

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

Jnana Sangama, Santhibastawad Road, Machhe

Belagavi - 590018, Karnataka, India



**A REPORT
ON**

“Event Management ”

**Submitted in the partial fulfilment of the requirements for the completion of the VI
Semester File Structures Laboratory with Mini Project [18ISL67] course of**

**BACHELOR OF ENGINEERING
IN
INFORMATION SCIENCE AND ENGINEERING**

For the Academic Year 2022-2023

Submitted by

**ARUN M MIRLE
ASHISH B R**

**1JS20IS024
1JS20IS025**

Under the Guidance of

Mrs. Sahana V

Assistant Professor, Dept. of ISE, JSSATEB



2022-2023

**DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING
JSS ACADEMY OF TECHNICAL EDUCATION**

**(Affiliated to VTU Belgavi and Approved by AICTE New Delhi)
JSSATE-B Campus, Dr.Vishnuvardhan Road, Bengaluru-560060**

JSS MAHAVIDYAPEETHA, MYSURU
JSS ACADEMY OF TECHNICAL EDUCATION

JSS Campus, Dr.Vishnuvardhan Road, Bengaluru-560060

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING



CERTIFICATE

This is to certify that the File Structures Mini Project entitled “EVENT MANAGEMENT” has been successfully completed by **ARUN M MIRLE [1JS20IS024], ASHISH B R [1JS20IS025]** of VI semester Information Science and Engineering from JSS Academy of Technical Education under Visvesvaraya Technological University during the academic year 2022- 2023.

Signature of the Guide

Mrs. Sahana V
Assistant Professor,
Dept. of ISE, JSSATEB

Signature of the HOD

Dr. Rekha P M,
Professor & HOD,
Dept. of ISE, JSSATEB

Name of Examiners

Signature with date

- 1.
- 2.

TABLE OF CONTENTS

ACKNOWLEDGEMENT	4
ABSTRACT	5
CHAPTER 1	
INTRODUCTION	6
1.1 Introduction to File Structures	6
1.2 History	6
1.3 About the File	6
1.4 Application of File Structures	7
CHAPTER 2	
SYSTEM ANALYSIS	8
2.1 Analysis of Project	8
2.2 Structures used to store fields and records	8
2.3 Operations performed on file	10
2.4 Indexing used	11
CHAPTER 3	
SYSTEM DESIGN	12
3.1 Design of Fields and Records	12
3.2 User Interface	12
CHAPTER 4	
IMPLEMENNTATION	15
4.1 About C++	15
4.2 Testing	16
4.3 Code of Functions	19
4.4 Discussion of Results	33
CHAPTER 5	
CONCLUSION	37
CHAPTER 6	
REFERENCES	38

ACKNOWLEDGEMENT

The success and final outcome of this project required a lot of guidance and assistance from many people and we are extremely privileged to have got this all along the completion of the project.

First and foremost, we would like to express my sincere gratitude to His Holiness Jagadguru Sri Sri Shivarathri Deshikendra Mahaswamiji for his divine blessings, without which the work wouldn't have been possible.

We would like to thank Dr Mrathyunjaya V Latte, Principal, ISSATE Bengaluru for his constant encouragement.

We would like to thank Dr. Rekha P M, Associate Professor and Head of the Department. Information Science and Engineering who has shared his opinions and thoughts, which helps in giving our presentation successfully.

We would like to express our gratitude to Mrs Sahana V, Asst. Professor, Dept. of ISE. For JSSATE, Bengaluru her guidance and assistance.

Finally, we take this opportunity to extend my earnest gratitude and respect to our parents, teaching & non-teaching staffs of the department and all my friends, for giving us valuable advices and support at all times in all possible ways.

ARUN M MIRLE [1JS20IS024]

ASHISH B R [1JS20IS025]

ABSTRACT

Event management systems are widely used in various domains to efficiently organize and manage events. In this project, we propose an Event Management System implemented using file structures and indexing techniques. The system aims to provide efficient storage, retrieval, and manipulation of event records.

The key features of the proposed system include the ability to add new events, update existing event details, delete events, and perform efficient searches based on various criteria, such as event name, date, location, or type. The indexing mechanism ensures fast access to event records, reducing the search time and improving overall system performance. The system incorporates error-handling mechanisms to handle exceptions, validate user input, and maintain data integrity. It also supports features like sorting events based on different attributes.

Using file structures and indexing offers a robust and efficient solution for organizing and managing events. It provides a scalable and flexible architecture that can handle a large volume of event data while ensuring optimal performance in terms of storage utilization and search operations.

CHAPTER 1

INTRODUCTION

1.1 Introduction to File Structures

File Structure is a combination of representations of data in files and operations for accessing the data. It allows applications to read, write and modify data. It supports finding the data that matches some search criteria or reading through the data in some particular order.

1.2 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.

1.3 About the File

There is one important distinction in file structures and that is the difference between the logical and physical organization of the data. On the whole a file structure will specify the logical structure of the data, that is the relationships that will exist between data items independently of the way in which these relationships may actually be realized within any computer. It is this logical aspect that we will concentrate on.

The physical organization is much more concerned with optimizing the use of the storage medium when a particular logical structure is stored on, or in it. Typically for every unit of a physical store there will be a number of units of the logical structure (probably records) to be stored in it.

A file system consists of two or three layers. Sometimes the layers are explicitly

separated, and sometimes the functions are combined. The logical file system is responsible for interaction with the user application. It provides the application program interface (API) for file operations- open, close, read etc., and passes the requested operation to the layer below it for processing. The logical file system manages open file table entries and per-process file descriptors. This layer provides file access, directory operations, and security and protection.

The second optional layer is the virtual file system. "This interface allows support for multiple concurrent instances of physical file systems, each of which is called a file system implementation.

The third layer is the physical file system. This layer is concerned with the physical operation of the storage device (eg. disk). It processes physical blocks being read or written. It handles buffering and memory management and is responsible for the physical placement of blocks in specific locations on the storage medium. The physical file system interacts with the device drivers or with the channel to drive the storage device.

