

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

JNANA SANGAMA, BELAGAVI – 590 018



A Mini Project Report On

MUSIC INSTRUMENT MANAGEMENT SYSTEM USING B-TREE INDEXING TECHNIQUE

*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** of
Visvesvaraya Technological University, Belagavi*

Submitted By

ADITYA D
1RN19IS011

AKSHAY P
1RN19IS018

Under the Guidance of

Mr. Santhosh Kumar
Assistant Professor
Department of ISE, RNSIT



Department of Information Science and Engineering

RNS Institute of Technology

Channasandra, Dr. Vishnuvardhan Road, RR Nagar Post

Bengaluru – 560 098

2021 – 2022

RNS Institute of Technology

Channasandra, Dr. Vishnuvardhan Road, RR Nagar Post

Bengaluru – 560 098

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING



CERTIFICATE

This is to certify that the Mini Project Report entitled **MUSIC INSTRUMENTS MANAGEMENT SYSTEM USING B-TREE INDEXING TECHNIQUE** has been successfully completed by **ADITYA D** bearing USN **1RN19IS011** and **AKSHAY P** bearing USN **1RN19IS018**, presently VI semester students of **RNS Institute of Technology** in partial fulfillment of the requirements as a part of the File Structures Laboratory for the award of the degree **Bachelor of Engineering in Information Science and Engineering** under **Visvesvaraya Technological University**, Belagavi during academic year 2021 – 2022. 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.

Mr. Santosh Kumar

Project Guide

Dr. Suresh L

Professor and HOD

External Viva

Name of the Examiners

Signature with Date

1. _____

2. _____

ABSTRACT

The **Music Instruments Management System** is a File structure management system. It provides a user-friendly, interactive Menu Driven Interface (MDI) based on local file systems. All data is stored in files on disk. The application uses file handles to access the files. This project signifies the need for efficient management of Instruments presently available. The proposed system enables the administrator to keep track of all the Instruments records.

In this project, we have a fixed number of fields, each with a variable length, that combine to make a data record and variable length record. Fixing the number of fields in a record does not imply that the size of fields in the record is fixed. The records are used as containers to hold a mix of fixed and variable-length fields within a record.

Indexing technique used in this project is **B-Tree**. B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalized form of the binary search tree. B-tree can store many keys in a single node and can have multiple child nodes. This decreases the height significantly allowing faster disk accesses.

Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. The need for B-tree arose with the rise in the need for lesser time in accessing the physical storage media like a hard disk. The secondary storage devices are slower with a larger capacity. B-tree allows searches, insertions, and deletions in logarithmic amortized time. It is most commonly used in database and file systems.

ACKNOWLEDGEMENT

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 **Principal Dr. M K Venkatesha** for his support and inspired me 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, mastermind behind my Mini Project work for guiding me and helping me out with the report.

We would like to express our profound and cordial gratitude to our Mini Project Guide **Mr. Santosh Kumar**, Assistant Professor, Department of Information Science and Engineering for his valuable guidance, constructive comments and continuous encouragement throughout the Mini Project work.

We would like to thank all other teaching and non-teaching staff of Information Science & Engineering who have directly or indirectly helped me to carry out the mini project work. And lastly, 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.

Aditya D

1RN19IS011

Akshay P

1RN19IS018

TABLE OF CONTENTS

CERTIFICATE	
ABSTRACT	i
ACKNOWLEDGEMENT	ii
TABLE OF CONTENTS	iii
LIST OF FIGURES	v
LIST OF TABLES	vi
ABBREVIATIONS	vii
1. INTRODUCTION	1
1.1 Introduction to File Structures	1
1.1.1 History	1
1.1.2 About the File	2
1.1.3 Various Kinds of Storage of Fields and Records	2
1.1.4 Application of File Structure	5
2. SYSTEM ANALYSIS	6
2.1 Analysis of the Project	6
2.2 Structure Used to Store the Fields and Records	6
2.3 Operations Performed on a File	7
2.4 Indexing Used	8
3. SYSTEM DESIGN	9
3.1 Design of the Fields and Records	9
3.2 User Interface	9
3.2.1 Insertion of a Record	9
3.2.2 Display Records	10
3.2.3 Search a Record	10
3.2.4 Deletion of a Record	10
3.2.5 Modify a Record	11
4. IMPLEMENTATION	13
4.1 About C++	13
4.1.1 File Handling	14
4.2 Pseudocode	14
4.2.1 Inserting a New Record	14

4.2.2 Displaying Records	16
4.2.3 Searching a Record	17
4.2.4 Deleting a Record	18
4.2.5 Modifying a Record	19
4.3 Testing	19
4.3.1 Unit Testing	20
4.3.2 Integration Testing	21
4.3.3 System Testing	24
4.4 Discussion of Results	25
4.4.1 Menu Options	25
4.4.2 Insertion Operation	25
4.4.3 Display Operation	26
4.4.4 Search Operation	27
4.4.5 Delete Operation	29
4.4.6 Modify Operation	30
4.4.7 B-Tree Traversal	31
4.4.8 File Contents – Record File & Index File	32
5. CONCLUSIONS AND FUTURE ENHANCEMENTS	33
5.1 Conclusion	33
5.2 Future Enhancements	33
REFERENCES	34

