENTATION

Implementation is the process of: defining how the system should be built, ensuring that it is operational 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++

C++ is a general-purpose programming language. It has imperative, object – oriented and generic programming features, also providing facilities for low level.

4.1.1 Classes and Objects

The product file is declared as class with the Inmate ID, Prison name as its data members and read (), search (), display (), display all data (), remove() as class methods. 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. Each an instance of the class type index. The data stored in file according to hash() which returns the address at which the corresponding data record is stored in 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 parameter.

4.2 Pseudocode:

4.2.1 Structures:

```
// Structure for Prison details to store prison record details
struct Prison
{
    int id;
    std::string name;
    std::string location;
};

// Structure for inmate details to store inmate details
struct Inmate
{
    int id;
    std::string name;
    int PrisonId;
};
```

4.2.2 Hash Function

```
int hashFunction(int key, int tableSize)
{
    std::hash<int> hasher;
    return hasher(key) % tableSize;
}
```

4.2.3 Function to Add Inmate

```
void addInmate(std::unordered_map<int, std::vector<Inmate>>
&inmateHashTable, std::unordered_map<int, int> &inmateHashKey, int
tableSize, const std::unordered_map<int, std::vector<Prison>>
&prisonHashTable, std::unordered_map<int, int> &prisonHashKey)
{
    Inmate inmate;
    std::cout << "Enter Inmate ID: ";
    std::cin >> inmate.id;
    // Check if the inmate ID already exists
    if (inmateHashKey.count(inmate.id) > 0)
    {
        std::cout << "Inmate ID already exists. Please enter a unique
ID." << std::endl;
```

```

        return;}
    std::cout << "Enter Inmate Name: ";
    std::cin.ignore();
    std::getline(std::cin, inmate.name);

    // Check if the entered prison ID exists
    int prisonId;
    std::cout << "Enter Prison ID: ";
    std::cin >> prisonId;
    if (prisonHashKey.count(prisonId) == 0)
    {
        std::cout << "Prison with ID " << prisonId << " does not
exist." << std::endl;
        return;}
    inmate.prisonId = prisonId;

    int hashValue = hashFunction(inmate.id, tableSize);

    // Check if the linked list length has reached the maximum limit
    if (inmateHashTable[hashValue].size() >= 2)
    {
        std::cout << "Cannot add inmate. Maximum limit reached for the
hash value." << std::endl;
        return; }

    inmateHashTable[hashValue].push_back(inmate);
    inmateHashKey[inmate.id] = hashValue;

    // Save the details to the files
    saveInmateDetails(inmateHashTable);
    saveInmateHashDetails(inmateHashKey);
}

```

4.2.4 Function to add new Prison

```

void addPrison(std::unordered_map<int, std::vector<Prison>>
&prisonHashTable, std::unordered_map<int, int> &prisonHashKey, int
tableSize)
{
    Prison;
    std::cout << "Enter Prison ID: ";
    std::cin >> prison.id;
    // Check if the prison ID already exists
    if (prisonHashKey.count(prison.id) > 0)
    {
        std::cout << "Prison ID already exists. Please enter a unique
ID." << std::endl;
        return;
    }
}

```



```

    }
    std::cout << "Enter Prison Name: ";
    std::cin.ignore();
    std::getline(std::cin, prison.name);
    std::cout << "Enter Prison Location: ";
    std::getline(std::cin, prison.location);

    int hashValue = hashFunction(prison.id, tableSize);
    // Check if the linked list length has reached the maximum limit
    if (prisonHashTable[hashValue].size() >= 2)
    {
        std::cout << "Cannot add prison. Maximum limit reached for
linked list." << std::endl;
        return;
    }
    prisonHashTable[hashValue].push_back(prison);
    prisonHashKey[prison.id] = hashValue;

    // Save the details to the files
    savePrisonDetails(prisonHashTable);
    savePrisonHashDetails(prisonHashKey);
}