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. Therefore, 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, nonvolatile capacity, is the driving force behind file structure design

CHAPTER 2

SYSTEM ANALYSIS

Systems analysis is the process of observing systems for troubleshooting or development purposes. It is applied to information technology, where computer-based systems require defined analysis according to their makeup and design.

2.1 Analysis of Project

The application is concerned with the Event Management System, which is where the organizer can keep track of the events and their schedules. The user has the option to show the menu, add, remove, and update the events details, as well as to change the data that has already been recorded. In this project, the indexing approach is used to add, remove, and modify records.

2.2 Structure used to store fields and records

2.2.1 Field Structure

A field is the smallest, logically meaningful, unit of information in a file. Four of the most common methods of adding structure to files to 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

Force the fields into a predictable length. This method relies on creating fields of predictable fixed size.

Disadvantages:

- A lot of wasted space due to padding of fields with whitespace or empty space.
- Data values may not fit in the predetermined length of the fields.

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 a Delimiter

This method requires that the fields be separated by a selected special character or a sequence of characters called a delimiter. However, choosing the right delimiter is very important. This method is most often used.

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

This method requires that each field data be preceded with the field identifier/ keyword. It can also be used with the combination of delimiter to mark the field ends.

Advantages :

- Each field provides information about itself.
- Good format for dealing with missing fields

Disadvantage:

A lot of space may be wasted on field keywords.

Here, in this project we are using the method of separating fields with a delimiter , ‘|’ for representing our fields in the files.

2.2.2 Record Structures

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

- Require the records to be a 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.

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

This method specifies the number of fields in each record. Regardless of the method for storing fields, this approach allows for relatively easy means of calculating record boundaries.

Method 3: Begin Each Record with a Length Indicator

We can communicate 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 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 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. Here, we use a '#' character as the record delimiter.

2.3 Operation Performed on a File**2.3.1 Insertion**

Insertion function adds new records into the file. In this process the inserted node descends to the leaf where the key fits. If the node has an empty space, insert the key/reference pair into the node. If the node is already full, split at the median push the median key upwards and make the left keys as a left child and the right keys as a right child.

2.3.2 Deletion

Deletion is the process in which on the parent node to remove the split key that previously separated these merged nodes unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root

2.3.3 Modify

The system can also be used to modify existing records from all production inventory details. The user is prompted for a key, which is used as to search the records. Once the record is found we rewrite details of the corresponding product. Hence modify () is followed by the read (). If the record is not found then it returns -1 value and display 'RECORD DOES NOT EXISTS' message on the console.

2.3.4 Searching

Searching is similar to searching a binary search tree. Starting at the root, the tree is recursively traversed from top to bottom. At each level, the search reduces its field of view to the child pointer (subtree) whose range includes the search value. A subtree's range is defined by the values, or keys, contained in its parent node.

2.4 Indexing Used

Indexing in file structures involves creating data structures that enable efficient data retrieval. It typically includes techniques such as primary indexing, where records are organized based on a primary key, and secondary indexing, which creates additional indexes based on non-key attributes to facilitate alternate access paths.

In primary indexing, a file is organized based on a primary key, which is a unique identifier for each record. This key is used to create an index that maps the key values to the corresponding physical location of the records in the file. This allows for direct access to specific records using their primary key.

Secondary indexing, on the other hand, involves creating an index based on non-primary key attributes. It allows for multiple access paths to the data, enabling efficient retrieval based on different search criteria. Secondary indexes are typically implemented using B-trees, providing sorted access to the records based on the indexed attribute.

By combining primary and secondary indexing, file structures can optimize data retrieval operations. Primary indexes offer direct access to records based on the primary key, while secondary indexes provide efficient access to records based on other attributes. This combination enables quick and flexible data retrieval, supporting a variety of search patterns and improving overall system performance.

CHAPTER 3

SYSTEM DESIGN

The System Design Document describes the system requirements, operating environment, system and subsystem architecture, files and database design, input formats, output layouts, human-machine interfaces, detailed design, processing logic, and external interfaces. Systems design is the process of defining the architecture, modules, interfaces, and data for a system to satisfy specified requirements. Systems design could be seen as the application of systems theory to product development.

3.1 Design of the Fields and Records

We use fixed length records and fixed size fields. In a fixed length record, it will take the maximum space needed for that field and at the end of the record there will be a delimiter placed indicating that it is the end of the record. A delimiter is a sequence of one or more characters used to specify the boundary between separate, independent regions in plain text or other data streams. The fields are separated using a delimiter ('|').

Index File: It contains all of the index data that was saved in this file by the application. It includes the patient's phone number or contact information as well as their index number.

3.2 User Interface

The user interface (UI), in the industrial design field of human-computer interaction, is the space where interactions between humans and machines occur. The goal of this interaction is to allow effective operation and control of the machine from the human end, whilst the machine simultaneously feeds back information that aids the operators' decision-making process.

```

      FILE STRUCTURE
M A I N   M E N U
1. EVENT MANAGEMENT
2. SEARCHING
3. QUIT
  PLEASE ENTER YOUR CHOICE [1-3] : 1
      Event Management
B M A I N   M E N U
1.> INSERT
2.> UPDATE
3.> DISPLAY ALL
4.> ASSIGN
5.> DELETE
6.> QUIT
  PLEASE ENTER YOUR CHOICE [1-6] :
|

```

Figure 3.1 User Interface

3.2.1 Insertion of a record

To insert an event record in the event management system, the user is prompted to enter the Event ID, Event Name, Event Location, Event Time, and Event Price. Each input is obtained from the user until values for all fields have been read. Once all the values are accepted, an indexing function is called to update the index based on the Event ID. This indexing function does mapping to the Event ID to the record's position in the data file. Furthermore, the record is stored in the data file by appending it to the end. After successful insertion, a message is displayed to confirm the record's insertion. The user can then press any key to return back to the menu screen for further actions.