LIST OF FIGURES

Figure No.	Description	Page No.
Figure 4.1	Inserting a new record	14
Figure 4.2	Insertion in a B-Tree	15
Figure 4.3	Displaying records	16
Figure 4.4	Get B-Tree keys in sorted order	16
Figure 4.5	Searching a record	17
Figure 4.6	Searching in a B-Tree	17
Figure 4.7	Deleting a record	18
Figure 4.8	Deletion in a B-Tree	18
Figure 4.9	Modifying a record	19
Figure 4.10	User Menu Screen	25
Figure 4.11	Successful insertion of record	26
Figure 4.12	Insertion of duplicate record	26
Figure 4.13	Display records – entry sequenced	27
Figure 4.14	Display records – sorted by ID	27
Figure 4.15	Search of a record present in the file	28
Figure 4.16	Search of a record not present in the file	28
Figure 4.17	Deleting a record	29
Figure 4.18	Modifying a record	30
Figure 4.19	B-Tree traversal	31
Figure 4.20	Record file contents	32
Figure 4.21	Index file contents	32

LIST OF TABLES

Table No.	Description	Page No.
Table 4.1	Unit Testing for Music Instruments Management System	20
Table 4.2	Integration Testing for Music Instrument Management System	22
Table 4.3	System Testing for Music Instruments Management System	24

ABBREVIATIONS

ASCII:	American Standard Code for Information Interchange
AVL:	Adelson-Velskii and Landis
CGPA:	Cumulative Grade Point Average
MDI:	Menu Driven Interface
OS:	Operating System
RAM:	Random Access Memory
UI:	User Interface

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.

1.1.1 History

Early work with files presumed that files were on tape, since most files were. Access was sequential and the cost of access grew in direct proportion to the size of the file. As files grew intolerably large for unaided sequential access and as storage devices such as hard disks became available, indexes were added to files. The indexes made it possible to keep a list of keys and pointers in a smaller file that could be searched more quickly. With key and pointer, the user had direct access to the large, primary file. But simple indexes had some of the same sequential flaws as the data file, and as the indexes grew, they too became difficult to manage, especially for dynamic files in which the set of keys changes. In the early 1960's, the idea of applying tree structures emerged. But trees can grow very unevenly as records are added and deleted, resulting in long searches requiring many disk accesses to find a record. In 1963, researchers developed an elegant, self-adjusting binary tree structure, called AVL tree, for data in memory. The problem was that, even with a balanced binary tree, dozens of accesses were required to find a record in even moderate sized files.

A method was needed to keep a tree balanced when each node of the tree was not a single record, as in a binary tree, but a file block containing dozens, perhaps even hundreds, of records. It took 10 years until a solution emerged in the form of a B-tree. Whereas AVL trees grow from the top down as records were added, B-trees grew from the bottom up. B- Trees provided excellent access performance, but there was a cost: no longer could a file be accessed sequentially with efficiency. The problem was solved by adding a linked list structure at the bottom level of the B-tree. The combination of a b-tree and a sequential linked list is called a B+ tree. B-trees and B+ trees provide access times that grow in proportion to $\log_k N$, where N is the number of entries in the file and k is the number of entries indexed in a single block of the B-tree structure. This means

that B-tree can guarantee that you can find 1 file entry among millions with only 3 or 4 trips to the disk. Further, B-trees guarantee that as you add and delete entries, performance stays about the same. Hashing is a good way to get what we want with a single request, with files that do not change size greatly over time. Hashed indexes were used to provide fast access to files. But until recently, hashing did not work well with volatile, dynamic files. Extendible dynamic hashing can retrieve information with 1 or at most 2 disk accesses, no matter how big the file became.

1.1.2 About the File

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

From the standpoint of an application program, a file is somewhat like a telephone line connection to a telephone network. The program can receive bytes through this phone line or send bytes down it, but it knows nothing about where these bytes come from or where they go. The program knows only about its end of the line. Even though there may be thousands of physical files on a disk, a single program is usually limited to the use of only about 20 files. The application program relies on the OS to take care of the details of the telephone switching system. It could be those bytes coming down the line into the program originate from a physical file they come from the keyboard or some other input device. Similarly, bytes the program sends down the line might end up in a file, or they could appear on the terminal screen or some other output device. Although the program doesn't know where the bytes are coming from or where they are going, it does know which line it is using. This line is usually referred to as the logical file, to distinguish it from the physical files on the disk or tape.

1.1.3 Various Kinds of Storage of Fields and Records

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

Field Structures: 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

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

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

Method 3: Separate the Fields with Delimiters

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

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

This has an advantage the others don't. It is the first structure in which a field provides information about itself. Such self-describing structures can be very useful tools for organizing files in many applications. It is easy to tell which fields are contained in a file, even if we don't know ahead of time which fields the file is supposed to contain. It is also a good format for dealing with missing fields. If a field is missing, this format makes it obvious, because the keyword is simply not there.

It is helpful to use this in combination with delimiters, to show division between each value and the keyword for the following field. But this also wastes a lot of space: 50% or more of the file's space could be taken up by the keywords. A record can be defined as a set of fields that belong together when the file is viewed in terms of a higher level of organization.

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

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

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

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

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

Method 2: Make Records a Predictable Number of Fields

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

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. 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, and then seek to the record in the data file.

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

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

1.1.4 Application of File Structure

Relative to other parts of a computer, disks are slow. 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, is the driving force behind file structure design.

Chapter 2

SYSTEM ANALYSIS

2.1 Analysis of the Project

In this project, we deal with the Instrument records which contain the fields such as ID, type of instrument, description, brand, colour, the quantity available and the price for each Instrument. It accepts the valid user input fields. It is a file structures project implemented using B-Tree indexing technique. It involves functions such as insertion, deletion, modification, searching and displaying of the data. A text file called instruments-list.txt, for storing the records has been used along with an index file called instrument-index.txt. The index file is re-written with every new insertion, deletion or modification of a record in the data file.