```

4.2.5 Function to delete Inmate record

```

void deleteInmate(std::unordered_map<int, std::vector<Inmate>>
&inmateHashTable, std::unordered_map<int, int> &inmateHashKey)
{
    int inmateId;
    std::cout << "Enter Inmate ID to delete: ";
    std::cin >> inmateId;
    auto it = inmateHashKey.find(inmateId);
    if (it != inmateHashKey.end())
    {
        int hashValue = it->second;
        std::vector<Inmate> &inmates = inmateHashTable[hashValue];
        for (auto iter = inmates.begin(); iter != inmates.end();
++iter){
            if (iter->id == inmateId)
            {
                inmates.erase(iter);
                inmateHashKey.erase(it);
            }
            std::cout << "Inmate deleted successfully." << std::endl;
            // Save the updated details to the files
            saveInmateDetails(inmateHashTable);
        }
    }
}

```

```

        saveInmateHashDetails(inmateHashKey);
        return;
    }}
}
std::cout << "Inmate not found." << std::endl;
}

```

4.2.6 Function to delete a Prison and Associate record

```

void deletePrison(std::unordered_map<int, std::vector<Prison>>
&prisonHashTable, std::unordered_map<int, int> &prisonHashKey,
std::unordered_map<int, std::vector<Inmate>> &inmateHashTable,
std::unordered_map<int, int> &inmateHashKey)
{
    int prisonId;
    std::cout << "Enter Prison ID to delete: ";
    std::cin >> prisonId;

    auto it = prisonHashKey.find(prisonId);
    if (it != prisonHashKey.end())
    {
        int hashValue = it->second;
        std::vector<Prison> &prisons = prisonHashTable[hashValue];
        for (auto iter = prisons.begin(); iter != prisons.end();
++iter)
        {
            if (iter->id == prisonId)
            {
                prisons.erase(iter);
                prisonHashKey.erase(it);
                std::cout << "Prison deleted successfully." << std::endl;
                // Delete associated inmates
                deleteAssociatedInmates(inmateHashTable, inmateHashKey, prisonId);
                // Save the updated details to the files
                savePrisonDetails(prisonHashTable);
                savePrisonHashDetails(prisonHashKey);
                saveInmateDetails(inmateHashTable);
                saveInmateHashDetails(inmateHashKey);
                return;
            }
        }
    }
    std::cout << "Prison not found." << std::endl; }

```

4.2.7 Function to display Prison File

```
void displayPrisons(const std::unordered_map<int,
std::vector<Prison>> &prisonHashTable)
{
    // Print table header
    std::cout << std::left << std::setw(12) << "Prison ID"
               << std::setw(20) << "Prison Name"
               << std::setw(20) << "Prison Location" << std::endl;
    // Print table rows
    for (const auto &entry : prisonHashTable)
    {
        for (const Prison &prison : entry.second)
        {
            std::cout << std::left << std::setw(12) << prison.id
                      << std::setw(20) << prison.name << std::setw(20) <<
prison.location << std::endl;
        }
    }
}
```

4.2.8 Function to display Inmate File

```
void displayInmates(const std::unordered_map<int, std::vector<Inmate>>
&inmateHashTable)
{
    std::cout << std::left << std::setw(12) << "Inmate ID" << std::setw(20)
<< "Inmate Name" << std::setw(20) << "Prison ID" << std::endl;
    for (const auto &entry : inmateHashTable)
    {
        for (const Inmate &inmate : entry.second)
        {
            std::cout << std::left << std::setw(12) << inmate.id
                      << std::setw(20) << inmate.name << std::setw(20) <<
inmate.prisonId << std::endl;
        }
    }
}
```

4.2.9 Function to search a Prison record

```
void searchPrison(const std::unordered_map<int, std::vector<Prison>>
&prisonHashTable, const std::unordered_map<int, int> &prisonHashKey,
int tableSize)
{
    int prisonId;
    std::cout << "Enter Prison ID to search: ";
    std::cin >> prisonId;
```

```
    auto it = prisonHashKey.find(prisonId);
    if (it != prisonHashKey.end())
    {
        int hashValue = it->second;
        const std::vector<Prison> &prisons =
prisonHashTable.at(hashValue);
        for (const Prison &prison : prisons)
        {
            if (prison.id == prisonId)
            {
                std::cout << "Prison ID: " << prison.id << std::endl;
                std::cout << "Prison Name: " << prison.name;
                std::cout << "Prison Location: " << prison.location;
                return;
            }
        }
    }

    std::cout << "Prison not found." << std::endl;
}
```