3.2.2 Display of a record

To display an event record in the event management system, the user is presented with options to display all the information present in the records. The system then displays the record(s) to the user, presenting the relevant information on the screen. The user can review the details of the event record and perform further actions if needed, such as editing or deleting the record. Once the user has finished reviewing the record, they can press any key to return back to the menu screen and proceed with other operations available in the event management system.

3.2.3 Deletion of a record

To delete an event record in the event management system, the user is provided with options to search for the specific event they wish to delete. The search can be performed using criteria such as the Event ID. Once the user enters the Event ID, the system leverages the indexing concept to efficiently locate the record in the data file or database. The matching event record is then displayed to the user, presenting the relevant information including the Event ID, Event Name, Event Location, Event Time, and Event Price. The user can review the details of the event before confirming the deletion. If the user confirms the deletion, the system proceeds to remove the event record from both the data file and the index structure, ensuring data integrity. A confirmation message is displayed, indicating the successful deletion of the event record. Finally, the user can press any key to return back to the menu screen and continue with other operations in the event management system.

3.2.4 Assigning of a record

To assign a record in the event management system, the user is provided with options to select and assign specific events to individuals or entities. The user can search for the desired event based on criteria such as the Event ID. Using the indexing concept, the system efficiently retrieves the relevant event record from the data file or database. The retrieved record displays important information including the Event ID, Event Name, Event Location, Event Time, and Event Price. The user can review the details of the event and then proceed to assign it to the intended recipient or entity. This could involve associating the event with a specific individual, group, or organization. Upon successful assignment, the system updates the record accordingly and provides a confirmation message to the user. The user can then press any key to return to the menu screen and perform further actions in the event management system.

3.2.5 Modification of a record

To modify an event record in the event management system, the user is given options to search for and retrieve the specific event they wish to modify. They can search using criteria such as the Event ID. Utilizing the indexing concept, the system efficiently retrieves the matching record from the data file or database. The retrieved record displays essential details, including the Event ID, Event Name, Event Location, Event Time, and Event Price. The user can review the current information and then proceed to modify the desired fields. The system allows the user to make changes such as updating the Event Name, Event Location, Event Time, or Event Price based on their requirements. Once the modifications are complete, the system updates the record in both the data file and the index structure to ensure consistency. A confirmation message is displayed, indicating the successful modification of the event record. Finally, the user can press any key to return to the menu screen and continue with other operations within the event management system.

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 object-oriented programming (OOP) language, developed by Bjarne Stroustrup, and is an extension of the C language. It is therefore possible to code C++ in a "C style" or "object-oriented style." In certain scenarios, it can be coded in either way and is thus an effective example of a hybrid language.

C++ is considered to be an intermediate-level language, as it encapsulates both high- and low-level language features. Initially, the language was called "C with classes" as it had all the properties of the C language with an additional concept of "classes." However, it was renamed C++ in 1983.

The main highlight of C++ is a collection of predefined classes, which are data types that can be instantiated multiple times. The language also facilitates declaration of user defined classes. Classes can further accommodate member functions to implement specific functionality. Multiple objects of a particular class can be defined to implement the functions within the class. Objects can be defined as instances created at run time. These classes can also be inherited by other new classes which take in the public and protected functionalities by default.

C++ includes several operators such as comparison, arithmetic, bit manipulation and logical operators. One of the most attractive features of C++ is that it enables the overloading of certain operators such as addition. A few of the essential concepts within the C++ programming language include polymorphism, virtual and friend functions, templates, namespaces and pointers.

4.2 Testing

4.2.1 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. It usually has one or a few inputs and usually a single output.

Unit testing is commonly automated, but may still be performed manually. The objective in unit testing is to isolate a unit and validate its correctness. A manual approach to unit testing may employ a step-by-step instructional document. Unit testing is the process of testing the part of the program to verify whether the program is working correctly or not. In this part the main intention is to check each and every input 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.

Unit testing may reduce uncertainty in the units themselves and can be used in a bottom-up testing style approach. By testing the parts of a program first and then testing the sum of its parts, integration testing becomes much easier. Unit test cases embody characteristics that are critical to the success of the unit.

Table 4.1 Unit Testing for Event

Test Case ID	Description	Input data	Expected Output	Actual Output	Status
1	Opening a file insert the data	Insert option is selected	First field to be entered should appear without any error messages	First field entry request appeared	Pass
2	Validating Event ID	12	Accept the Event ID.	Accepts Event ID and continues	Pass
3	Validating Event Time	12:00	Accepts the Event Time.	Accepts Event Time and continues	Pass
4	Validating Event price	100	Accepts the Event Price.	Accepts Event Time and saves the record in the file	Pass

4.2.2 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 are working fine or not. This testing can be done by choosing the options in the program and by giving suitable inputs it is tested.

Table 4.2 Integration Testing

Test Case ID	Description	Input data	Expected Output	Actual Output	Status
1	Take input from user	Event ID=12 Event Name = Verve Event Location= Bangalore Event Time= 18:00 Event Price = 300	The information has been inserted in to list.txt	The information has been inserted in to list.txt	Pass
2	Update a record	Event Name = Samyog	The record has been updated	The record has been updated	Pass
3	Display all the event records	Display in the order they were inserted.	All event records must be fetched from list.txt and records should be displayed	All event records are fetched from list.txt and records are displayed	Pass
4	Assign a record	Request for the Event Id to be assigned to	Requests for details of the new event	New event is created which assigned to the Event ID entered.	Pass
5	Delete a record	Event ID=12	Record deletion successful	Record was deleted successfully from the list.txt file	Pass

4.2.3 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 prerequisite 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. And is implemented in below table 4.3

Table 4.3 System Testing

Test Case ID	Description	Input data	Expected Output	Actual Output	Status
1	To modify details	For valid Event ID, Event Time & Event Price	After displaying the current event information, a prompt asking the user to enter the information that needs to be changed for that event is displayed.	Displays the current event information, then prompt asks the user to re-enter the details for that event	Pass
2	To Display all events	Display in the order they were inserted	All event record must be fetched from list.txt and the records should be displayed	All patient record is fetched from memory.txt and the records are displayed	Pass