2.2 Structure Used to Store the Fields and Records

This project uses variable-length and delimiter for both field and records. A variable length field is not always the same physical size. Such fields are nearly always used for text fields that can be large, or fields that vary greatly in length. Reserving a fixed length field of some length would be inefficient because it would enforce a maximum length on abstracts, and because space would be wasted in most records. Database implementations commonly store varying-length fields in special ways, in order to make all the records of a given type have a uniform small size. Doing so can help performance. On the other hand, data in serialized forms such as stored in typical file systems, transmitted across networks, and so on usually uses quite different performance strategies. The choice depends on factors such as the total size of records, performance characteristics of the storage medium, and the expected patterns of access. In this project, there are fields like:

1. Instrument ID
2. Type of Instrument
3. Description
4. Brand Name
5. Color
6. Price and
7. Quantity

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. In this project, delimiter used for records is '\n' and for fields is '|'.

2.3 Operations Performed on a File

- **Insertion**

Insertion function adds new records into the data file and index file. With every insertion in the into the data file, the B-Tree is altered with the new insertion and the records containing the Instrument Id and its respective address is re-written into the index file.

- **Deletion**

In deletion, the data is deleted from the data file by adding a dollar (\$) sign in place of the first character. In the index file, the key whose record is to be deleted is searched in the B-Tree in memory. If found, the key is removed from its place and the index file is re-written with the updated records.

- **Modification**

Modify function involves deletion and insertion. When a record is to be modified, the record is first searched in the data file. On finding the record, the deletion function is called to delete the record in the same manner as explained above. After successful deletion, the insert function is called to ask user inputs for the new record. The new/modified record is appended at the end of data file. The index file is re-written with the key and the address to the new record.

- **Search**

In the search function, the user enters the key of the record to be searched. The key is searched in the B-Tree that is present in the memory. If the key is present, the address of the record is returned and the particular record is retrieved from the file and displayed to the user.

2.4 Indexing Used

Indexing used in this project is primary indexing implemented using B-Tree. A primary index is an ordered file whose records are of fixed length with two fields, and it acts like an access structure to efficiently search for and access the data records in a data file. The data file records could be ordered, unordered, or hashed. The index file is an ordered file with two fields: the primary key (Instrument ID) and the reference field (address of the record in the file).

B-Tree is a self-balancing search tree. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, etc.) require $O(h)$ disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size. Since the height of the B-tree is low that total disk accesses for most of the operations are reduced significantly.

Chapter 3

SYSTEM DESIGN

3.1 Design of the Fields and Records

In this project, we use variable length record and variable size fields. In variable length record, it will take the minimum space needed for that field and at the end of the record a delimiter is placed indicating that it is the end of the record. The fields are separated using a delimiter (‘|’) whereas newline (‘\n’) denotes end of a record. There are seven fields. These fields are:

1. ID
2. Type
3. Description
4. Brand
5. Colour
6. Price
7. Quantity

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.

3.2.1 Insertion of a Record

In the insertion process, the user selects option 1 from the menu. First, the user is asked to enter the ID of the new INSTRUMENT. A validation subroutine is executed to check if that key entered already exists. If yes, the user is asked to enter again. After reading the ID from standard input, the user is asked to enter the values of the other fields. Once all fields have been read from the standard input, the fields are packed one by one to the buffer, each separated by the delimiter ‘|’. The record file is opened in append mode. Then the buffer is written to the file and the address

where the new record is written to is returned. The key of the new record is inserted into the B-Tree in memory

Since the B-Tree has been modified in memory, the index file has to be re-written before the main program terminates.

3.2.2 Display Records

There are two choices to display the records:

1. Display the records based on entry-sequence (Option 2)

The record file is opened in read mode. Then using a loop, each record from the file is read into the buffer, unpacked one by one and the record is printed on the screen.

2. Display the records sorted by ID (Option 3)

The record file is opened in read mode. The keys and the record addresses from the B-Tree are fetched into a C++ vector of pair. This vector will contain the keys in sorted order. The vector is traversed and a direct read is performed at the current record address from file and the buffer is unpacked. Then the record is printed on the screen.

3.2.3 Search a Record

In the search process, the user selects option 4 from the menu. The user is asked to enter the key to be searched. The key is passed to the search() function of the B-Tree and the B-Tree returns the corresponding address of the key if it is present, otherwise -1. If the key is present, a direct read is performed at that address. The record is read into the buffer and unpacked. The final result is printed on the screen. If the subroutine returned -1, display a message saying the record was not found.

3.2.4 Deletion of a Record

In the deletion process, the user selects option 6 from the menu. The user is asked to enter the key to be deleted. Now the search subroutine is called to find the address of the record with that key. If the key was not present, exit the delete subroutine and display that the key doesn't exist. The file is opened in write mode and seek() function is called to position the pointer at the record address. The first letter of the character is replaced with the '\$' character to mark the record as deleted. Then the key is deleted from the B-Tree in memory and the index file is re-written before the main program terminates.

3.2.5 Modify a Record

In the modification process, the user selects option 5 from the menu. The user is asked to enter the key to be modified. The key's address in the record file is retrieved using the search subroutine. Then the record is marked as deleted using the same logic as the delete subroutine. Once the record is marked as deleted, the key is removed from the B-Tree in memory. Now, the user is given a choice to select which fields to modify and after the required changes, the modified record is inserted at the end of the file and the new record key and address is inserted into the B-Tree. Since the B-Tree was modified, it is required to re-write the index file before terminating the main program.