4.2.10 Function to search an Inmate

```
void searchInmate(const std::unordered_map<int, std::vector<Inmate>>
&inmateHashTable, const std::unordered_map<int, int> &inmateHashKey,
int tableSize)
{
    int inmateId;
    std::cout << "Enter Inmate ID to search: ";
    std::cin >> inmateId;
    auto it = inmateHashKey.find(inmateId);
    if (it != inmateHashKey.end())
    {
        int hashValue = it->second;
        const std::vector<Inmate> &inmates =
inmateHashTable.at(hashValue);
        for (const Inmate &inmate : inmates)
        {
            if (inmate.id == inmateId)
            {
                std::cout << "Inmate ID: " << inmate.id;
                std::cout << "Inmate Name: " << inmate.name;
                std::cout << "Prison ID: " << inmate.prisonId;
                return;
            }
        }
    }

    std::cout << "Inmate not found." << std::endl;
}
```

4.3 Testing

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 or application with the intent of finding errors. Software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test.

4.3.1 Unit Testing

Unit testing is a level of software testing where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software.

Case id	Description	Input data	Expected Output	Actual Output	Status
1	On selecting the choice, the particular operation is allowed to access	choice	Particular operation access will be allowed	Access allowed	Pass
2	Add PRISON, Enter PRISONID, name and location	PRISONID:1 name: Bgl_prison location: Bglr	PRISON Record is saved in file, and also is hashed.	PRISON Record is saved in file, and also is hashed.	Pass
3	Add Inmate, Enter InmateID, name and PRISONID	InmateID:11 name: Amar PRISONID:1	Inmate Record is saved in file, and also is hashed	Inmate Record is saved in file, and also is hashed	Pass
4	Search PRISON	PRISONID:1	It displays record with PRISONID:1	It displays record with PRISONID:1	Pass
5	Display PRISON records	No input	It should unpack and display PRISON records	Displays PRISON record	Pass
6	Delete PRISON	Enter PRISONID	Record will be deleted updated file will be displayed	Record will be deleted updated file will be displayed	Pass

Table 4.1 Unit testing for all modules

4.3.2 Integration Testing

Integration Testing is a level of software testing where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units. Test drivers and test stubs are used to assist in Integration Testing.

Case ID	Description	Input	Expected Output	Actual Output	Status
1	Add an Inmate and a PRISON associated with that Prison	Add an Inmate, Add a PRISON with the same Prison ID	Inmate and PRISON should be added successfully	Inmate and PRISON should be added successfully	Pass
2	Add a PRISON with a non-existing Inmate ID	Add a PRISON with an Inmate ID that does not exist	Inmate with ID [Inmate ID] does not exist error message	Inmate with ID [Inmate ID] does not exist error message	Pass
3	Delete an Inmate and associated PRISONs	Add an Inmate, add multiple PRISONs associated with Inmate	Delete the Inmate and associated PRISONs successfully	Delete the Inmate and associated PRISONs successfully	Pass
4	Search for a PRISON associated with a specific Inmate	Add an Inmate, add multiple PRISONs associated with Inmate	Search for a PRISON using the Inmate ID should display the PRISON details	Search for a PRISON using the Inmate ID should display the PRISON details	Pass
5	Search for an Inmate after deleting associated PRISONs	Add an Inmate, add multiple PRISONs associated with Inmate	Delete the associated PRISONs, Search for the Inmate should not display any associated PRISONs	Delete the associated PRISONs, Search for the Inmate should not display any associated PRISONs	Pass
6	Add multiple Inmates and PRISONs and display all	Add multiple Inmates, Add multiple PRISONs	Display all Inmates should show all added Inmates	Display all Inmates should show all added Inmates	Pass

Table 4.2 Integration test case for all module

4.3.3 System Testing

System Testing is a level of the software testing where a complete and integrated software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements. The application is run to check if all the modules (functions) can be executed concurrently, if each return correct results of the operations performed by them, and if the data and index files are left in consistent states by each module, as shown in Table 4.8.