4.3 Code of Functions

*/*Event class*/*

class Event

{

public:

char eid[30],ename[30],elocation[30],etime[30],eprice[30];

Event()

{

eid[0]=0;

ename[0]=0;

elocation[0]=0;

etime[0]=0;

eprice[0]=0;

}

void Input();

void Modify();

void Assign1(Event &);

void Assign();

void Display();

void Secondary();

*int Binary(char *);*

void SecondSort();

void Index();

void del();

void KeySort();

void Indexdisplay(int);

*void Secondsearch(char *);*

void Invertprimary();

void Invertsecondary();

void Invertsecondarysort();

void CheckDup();

void InvertSecondarySort();

*int InvertSecondSearch(char *);*

};

class fsearch

{

public:

*int Search(char *);*

};

```

void Event::Modify()
{
    char key[20],Buff[100];
    int size=0,found=0,asize=75,Next=0;
    cout<<"\n\tENTER THE SEARCH KEY: ";
    cin>>key;
    VariableFieldBuffer b;
    fstream file("list.txt",ios::in/ios::out/ios::ate/ios::binary);
    file.seekg(0,ios::beg);
    while(b.unpack(*this,file))
    {
        if(strcasecmp(eid,key)==0)
        {
            found=1;
            break;
        }
        size=size+75;
    }
    if(found==1)
    {
        char choice[10];
        cout<<"\n\n\t\tRecord is found ...."<<endl<<endl;
        cout<<"\n\t\tDetails Of Record";
        cout<<"\n\t\n";
        cout<<"\n\t\tEvent Id   : "<<eid<<endl;
        cout<<"\n\t\tEvent Name   : "<<ename<<endl;
        cout<<"\n\t\tEvent Location : "<<elocation<<endl;
        cout<<"\n\t\tEvent Time   : "<<etime<<endl;
        cout<<"\n\t\tEvent Price  : "<<eprice<<endl;

        cout<<"\nDO You Want To Update Event Name(Y/N)   : ";
        cin>>choice;
        if(choice[0]=='y' || choice[0]=='Y')
        {
            cout<<"\n\t\tENTER THE Event Name       : ";
            cin>>ename;
        }

        cout<<"\nDO You Want To Update Event Location(Y/N) : ";
        cin>>choice;

        if(choice[0]=='y' || choice[0]=='Y')
        {
            cout<<"\n\t\tENTER THE Event Location       : ";
            cin>>elocation;
        }

        cout<<"\nDO You Want To Update Event Time(Y/N) : ";
        cin>>choice;

        if(choice[0]=='y' || choice[0]=='Y')
        {
            cout<<"\n\t\tENTER THE Event Time       : ";
            cin>>etime;
        }

        cout<<"\nDO You Want To Update Event Price(Y/N) : ";
        cin>>choice;

        if(choice[0]=='y' || choice[0]=='Y')
        {
            cout<<"\n\t\tENTER THE Event Price       : ";
            cin>>eprice;
        }

        for(int i=0;i<75;i++)
    }
}

```

```
{
    Buff[i]=' ';
}

this->Assign1(s1);
memcpy(&Buff,&asize,sizeof(asize));
strcpy(Buff,s1.eid); strcat(Buff,"|");
strcat(Buff,s1.ename);strcat(Buff,"|");
strcat(Buff,s1.elocation);strcat(Buff,"|");
strcat(Buff,s1.etime);strcat(Buff,"|");
strcat(Buff,s1.eprice);strcat(Buff,"|");
Next=strlen(Buff);

if(Next>asize)
    cout<<"\n\n\t\t\tRecord Overflow .... \n";
else
{
    file.seekp(size);
    file.write(Buff,asize);
    cout<<"\n\n\t\t\tRecord Is Updated ....\n";
}

}

else
    cout<<setw(40)<<"\n\n\n\t\t\tRecord Not Found ....\n";

file.close();
}
```

```

void Event::Assign1(Event &s1)
{
    strcpy(s1.eid,eid);
    strcpy(s1.ename,ename);
    strcpy(s1.elocation,elocation);
    strcpy(s1.etime,etime);
    strcpy(s1.eprice,eprice);
}

bool isValidTime(const char* etime) {
    // Check if the time string is not empty
    if (strlen(etime) != 5) {
        cout << "Invalid number of characters" << endl;
        return false;
    }

    // Extract hours and minutes from the time string
    int hours = (etime[0] - '0') * 10 + (etime[1] - '0');
    int minutes = (etime[3] - '0') * 10 + (etime[4] - '0');

    // Validate hours and minutes
    if (hours < 0 || hours > 23 || minutes < 0 || minutes > 59) {
        cout << "Invalid time format! Please enter a valid time in the 24-hour format (HH:MM)." << endl;
        return false;
    }

    return true;
}

bool isValidNumeric(const char* input) {
    // Check if the input string is empty
    if (input[0] == '\0')
        return false;

    // Check each character in the input string
    for (int i = 0; input[i] != '\0'; i++) {
        // If any character is not a digit, return false
        if (!isdigit(input[i]))
            return false;
    }

    return true;
}

bool isValidEID(char *eid) {
    bool isValid = true;
    if (!isdigit(eid[0]))
        isValid = false;

    return isValid;
}