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 software-based service or component into the requirements of end users.

4.1 About C++

Here the entire project is written in the programming language “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.

4.1.1 File Handling

Files form the core of this project and are used to provide persistent storage for user entered information on disk. The `open()` and `close()` methods, as the names suggest, are defined in the C++ Stream header file `fstream.h`, to provide mechanisms to open and close files. The physical file handles used to refer to the logical filenames, are also used to ensure files exist before use and that existing files aren't overwritten unintentionally.

The 2 types of files used are data files and index files. `open()` and `close()` are invoked on the file handle of the file to be opened/closed. `open()` takes 2 parameters- the filename and the mode of access. `close()` takes no parameters.

4.2 Pseudocode

Pseudo code is a detailed yet readable which represents the programming language statements for each module which is easier to understand of our project and its main idea is to give out its functionality.

4.2.1 Inserting a New Record

This function is used to read the input from the user and validates the given ID and then prompts the user to enter the instrument id, type, description, brand, color, price, and quantity. Then it packs the record into a buffer and the buffer is appended to the file. The key and record addresses of the new record are inserted into the B-Tree.

```
void addNewInstrumentRecord(RecordFile<Instrument> &File, Instrument &instrument, BTree &tree)
{
    File.Open(ios_base::out | ios_base::app | ios_base::ate);
    cout << "\nEnter the Instrument details:\n\n";
    instrument.ReadFromStandardInput(tree);
    int recAddr = File.Write(instrument);
    cout << GREEN << "\nNew record written at address " << recAddr << "\n";
    File.Close();
    tree.insert(instrument.getID(), recAddr);
}
```

Figure 4.1 Inserting a new record

```
void BTree::insert(int key, int recAddr) {
    if (root == NULL) {
        root = new BTreeNode(t, true);
        root->Keys[0] = key;
        root->RecAddrs[0] = recAddr;
        root->n = 1;
    }
    else {
        if (root->n == (2 * t - 1)) {
            BTreeNode * s = new BTreeNode(t, false);
            s->C[0] = root;
            s->splitChild(0, root);
            int i = 0;
            if (s->Keys[0] < key)
                i++;
            s->C[i]->insertNonFull(key, recAddr);
            root = s;
        }
        else
            root->insertNonFull(key, recAddr);
    }
}
```

Figure 4.2 Insertions in a B-Tree

4.2.2 Displaying Records

Records can be displayed in two ways – Display based on entry-sequenced and Display sorted by ID.

```
void displayAllInstrumentRecordsEntrySequenced(RecordFile<Instrument> &File, Instrument &instrument)
{
    File.Open(ios_base::in);
    cout << "\nInstrument Records - Based on entry-sequence:\n\n";
    instrument.PrintHeadings();
    while (true)
    {
        if (File.Read(instrument) == -1)
            break;
        if (instrument.ID[0] != '$')
            instrument.PrintRecord();
    }
    cout << YELLOW << string(140, '-') << "\n"
         << RESET;
    File.Close();
}

void displayAllInstrumentRecordsSortedById(RecordFile<Instrument> &File, Instrument &instrument, BTree &tree)
{
    File.Open(ios_base::in);
    cout << "\nInstrument Records - Sorted by ID:\n\n";
    vector<pair<int, int>> nodes = tree.fetchAll();
    instrument.PrintHeadings();
    for (auto node : nodes)
    {
        if (File.Read(instrument, node.second) == -1)
            break;
        instrument.PrintRecord();
    }
    cout << YELLOW << string(140, '-') << "\n"
         << RESET;
    File.Close();
}
```

Figure 4.3 Displaying records

```
vector<pair<int, int>> BTreeNode::fetchAll() {
    vector<pair<int, int>> nodes;
    int i;
    for (i = 0; i < n; i++) {
        vector<pair<int, int>> temp;
        if (!leaf)
            temp = C[i]->fetchAll();
        for (auto it: temp)
            nodes.push_back(it);
        nodes.push_back(make_pair(Keys[i], RecAdrs[i]));
    }
    vector<pair<int, int>> temp;
    if (!leaf)
        temp = C[i]->fetchAll();
    for (auto it: temp)
        nodes.push_back(it);
    return nodes;
}
```

Figure 4.4 Get B-Tree keys in sorted order

4.2.3 Searching a Record

This function is used to search for a record by key. First, the key is searched in the B-Tree. If it is present, the corresponding record address is returned and a direct read from the file is performed to retrieve the record's fields.

```
int searchInstrumentRecord(int key, RecordFile<Instrument> &File, Instrument &instrument, BTree &tree)
{
    cout << MAGENTA << ITALIC << "\nSearching based on ID...\n\n"
        << RESET;
    int result = tree.search(key);
    if (result == -1)
        cout << RED << "\nRecord not found.\n"
            << RESET;
    else
    {
        File.Open(ios_base::in);
        File.Read(instrument, result);
        File.Close();
        cout << GREEN << "\nRecord found.\n"
            << RESET;
        cout << instrument << "\t";
    }
    return result;
}
```

Figure 4.5 Searching a record

```
int BTreeNode::search(int key) {
    int i = 0;
    while (i < n && key > Keys[i])
        i++;
    if (key == Keys[i])
        return RecAddrs[i];
    if (leaf == true)
        return -1;
    return C[i]->search(key);
}
```