Case ID	Description	Input Data	Expected Output	Actual Output	Status
1	Add multiple Inmates and PRISONs to the database	Add several Inmates and PRISONs to the database	Inmates and PRISONs should be added successfully	Inmates and PRISONs should be added successfully	Pass
2	Search for an Inmate that exists in the database	Inmate ID to search: Existing Inmate ID	Inmate details (ID, name, location) should be displayed	Inmate details (ID, name, location) should be displayed	Pass
3	Search for an Inmate that does not exist in the database	Inmate ID to search: Non-existing Inmate ID	"Inmate not found" message should be displayed	"Inmate not found" message should be displayed	Pass
4	Search for a PRISON that exists in the database	PRISON ID to search: Existing PRISON ID	PRISON details (ID, name, Inmate ID) should be displayed	PRISON details (ID, name, Inmate ID) should be displayed	Pass
5	Search for a PRISON that does not exist in the database	PRISON ID to search: Non-existing PRISON ID	"PRISON not found" message should be displayed	"PRISON not found" message should be displayed	Pass
6	Delete an existing Inmate and associated PRISONs from the database	Inmate ID to delete: Existing Inmate ID	Inmate and associated PRISONs should be deleted successfully	Inmate and associated PRISONs should be deleted successfully	Pass

Table 4.3 System test case for all modules

4.3.4 Acceptance Testing

Acceptance Testing is a level of software testing where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery. It is performed by:

- Internal Acceptance Testing (Also known as Alpha Testing) is performed by members of the organization that developed the software but who are not directly involved in the project (Development or Testing).
- External Acceptance Testing is performed by people who are not employees of the organization that developed the software.
- Customer Acceptance Testing is performed by the customers of the organization that developed the software.
- User Acceptance Testing (Also known as Beta Testing) is performed by the user.

Case ID	Description	Input	Expected output	Actual Output	Status
1	Add a new Inmate	Inmate ID: 11, Inmate Name: John Doe, prison ID: 1	The Inmate should be added successfully. File "Inmates.txt" updated.	The Inmate should be added successfully. "Inmates.txt" updated.	Pass
2	Add a new PRISON	PRISON ID: 1, PRISON Name: Mysr_prison location: Mysuru	The PRISON should be added successfully. File "PRISONS.txt" updated.	The PRISON should be added successfully. "PRISONS.txt" updated.	Pass
3	Search for an Inmate	Inmate ID to search: 1	Inmate details (ID, name, PrisonID) should be displayed.	Inmate details should be displayed.	Pass
4	Search for a PRISON	PRISON ID to search: 1	PRISON details (ID, name, location) should be displayed.	PRISON details should be displayed.	Pass

Table 4.3 Acceptance test case for all details

4.4 Discussion of Results

All the menu options provided in the application and its operations have been presented in as screen shots.

4.4.1 Starting Menu Options

```
Enter the hash table size: 30
*****
          WELCOME TO PRISON MANAGEMENT
*****
          MAIN MENU
*****
1 . Add Prison
2 . Add Inmate
3 . Search Prison
4 . Search Inmate
5 . Delete Prison
6 . Delete Inmate
7 . Display Prisons
8 . Display Inmates
0 . Exit
-----
Enter your choice: █
```

Figure 4.1 Starting Menu Screen.