```

```

void Event::Input()
{
    int k,value;
    char usn[30];
    Event a;
    fsearch v;
    VariableFieldBuffer b;
    fstream file;
    cout<<"\n\t-----\n\n";
    cout<<"\n\tENTER THE Event Id   :";
    cin>>eid;
    while(! isValidEID(eid)) {
        cout << "The Event ID isn't valid!" << endl << "Enter Event ID again: ";
        cin >> eid;
    }
    while(v.Search(eid)==1)
    {
        cout<<"\n\nDuplicate Entry Re-Enter The Event Id Value: ";
        cin>>eid;
    }
    cout<<"\n\tENTER THE Event Name   :";
    cin>>ename;

    cout<<"\n\tENTER THE Event Location :";
    cin>>elocation;
    cout << "\n\tENTER THE Event Time (in 24-hour format): ";
    cin >> etime;
    while (!isValidTime(etime)) {
        cout << "Please re-enter the time in 24-hour format: ";
        cin >> etime;
    }
    cout<<"\n\tENTER THE Event Price  :";
    while (true)
    {
        cin >> eprice;

        if (!isValidNumeric(eprice)) {
            cout << "Invalid price! Please enter a numerical value for Event Price: ";
            continue;
        }

        if (strcmp(eprice, "0") == 0) {
            cout << "Invalid price! Please enter a non-zero value for Event Price: ";
            continue;
        }

        // Break out of the loop if a valid numeric value other than zero is entered
        break;
    }

    file.open("list.txt",ios::app);
    k=b.pack(*this,file);
    file.close();
    if(k==1)
        cout<<"\n\n\t\t\tRecord Inserted ....\n\n";
    else
        cout<<"some problem";
}

```

```
/*Display function*/
```

```
void Event :: Display()
```

```
{
    VariableFieldBuffer c;
    fstream file("list.txt",ios::in);
    file.seekg(0,ios::beg);
    cout<<setw(25)<<" "<<"DISPLAYING RECORD'S"<<endl<<endl<<endl<<endl;
    while(c.unpack(*this,file))
    {
        int i;
        cout<<setw(20)<<" "<<"Event ID : "<<eid<<endl<<endl<<endl;
        cout<<setw(20)<<" "<<"Event Name      : "<<ename<<endl<<endl<<endl;
        cout<<setw(20)<<" "<<"Event Location    : "<<elocation<<endl<<endl<<endl;
        cout<<setw(20)<<" "<<"Event Time      : "<<etime<<endl<<endl<<endl;
        cout<<setw(20)<<" "<<"Event Price     : "<<eprice<<endl<<endl<<endl<<endl;
        for(i=0;i<80;i++) cout<<"*";
        cout<<endl<<endl<<setw(39)<<" ";

    }
    if(!(c.unpack(*this,file)))
    {
        cout<<endl<<endl<<endl<<endl<<setw(40)<<"NO RECORDS FOUND ....\n";

        return;
    }

    file.clear();
    file.close();
}
```

```
/*Assign function*/
```

```
void Event::Assign()
```

```
{
    char key[20],Buff[100];
    int size=0,found=0,asize=75,Next=0;
    cout<<"\n\tENTER THE SEARCH KEY :";
    cin>>key;
    VariableFieldBuffer b;
    fsearch v;
    fstream file("list.txt",ios::app|ios::in|ios::out|ios::ate|ios::binary);
    file.seekg(0,ios::beg);
    while(b.unpack(*this,file))
    {
        if(strcasecmp(eid,key)==0)
        {
            found=1;
            break;
        }
        size=size+75;
    }
    if(found==1)
    {
        char choice[10];
        cout<<"\n\n\t\tRecord is found ...."<<endl<<endl;
        cout<<"\n\t\t\tDetails Of Record";
        cout<<"\n\t\t\n";
        cout<<"\n\t\t\tEvent Id      : "<<eid<<endl;
        cout<<"\n\t\t\tEvent Name    : "<<ename<<endl;
        cout<<"\n\t\t\tEvent Location : "<<elocation<<endl;
        cout<<"\n\t\t\tEvent Time   : "<<etime<<endl;
    }
}
```



```

cout<<"\n\t\tEvent Price  : "<<eprice<<endl;

cout<<"\n\t\tENTER THE Event Id      : ";
cin>>eid;

while(v.Search(eid)==1)
{
    cout<<"\n\nDuplicate Entry Re-Enter The Event Id Value  : ";
    cin>>eid;}
cout<<"\n\t\tENTER THE Event Name      : ";
cin>>ename;

cout<<"\nDO You Want To Update Event Location(Y/N)  : ";
cin>>choice;

if(choice[0]=='y' || choice[0]=='Y')
{
    cout<<"\n\t\tENTER THE Event Location      : ";
    cin>>elocation;}

cout<<"\nDO You Want To Update Event Time(Y/N)  : ";
cin>>choice;

if(choice[0]=='y' || choice[0]=='Y')
{
    cout<<"\n\t\tENTER THE Event Time      : ";
    cin>>etime;}

cout<<"\nDO You Want To Update Event Price(Y/N)  : ";
cin>>choice;

if(choice[0]=='y' || choice[0]=='Y')
{
    cout<<"\n\t\tENTER THE Event Price      : ";
    cin>>eprice;}

for(int i=0;i<75;i++)
{Buff[i]=' ';}

this->Assign1(s1);
memcpy( &Buff,&asize,sizeof(asize));
strcpy(Buff,s1.eid); strcat(Buff,"|");
strcat(Buff,s1.ename);strcat(Buff,"|");
strcat(Buff,s1.elocation);strcat(Buff,"|");
strcat(Buff,s1.etime);strcat(Buff,"|");
strcat(Buff,s1.eprice);strcat(Buff,"|");
Next=strlen(Buff);

if(Next>asize)
    cout<<"\n\n\t\tRecord Overflow .... \n";
else
{
    file.write(Buff,asize);
    cout<<"\n\n\t\tRecord Is Updated ....\n";
}

}

else
    cout<<setw(40)<<"\n\n\n\t\tRecord Not Found ....\n";

file.close();
}