Figure 4.6 Searching in a B-Tree

4.2.4 Deleting a Record

This function is used to delete a record by key. Initially, the record is searched using the key. If the key is found, we get the record address and seek to that position in the file and mark the record as deleted. Then we remove the key from the B-Tree.

```
int removeInstrumentRecord(int key, RecordFile<Instrument> &File, Instrument &Instrument, BTree &tree)
{
    int recAddr = searchInstrumentRecord(key, File, Instrument, tree);
    if (recAddr != -1)
    {
        cout << RED << ITALIC << "\nAre you sure you want to delete the record? [y/n]\n"
              << RESET;
        char ch;
        cin >> ch;

        if (ch == 'y' || ch == 'Y')
        {
            File.Open(ios_base::in | ios_base::out);
            File.MarkAsDeleted(recAddr);
            File.Close();
            tree.remove(key);
            cout << GREEN << "\nRecord deleted successfully.\n"
                  << RESET;
        }
        else
            return -1;
    }
    return recAddr;
}
```

Figure 4.7 Deleting a record

```
void BTree::remove(int key) {
    if (root == NULL) {
        cout << "The tree is empty\n";
    }
    root->remove(key);
    if (root->n == 0) {
        BTreeNode * temp = root;
        if (root->leaf)
            root = NULL;
        else
            root = root->C[0];
        delete temp;
    }
}
```

Figure 4.8 Deletions in a B-Tree

4.2.5 Modifying a Record

This function is used to modify a record by key. Initially, the record is searched using the key. If the key is found, we get the record address and seek to that position in the file and mark the record as deleted. Then we remove the key from the B-Tree. The user modifies the fields of the record and then the new values packed into a buffer and the buffer is appended to the file. The key is inserted back to the B-Tree.

```
int modifyInstrumentRecord(int key, RecordFile<Instrument> &File, Instrument &instrument, BTree &tree)
{
    int recAddr = searchInstrumentRecord(key, File, instrument, tree);
    int result;
    if (recAddr != -1)
    {
        File.Open(ios_base::in | ios_base::out);
        File.MarkAsDeleted(recAddr);
        File.Close();
        tree.remove(key);

        instrument.ModifyRecord();

        File.Open(ios_base::out | ios_base::app | ios_base::ate);
        result = File.Write(instrument);
        cout << GREEN << "Modified record written at address " << result << "\n"
             << RESET;
        File.Close();
        tree.insert(instrument.getID(), result);
        return result;
    }
    return recAddr;
}
```

Figure 4.9 Modifying a record

4.3 Testing

4.3.1 Unit Testing

Unit testing is a level of software testing where individual units/components of 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. The unit testing is the process of testing the part of the program to verify whether the program is working correct or not. In this part the main intention is to check the each and every 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. This testing is demonstrated in the table 4.1.

Case ID	Description	Input Data	Expected Output	Actual Output	Status
1	Opening a file to insert data	Insert option is selected	File should be opened in append mode without any errors	File opened in append mode	Pass
2	Validating Instrument ID	Try '12'	Accept the instrument ID and prompt for Instrument type.	"Enter the Instrument Type:"	Pass
3	Validating Instrument ID	Try 'x'	Invalid Instrument ID and the message "Please enter a positive integer for ID"	"Please enter a positive integer for ID"	Pass

4	Validating price	Try '3500'	Accept the price and prompt for quantity	"Enter the quantity:"	Pass
5	Validating price	Try '35000'	Invalid price and the message "Please enter a valid price"	"Please enter a valid price"	Pass
6	Validating quantity	Try '5'	Accept the quantity and write the record to the file	Record is written to the file	Pass
7	Validating quantity	Try 'mn'	Invalid quantity and the message "Please enter a valid quantity"	"Please enter a valid quantity"	Pass
8	Closing file	-	File should be closed without any error	File is closed	Pass

Table 4.1 Unit Testing for Music Instruments Management System

4.3.2 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 and is shown in table 4.2.

Case ID	Description	Input Data	Expected Output	Actual Output	Status
1	To insert a new record into the file	Enter option 1 in the menu	Display the record entry form	Displays the record entry form	Pass
2	Display all records based on entry-sequence	Enter option 2 in the menu	Display all records based on entry sequence	Displays all records based on entry sequence	Pass
3	Display all records sorted by ID	Enter option 3 in the menu	Display all records sorted by ID	Displays all records sorted by ID	Pass
4	Search for a record by key	Enter option 4 in the menu and enter the key: Try '12'	Record found and display the field values	Record found and displays the field values	Pass
5	Search for a record by key	Enter option 4 in the menu and enter the key: Try '19'	Record not found	Record not found	Pass
6	Delete a record by key	Enter option 6 in the menu and enter the key: Try '5'	Record found and mark the record as deleted upon user confirmation and update B-Tree index	Record found and marked as deleted and B-Tree index updated	Pass

7	Delete a record by key	Enter option 5 in the menu and enter the key: Try '5'	Record not found	Record not found	Pass
8	Modify a record by key	Enter option 5 in the menu and enter the key: Try '12'	Record found and mark the record as deleted and prompt user to modify fields by choice and insert new record at the end of file and update B-Tree index	Record found and marked as deleted and prompted user to modify fields by choice and new record inserted at the end of file and B-Tree index updated	Pass
9	Modify a record by key	Enter option 5 in the menu and enter the key: Try '25'	Record not found	Record not found	Pass
10	B-Tree traversal	Enter option 7 in the menu	Display B-Tree keys in sorted order	Displays B-Tree keys in sorted order	Pass
11	Exit program	Enter option 8 in the menu	Update index file and quit	Updates index file and quit	Pass

Table 4.2 Integration Testing for Music Instruments Management System

4.3.3 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 pre-requisite and is done by the testing team. System testing is done after integration testing is complete. System testing should test functional and non-functional requirements of the software and is implemented in below table 4.3.