4.4.2 Add Prison record

```
*****
          WELCOME TO PRISON MANAGEMENT
*****
          MAIN MENU
*****
1 . Add Prison
2 . Add Inmate
3 . Search Prison
4 . Search Inmate
5 . Delete Prison
6 . Delete Inmate
7 . Display Prisons
8 . Display Inmates
0 . Exit
-----
Enter your choice: 1
Enter Prison ID: 101
Enter Prison Name: Bangalore prison
Enter Prison Location: Bangalore
```

Figure 4.2: Adding a new Prison record

4.4.3 Add Inmate record

```
*****
WELCOME TO PRISON MANAGEMENT
*****
MAIN MENU
*****
1 . Add Prison
2 . Add Inmate
3 . Search Prison
4 . Search Inmate
5 . Delete Prison
6 . Delete Inmate
7 . Display Prisons
8 . Display Inmates
0 . Exit
-----
Enter your choice: 2
Error opening file: inmate_hash.txt
Enter Inmate ID: 201
Enter Inmate Name: Amardeep
Enter Prison ID: 101
```

Figure 4.3 Adding a new Inmate record

4.4.4 Search Prison Record

```
*****
WELCOME TO PRISON MANAGEMENT
*****
MAIN MENU
*****
1 . Add Prison
2 . Add Inmate
3 . Search Prison
4 . Search Inmate
5 . Delete Prison
6 . Delete Inmate
7 . Display Prisons
8 . Display Inmates
0 . Exit
-----
Enter your choice: 3
Enter Prison ID to search: 101
Prison ID: 101
Prison Name: Bangalore_prison
Prison Location: Bangalore
```

Figure 4.4 Search a particular Prison record

4.4.5 Search an Inmate Record

```
*****
WELCOME TO PRISON MANAGEMENT
*****
MAIN MENU
*****
1 . Add Prison
2 . Add Inmate
3 . Search Prison
4 . Search Inmate
5 . Delete Prison
6 . Delete Inmate
7 . Display Prisons
8 . Display Inmates
0 . Exit
-----
Enter your choice: 4
Enter Inmate ID to search: 201
Inmate ID: 201
Inmate Name: Amardeep
Prison ID: 101
```

Figure 4.5 Search a particular Inmate record

4.4.6 Delete Prison Record

```
*****
WELCOME TO PRISON MANAGEMENT
*****
MAIN MENU
*****
1 . Add Prison
2 . Add Inmate
3 . Search Prison
4 . Search Inmate
5 . Delete Prison
6 . Delete Inmate
7 . Display Prisons
8 . Display Inmates
0 . Exit
-----
Enter your choice: 5
Enter Prison ID to delete: 101
Prison deleted successfully.
```

Figure 4.6 Delete a Prison record

4.4.7 Delete Inmate Record

```
*****
WELCOME TO PRISON MANAGEMENT
*****
MAIN MENU
*****
1 . Add Prison
2 . Add Inmate
3 . Search Prison
4 . Search Inmate
5 . Delete Prison
6 . Delete Inmate
7 . Display Prisons
8 . Display Inmates
0 . Exit
-----
Enter your choice: 6
Enter Inmate ID to delete: 201
Inmate not found.
```

Figure 4.7 Delete an Inmate record

4.4.8 Display all Prison record

```
*****
WELCOME TO PRISON MANAGEMENT
*****
MAIN MENU
*****
1 . Add Prison
2 . Add Inmate
3 . Search Prison
4 . Search Inmate
5 . Delete Prison
6 . Delete Inmate
7 . Display Prisons
8 . Display Inmates
0 . Exit
-----
Enter your choice: 7
Prison ID   Prison Name   Prison Location
101         Bangalore_prison   Bangalore
```

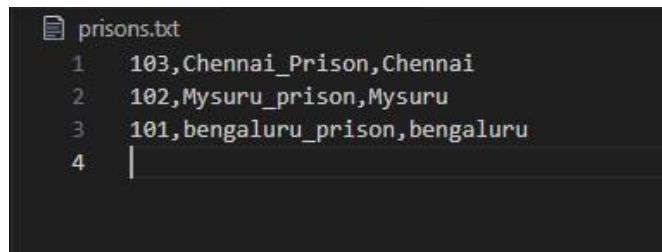
Figure. 4.8 Displaying all Prison records

4.4.9 Display all Inmate record

```
*****
WELCOME TO PRISON MANAGEMENT
*****
MAIN MENU
*****
1 . Add Prison
2 . Add Inmate
3 . Search Prison
4 . Search Inmate
5 . Delete Prison
6 . Delete Inmate
7 . Display Prisons
8 . Display Inmates
0 . Exit
-----
Enter your choice: 8
Inmate ID   Inmate Name   Prison ID
201         Amardeep      101
```