```

```

/*Delete function*/

void Event::del()
{
    char key[20],Buff[100];
    int size=0,asize=75,found=0;
    cout<<"\nEnter THE RECORD KEY TO BE DELETED :";
    cin>>key;
    fsearch u;
    found=u.Search(key);
    if(found==1)
    {

        cout<<"\nRecord Is Found ....";
        VariableFieldBuffer b;
        fstream file,file1;

        file.open("list.txt",ios::app/ios::in/ios::out/ios::ate/ios::binary);
        file.seekg(0,ios::beg);
        file1.open("l1list.txt",ios::out);
        file1.seekg(0,ios::beg);
        while(b.unpack(*this,file))
        {
            if(strcmp(eid,key)==0)
            {
                cout<<"\n\n\n\n\t\tRecord Is Deleted ....\n";
                size=size+75;
            }
            else
            {
                for(int i=0;i<75;i++)
                {
                    Buff[i]=' ';
                }
                this->Assign1(s1);
                memcpy( &Buff,&asize,sizeof(asize));
                strcpy(Buff,s1.eid); strcat(Buff,"|");
                strcat(Buff,s1.ename);strcat(Buff,"|");
                strcat(Buff,s1.elocation);strcat(Buff,"|");
                strcat(Buff,s1.etime);strcat(Buff,"|");
                strcat(Buff,s1.eprice);strcat(Buff,"|");
                file1.write(Buff,asize);
                size=size+75;
            }
        }
        file1.close();
        file.close();
    }
    else
        cout<<"\nRecord Is Not Found ....";

    if(found==1)
    {
        unlink("list.txt");
        rename("l1list.txt","list.txt");
    }
}

```

```

void Event::Index()
{
    int size=0,length=75,i;
    char uns[15]="0",IBuff[75]="";
    count=0;
    ifstream ofile("list.txt",ios::in);
    ofile.seekg(0,ios::beg);
    fstream nfile("index.txt",ios::out);
    if(ofile.fail())
        cout<<"file not exist";
    else
        while(1)
        {
            ofile.read(IBuff,length);
            if(ofile.fail())
                break;
            for(i=0;IBuff[i]!='\0';i++)
                uns[i]=IBuff[i];
            count++;
            nfile.seekp(size);
            nfile<<uns<<"\n"<<count<<"\n";
            size=size+15;
        }
    nfile.close();
    ofile.close();
}

```

```

void Event::Secondary()
{
    int size=0,length=75, i;
    char uns[20]="0",ename[20]="0",Buffer[75]="";
    count=0;
    ifstream ofile("list.txt",ios::in);
    ofile.seekg(0,ios::beg);
    fstream nfile("Second.txt",ios::out);
    if(ofile.fail())
        cout<<"file not exist";
    else
        while(1)
        {
            ofile.read(Buffer,length);
            if(ofile.fail())
                break;

            int j=0;

            for(i=0;Buffer[i]!='\0';i++)
                uns[j++]=Buffer[i];
            uns[j]='\0';
            i++;

            for(j=0;Buffer[i]!='\0';i++)
                ename[j++]=Buffer[i];
            ename[j]='\0';

            count++;
            nfile.seekp(size);
            nfile<<ename<<"\n"<<uns<<"\n";
            size=size+30;
        }
    nfile.close();
    ofile.close();
}

```

```

void Event::SecondSort()
{
    char Buffer[30],str1[30]="",str2[30]="";
    int length=30,ptr,i=0,m,j;
    fstream file("second.txt",ios::in);
    file.seekg(0,ios::beg);
    while(1)
    {
        file.read(Buffer,length);
        if(file.fail())
            break;

        for(j=0;j<30;j++)
            knodes[i][j]=Buffer[j];
        i++;
    }

    for(j=0;j<30;j++)
        knodes[i][j]=Buffer[j];

    file.close();

    for(i=0;i<count;i++)
    {
        j=0;
        for(m=0;m<30;m++)
            str1[m]=knodes[i][m];
        j=i-1;
        a:

        for(m=0;m<30;m++)
            str2[m]=knodes[j][m];
        ptr=strcasecmp(str1,str2);

        while(j>=0&&ptr<0)
        {
            for(m=0;m<30;m++)
                knodes[j+1][m]=knodes[j][m];
            j--;
            goto a;
        }

        for(m=0;m<30;m++)
            knodes[j+1][m]=str1[m];
    }

    knodes[i][j]=Buffer[j];
}

```

```

void Event::Invertprimary()
{
    int size=0,length=75;
    char uns[20]="0",IBuff[75]="";
    count=0;
    ifstream ofile("list.txt",ios::in);
    ofile.seekg(0,ios::beg);
    fstream nfile("index1.txt",ios::out);
    if(ofile.fail())
        cout<<"file not exist";
    else
        while(1)
        {
            ofile.read(IBuff,length);
            if(ofile.fail())
                break;
            for(int i=0;IBuff[i]!='\0';i++)
                uns[i]=IBuff[i];
            count++;
            nfile.seekp(size);
            nfile<<count<<"|"<<uns<<"|"<<-1<<"|";
            size=size+25;
        }

    nfile.close();
    ofile.close();
}

void Event::Invertsecondary()
{
    int size=0,length=75, i, k;
    char uns[20]="0",IBuff[75]="",ename[20]="";
    count=0;
    Event as;
    ifstream ofile("list.txt",ios::in);
    ofile.seekg(0,ios::beg);
    fstream nfile("second_index.txt",ios::out);
    if(ofile.fail())
        cout<<"file not exist";
    else
        while(1)
        {
            ofile.read(IBuff,length);
            if(ofile.fail())
                break;

            for(i=0;IBuff[i]!='\0';i++)
                uns[i]=IBuff[i];
            i++;

            for(k=0;IBuff[i]!='\0';i++)
                ename[k++]=IBuff[i];
            ename[k]='\0';
            count++;
            nfile.seekp(size);
            nfile<<ename<<"|"<<count<<"|";
            size=size+25;
        }

    nfile.close();
    ofile.close();
}

void Event::Invertsecondarysort()
{