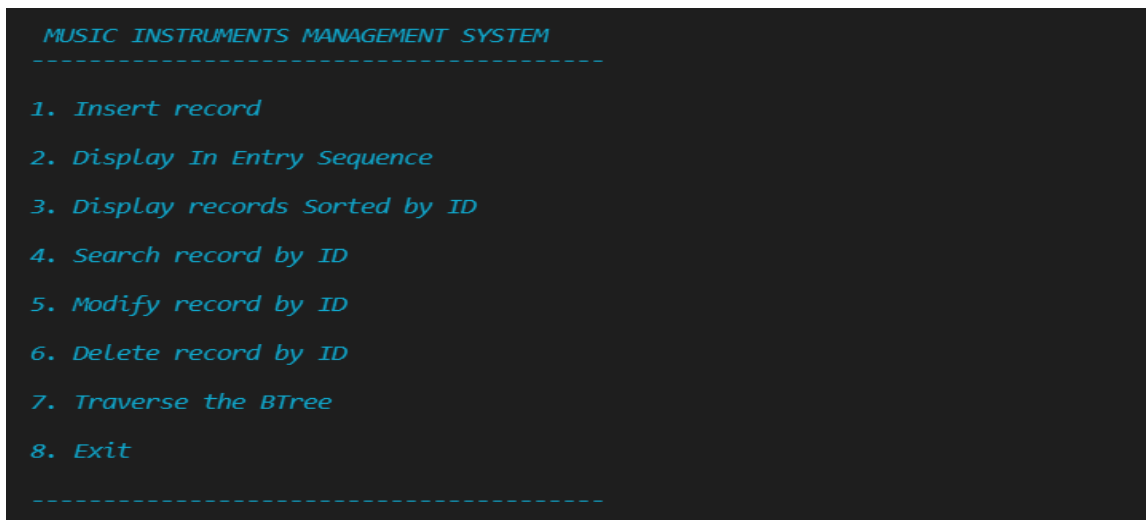
Case ID	Description	Input Data	Expected Output	Actual Output	Status
1	Display the records	Option 2	Display records which are not marked as deleted (i.e., first character is not '\$' sign)	Records not marked as deleted are displayed	Pass
2	Search a record	Option 4	Display record contents if key is found, otherwise not found	Displays record contents if key is found, otherwise not found	Pass
3	Delete a record	Option 6	Display record if key is found and confirm deletion by user, otherwise not found	Displays record if key is found and confirms deletion by user, otherwise not found	Pass
4	Modifying a record	Option 5	Display record if key is found and modify record fields, otherwise not found	Displays record if key is found and modification form is displayed, otherwise not found	Pass
5	B-Tree traversal	Option 7	Display B-Tree keys in sorted order	Displays B-Tree keys in sorted order	Pass

Table 4.3 System Testing for Music Instrument Store Management System

4.4 Discussion of Results

All the menu options provided in the project and its operations have been presented as snapshots. A detailed view of all the snapshots of the output screen is given here.

4.4.1 Menu Options



```
MUSIC INSTRUMENTS MANAGEMENT SYSTEM
-----
1. Insert record
2. Display In Entry Sequence
3. Display records Sorted by ID
4. Search record by ID
5. Modify record by ID
6. Delete record by ID
7. Traverse the BTree
8. Exit
-----
```

Figure 4.10 User Menu Screen

The Figure 4.10 shows the menu options available to the user to carry out the desired operations. If the user chooses option 1, the user will be allowed to insert a new Instrument record into the file system. On choosing option 2 or 3, the console displays the records present in a tabular format. Option 4 allows the user to search for a particular record. Option 5 allows the user to delete a record. Option 6 lets the user modify the existing details of any present records and finally option 7 displays the B Tree.

4.4.2 Insertion Operation

On selection of ‘Insert new Instrument Record’ operation from the menu, the user is prompted to enter the Instrument details such as Instrument ID, type of the Instrument, description, brand to which the Instrument belongs, color, price and the available number of Instruments of the type(quantity). The user is expected to enter a unique record with a unique ID. If the record is already present in the file, an error message is displayed and the user is prompted to insert a record again as shown in the Fig 4.11.

```

Enter your choice: 1

Enter the Instrument details:

Enter the ID: 8
Enter the Type: Keyboard
Enter the description: CTK-2550, 61 keys, Portable
Enter the Brand name : Casio
Enter the Color : Black
Enter the price: 7355
Enter the quantity: 3

New record written at address 412

```

Figure 4.11 Successful insertion of record

Once the details of the records are filled by the user, the record is written into the file, field by field and the address at which the record is written is displayed on the screen indicating a successful insertion. If a record with duplicate ID exists, the user is asked to enter a different ID.

4.4.3 Display Operation

When the user chooses option 2, the records are displayed in the order in which they were inserted into the file. The last entry in the file was an Instrument with the ID 4 (insertion shown in Figure 4.11) that is displayed at the last in the table.

```

Enter your choice: 2

Instrument Records - Based on entry-sequence:

```

ID	TYPE	DESCRIPTION	BRAND	COLOR	QUANTITY	PRICE
1	Guitar	MX125, Wood	Yamaha	Black	3	7000
3	Keyboard	PSR-F51, 61 Keys, Portable	Yamaha	Black	3	6800
2	Drum-Set	XD80USB, Electronic, LCD Display	Behringer	Silver	1	14000
4	Flute	Bamboo, C Natural, 7 holes, 19 inches	FOXIT	Kempas	5	1200
7	Acoustic Guitar	ZEB6, Brown, Wooden body, 6 strings	Juarez	Brown	3	3400

Figure 4.13 Display records – entry sequenced

When the user chooses option 3, the records are displayed with Instrument IDs sorted in the ascending order.