Figure. 4.9 Displaying all inmate records

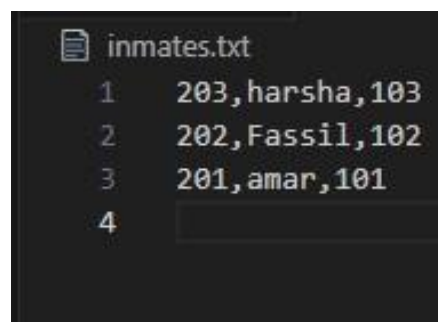
4.4.10 Output of Prison data file



```
prisons.txt
1 103,Chennai_Prison,Chennai
2 102,Mysuru_prison,Mysuru
3 101,bengaluru_prison,bengaluru
4 |
```

Figure. 4.10 Contents of Prison Data File

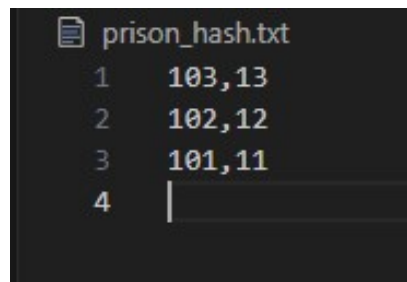
4.4.11 Output of Inmate data file



```
inmates.txt
1 203,harsha,103
2 202,Fassil,102
3 201,amar,101
4 |
```

Figure. 4.11 Contents of Inmate data File

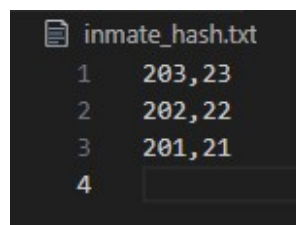
4.4.12 Output of Prison hash file



```
prison_hash.txt
1 103,13
2 102,12
3 101,11
4 |
```

Figure. 4.12 Contents of Prison hash File

4.4.13 Output of Inmate hash file



```
inmate_hash.txt
1 203,23
2 202,22
3 201,21
4 |
```

Figure. 4.13 Contents of Inmate Hash File

Chapter 5

CONCLUSION AND FUTURE ENHANCEMENTS

In conclusion, the implementation of a prison management system using a hashing file structure with collision avoidance techniques provides an efficient and reliable solution for storing and retrieving prison records. The system effectively utilizes hashing to distribute records across multiple files, ensuring balanced load distribution and minimizing access time.

The system allows for fast record retrieval based on various search criteria, leveraging the hashed index to minimize search time and provide quick access to desired records. It is scalable and performs well even with a growing number of prison records, thanks to the chosen file structure and collision resolution technique. The system maintains data integrity and consistency, providing mechanisms for accurate record insertion, updating, and deletion while handling collisions and file structure modifications.

Future Enhancements

There are several avenues for future enhancements to further improve the prison management system:

- **User Authentication:** Implement secure login system.
- **Role-Based Access Control:** Control user access based on roles.
- **Enhanced Search Functionality:** Improve search options for inmates and prisons.
- **Data Visualization:** Present data in graphical form for better understanding.
- **Reporting and Analytics:** Generate useful reports and insights.
- **Integration with External Systems:** Connect with other relevant systems.
- **Modification Function:** Currently, there is no modification function implemented in the system. Adding a modify function would enable users to update prison and inmate details easily.

REFERENCES

- [1] Michael J. Folk, Bill Zoellick, Greg Riccardi: File Structures-An Object-Oriented Approach in C++, PEARSON, 3rd Edition, Pearson Education, 1998.
- [2] K.R. Venugopal, K.G. Srinivas, P.M. Krishnaraj: File structures Using C++, TATA McGraw Hill 2008.
- [3] www.wikipedia.org
- [4] www.w3cSchool.com/php?
- [5] www.tutorialspoint.com/sdlc/
- [6] www.geeksforgeeks.org
- [7] <https://forum.directadmin.com/showthread.php>