```

```

char Buffer[25],str1[30]="",str2[30]="";
int length=25,ptr,i=0, j, m;
fstream file("second_index.txt",ios::in);
file.seekg(0,ios::beg);
for(j=0;j<25;j++)
    knodes[i][j]='\0';
while(1)
{
    for(j=0;j<25;j++)
        Buffer[j]='\0';
    file.read(Buffer,length);

    if(file.fail())
        break;

    for( j=0;j<25;j++)
        knodes[i][j]=Buffer[j];
    i++;
}

for( j=0;j<25;j++)
    knodes[i][j]=Buffer[j];

file.close();

for(i=0;i<count;i++)
{
    j=0;
    for(m=0;m<25;m++)
        str1[m]=knodes[i][m];
    j=i-1;
a:
    for(m=0;m<25;m++)
        str2[m]=knodes[j][m];
    ptr=strcasecmp(str1,str2);

    while(j>=0&&ptr<0)
    {
        for(m=0;m<25;m++)
            knodes[j+1][m]=knodes[j][m];
        j--;
        goto a;
    }

    for(m=0;m<25;m++)
        knodes[j+1][m]=str1[m];
}

knodes[i][j]=Buffer[j];
}

```

```
//-----
/*Pack Function*/
void normal();
void screen();
void screen1();

int VariableFieldBuffer::pack(Event &s, fstream &file)
{
    int value;
    initialize();

    for(int i=0; i<75; i++)
    {
        Buffer[i]=' ';
    }

    memcpy(&Buffer[0], &Buffersize, sizeof(Buffersize));
    strcpy(Buffer, s.eid); strcat(Buffer, "|");
    strcat(Buffer, s.ename); strcat(Buffer, "|");
    strcat(Buffer, s.elocation); strcat(Buffer, "|");
    strcat(Buffer, s.etime); strcat(Buffer, "|");
    strcat(Buffer, s.eprice); strcat(Buffer, "|");
    Nextbyte = strlen(Buffer);

    if(Nextbyte > Buffersize)
        return 0;

    else
        value = Write(file);
    return value;
}
```

```
/*Unpack Function*/

int VariableFieldBuffer::unpack(Event &s, fstream &file)
{
    initialize();
    Nextbyte = 0;
    int i;
    int value = Read(file, Nextbyte);

    if (value == 0) {
        return 0;
    }

    if (Buffersize > 0) {
        int j = 0;

        for (i = 0; Buffer[i] != '|' && i < Buffersize; i++)
        {
            s.eid[j++] = Buffer[i];
        }
        s.eid[j] = '\0';
        i++;

        j = 0;
        for (; Buffer[i] != '|' && i < Buffersize; i++) {
            s.ename[j++] = Buffer[i];
        }
    }
}
```

```

    s.ename[j] = '\0';
    i++;

    j = 0;
    for (; Buffer[i] != '/' && i < Buffersize; i++) {
        s.elocation[j++] = Buffer[i];
    }
    s.elocation[j] = '\0';
    i++;

    j = 0;
    for (; Buffer[i] != '/' && i < Buffersize; i++) {
        s.etime[j++] = Buffer[i];
    }
    s.etime[j] = '\0';
    i++;

    j = 0;
    for (; Buffer[i] != '/' && i < Buffersize; i++) {
        s.eprice[j++] = Buffer[i];
    }
    s.eprice[j] = '\0';
    i++;

    Nextbyte = Buffersize + 75;
    return 1;
}

return 0;
}

```

*/*Function to search a key*/*

```

int fsearch::Search(char *key)
{
    int found=0;
    Event s1;
    VariableFieldBuffer b;
    fstream file("list.txt",ios::in|ios::out);
    file.seekg(0,ios::beg);
    while(b.unpack(s1,file))
        if(strcasecmp(s1.eid,key)==0)
            found=1;
    file.close();
    return found;
}

```


4.4 Discussions of results

4.4.1 Menu options

```

      FILE STRUCTURE
M A I N   M E N U
1. EVENT MANAGEMENT
2. SEARCHING
3. QUIT
  PLEASE ENTER YOUR CHOICE [1-3] :   1
      Event Management
B M A I N   M E N U
1.> INSERT
2.> UPDATE
3.> DISPLAY ALL
4.> ASSIGN
5.> DELETE
6.> QUIT
  PLEASE ENTER YOUR CHOICE [1-6] :
|

```

Figure 4.1 Menu options

```

      FILE STRUCTURE
M A I N   M E N U
1. EVENT MANAGEMENT
2. SEARCHING
3. QUIT
  PLEASE ENTER YOUR CHOICE [1-3] :   2
      SEARCHING
M A I N   M E N U
1.PRIMARY KEY SEARCH
2.SECONDARY KEY SEARCH
3.INVERTED LIST
4.QUIT
  PLEASE ENTER YOUR CHOICE [1-4] :

```

Figure 4.2 Additional menu options

Figure 4.1 and Figure 4.2 snapshots shows the menu, which has number of options and gives suggestions to user of the system. Event menu asks the user to Insert, Update, Display, Assign and Delete events, whereas the Search menu asks user for primary key search, secondary key search and Inverted list.

4.4.2 Read operations

Figure 4.3 snapshot shows the options for the user to enter various details like Event ID, Event Name, Event Location, Event Time and Event Price.

```

B M A I N   M E N U
1.> INSERT
2.> UPDATE
3.> DISPLAY ALL
4.> ASSIGN
5.> DELETE
6.> QUIT
  PLEASE ENTER YOUR CHOICE [1-6] :
1

-----

ENTER THE Event Id      :12
ENTER THE Event Name    :Verve
ENTER THE Event Location :Banglore
ENTER THE Event Time (in 24-hour format): 18:00
ENTER THE Event Price   :300

Record Inserted ....

```

Figure 4.3 Taking the details of events from the user

4.4.3 Display operations

Figure 4.4 Shows the details and displays all the information about the events which are stored in the file.

```

Event Management
B M A I N   M E N U
1.> INSERT
2.> UPDATE
3.> DISPLAY ALL
4.> ASSIGN
5.> DELETE
6.> QUIT
  PLEASE ENTER YOUR CHOICE [1-6] :
3

DISPLAYING RECORD'S

*****
Event ID : 12

Event Name      : Verve
Event Location   : Bangalore
Event Time      : 18:00
Event Price     : 300

*****
Event ID : 21

Event Name      : samyog
Event Location   : Bangalore
Event Time      : 19:00
Event Price     : 200

*****

```

Figure 4.4 Display of event records

4.4.4 Modify operations

```

Event Management
B M A I N   M E N U
1.> INSERT
2.> UPDATE
3.> DISPLAY ALL
4.> ASSIGN
5.> DELETE
6.> QUIT
PLEASE ENTER YOUR CHOICE [1-6] :
2

ENTER THE SEARCH KEY: 12

Record is found ....

Details Of Record

Event Id :12
Event Name :Verve
Event Location :Banglore
Event Time :18:00
Event Price :300
DO You Want To Update Event Name(Y/N) :Y
ENTER THE Event Name :Ignition
DO You Want To Update Event Location(Y/N) :N
DO You Want To Update Event Time(Y/N) :N
DO You Want To Update Event Price(Y/N) :Y
ENTER THE Event Price :200

Record Is Updated ....