Enter your choice: 3

Instrument Records - Sorted by ID:

ID	TYPE	DESCRIPTION	BRAND	COLOR	QUANTITY	PRICE
1	Guitar	MX125, Wood	Yamaha	Black	3	7000
2	Drum-Set	XD80USB, Electronic, LCD Display	Behringer	Silver	1	14000
3	Keyboard	PSR-F51, 61 Keys, Portable	Yamaha	Black	3	6800
4	Flute	Bamboo, C Natural, 7 holes, 19 inches	FOXIT	Kempas	5	1200
7	Acoustic Guitar	ZEB6, Brown, Wooden body, 6 strings	Juarez	Brown	3	3400

Figure 4.14 Display records – sorted by ID

4.4.4 Search Operation

On choosing option 4, the user is requested to enter the Instrument ID of the Instrument that needs to be accessed/searched for. If the search is unsuccessful the message ‘Record not found’ is displayed. If the search is successful the message ‘Record found’ along with the details of the record requested is displayed.

```
Enter your choice: 4
Enter the key (ID) to be searched: 2
Searching based on ID...

Record found.
ID: 2
Type: Drum-Set
Description: XD80USB, Electronic, LCD Display
Brand: Behringer
Color: Silver
Quantity: 1
Price: 14000
```

Figure 4.15 Search of a record present in the file

```
Enter your choice: 4
Enter the key (ID) to be searched: 9
Searching based on ID...

Record not found.
```

Figure 4.16 Search of a record not present in the file

4.4.5 Delete Operation

The Figure 4.17 shows the delete operation. When the instrument ID that has to be deleted is entered by the user, a search operation is done first to see if the record to be deleted exists in the file. If it is present, the details of the record are displayed along with a confirmation message. The user is prompted to press yes or no to delete the record.

```
Enter your choice: 6
Enter the key (ID) to be deleted: 2
Searching based on ID...
Record found.
ID: 2
Type: Drum-Set
Description: XD800USB, Electronic, LCD Display
Brand: Behringer
Color: Silver
Quantity: 1
Price: 14000
Are you sure you want to delete the record? [y/n]
y
Record deleted successfully.
```

Figure 4.17 Deleting a record

4.4.6 Modify Operation

Modify operation allows the user to modify a field of the user's choice. By giving the ID of the record that is to be modified, the user is prompted with the options to modify any of the fields. On choosing a particular field to modify, the user can enter the new details of the field. The menu options of the fields are displayed again after successful modification. The user can either continue changing other fields or save the changes by choosing option 7. The new/modified record is written in a new address in the file.

```
Enter your choice: 5
Enter the key (ID) to be modified: 3
Searching based on ID...

Record found.
ID: 3
Type: Keyboard
Description: PSR-123, 61 keys, Portable
Brand: Yamaha
Color: Black
Quantity: 2
Price: 7100

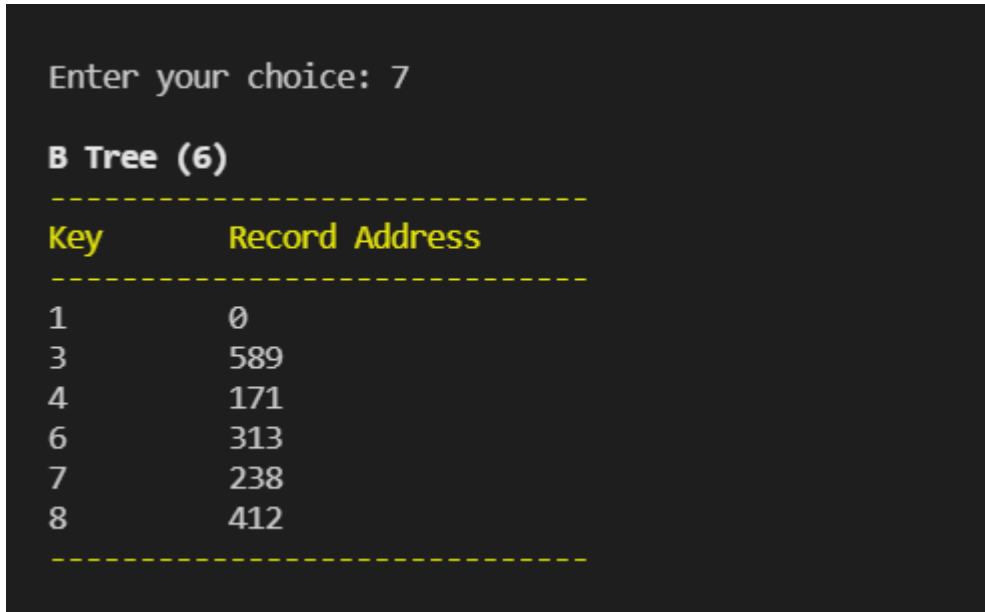
Which field would you like to modify?
1. ID  2. Type 3. Description  4. Brand    5. Color    6. Quantity  7. Price    8. Back to main menu
Enter the field no. : 7
Enter the new value: 6050

Which field would you like to modify?
1. ID  2. Type 3. Description  4. Brand    5. Color    6. Quantity  7. Price    8. Back to main menu
Enter the field no. : 8
Applying modifications...Modified record written at address 648
```

Figure 4.18 Modifying a record

4.4.7 B-Tree Traversal

On choosing option 7 to traverse the B-Tree, the entire B-tree is displayed to the user. The nodes of the tree that contains the keys and addresses of each record in the file are displayed along with the order of the B-Tree.



```
Enter your choice: 7

B Tree (6)
-----
Key      Record Address
-----
1        0
3        589
4        171
6        313
7        238
8        412
-----
```

The screenshot shows a terminal window with a dark background. At the top, it says "Enter your choice: 7". Below that, it says "B Tree (6)". Then, there is a table with two columns: "Key" and "Record Address". The table is enclosed in dashed lines. The data rows are: 1 (0), 3 (589), 4 (171), 6 (313), 7 (238), and 8 (412).