```

Figure 4.6 Updating Event Records

Figure 4.6 is a snap shot of performing updating of the record details of event of Event ID '12' where the Event Name and Event Price has been updated.

4.4.5 File Contents

```

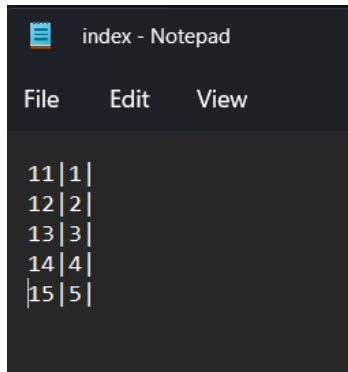
list - Notepad

File Edit View

11|Verve|Banglore|12:00|200|
12|Ignition|Banglore|01:00|300|
13|Samyog|Banglore|02:00|400|
14|DJ|Banglore|20:00|500|
15|Utsav|Banglore|21:00|500|

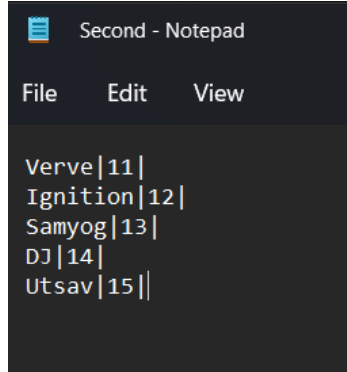
```

Figure 4.7 Contents of list.txt



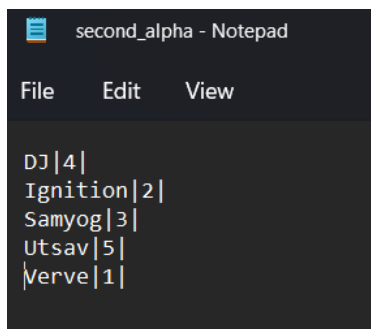
```
File Edit View

11|1|
12|2|
13|3|
14|4|
15|5|
```

Figure 4.8 Contents of index.txt

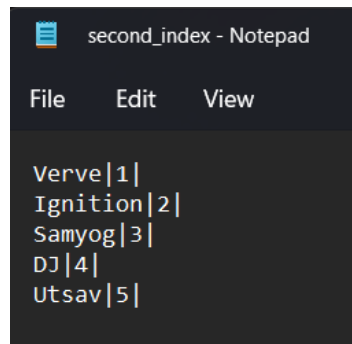
```
File Edit View

Verve|11|
Ignition|12|
Samyog|13|
DJ|14|
Utsav|15||
```

Figure 4.9 Contents of Second..txt

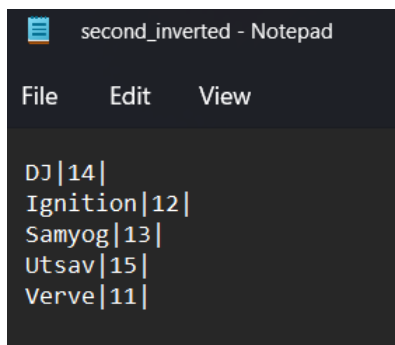
```
File Edit View

DJ|4|
Ignition|2|
Samyog|3|
Utsav|5|
Verve|1|
```

Figure 4.10 Contents of second_alpha.txt

```
File Edit View

Verve|1|
Ignition|2|
Samyog|3|
DJ|4|
Utsav|5|
```

Figure 4.11 Contents of second_index.txt

```
File Edit View

DJ|14|
Ignition|12|
Samyog|13|
Utsav|15|
Verve|11|
```

Figure 4.12 Contents of second_inverted.txt

CHAPTER 5

CONCLUSIONS

The use of file structures and applications in event management is instrumental in creating a well-organized and efficient planning process. By implementing a structured file system, event planners can easily locate and access critical documents, ensuring that important information is readily available. Additionally, event management applications facilitate seamless collaboration and communication among team members, enabling real-time updates, task assignments, and progress tracking.

These tools also aid in the creation and management of event schedules, budgets, and financial reports, streamlining the planning and financial aspects of events. Furthermore, the data analysis capabilities of these applications allow organizers to gain valuable insights into attendee engagement and preferences, leading to informed decision-making and improved future events. With automation features and adaptability to different event types and sizes, file structures and applications offer scalability and flexibility, contributing to the successful execution of events. Overall, leveraging these tools optimizes event-planning processes, enhances collaboration, and paves the way for memorable and successful events. The implementation of the system will reduce data entry time and provide readily calculated reports.

CHAPTER 6

REFERENCES

- [1] Michael J Folk, Bill Zoellick, Greg Riccardi: File Structures – An Object-Oriented Approach with C++, 3rd Edition, Pearson Education, 1998
- [2] K.R. Venugopal, K.G. Srinivas, P.M. Krishnaraj: File Structures Using C++, TataMcGraw-Hill, 2008.
- [3] Scot Robert Ladd: C++ Components and Algorithms, BPB Publications, 1993
- [4] Raghu Ramakrishnan and Johannes Gehrke: Database Management Systems, 3rd Edition, McGraw Hill, 2003
- [5] www.geeksforgeeks.com
- [6] www.medium.com
- [7] www.includeHelp.com
- [8] [Stack Overflow - Where Developers Learn, Share, & Build Careers](#)