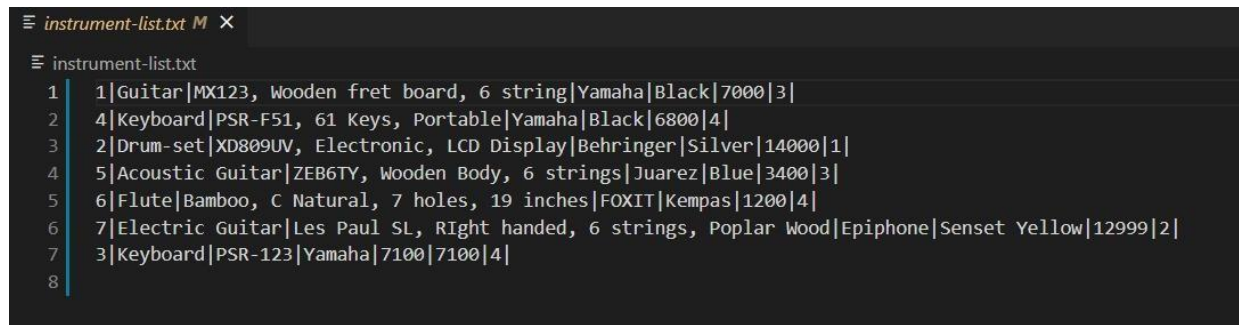
Key	Record Address
1	0
3	589
4	171
6	313
7	238
8	412

Figure 4.19 B-Tree traversal

4.4.8 File Contents – Record File & Index File

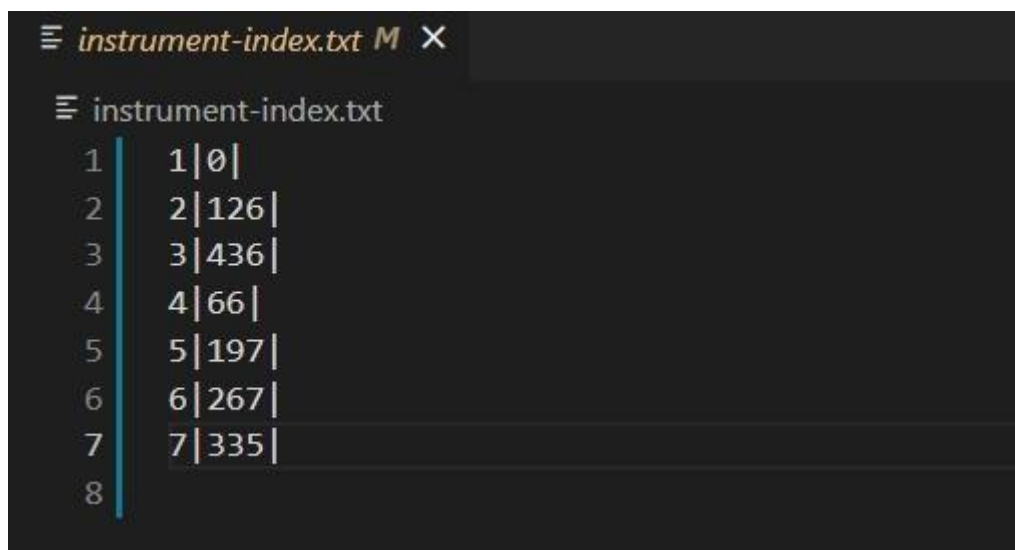
Figure 4.20 shows the records present the data file – ‘Instrument.txt’ where each field in a record is separated by the delimiter ‘|’ and each record is separated by a newline character ‘\n’.

Figure 4.21 shows the contents of the index file = ‘btreeindex.txt’ that stores the key and address of each record in the B-Tree.



```
instrument-list.txt M X
instrument-list.txt
1 1|Guitar|MX123, Wooden fret board, 6 string|Yamaha|Black|7000|3|
2 4|Keyboard|PSR-F51, 61 Keys, Portable|Yamaha|Black|6800|4|
3 2|Drum-set|XD809UV, Electronic, LCD Display|Behringer|Silver|14000|1|
4 5|Acoustic Guitar|ZEB6TY, Wooden Body, 6 strings|Juarez|Blue|3400|3|
5 6|Flute|Bamboo, C Natural, 7 holes, 19 inches|FOXIT|Kempas|1200|4|
6 7|Electric Guitar|Les Paul SL, Right handed, 6 strings, Poplar Wood|Epiphone|Senset Yellow|12999|2|
7 3|Keyboard|PSR-123|Yamaha|7100|7100|4|
8
```

Figure 4.20 Record file contents



```
instrument-index.txt M X
instrument-index.txt
1 1|0|
2 2|126|
3 3|436|
4 4|66|
5 5|197|
6 6|267|
7 7|335|
8
```

Figure 4.21 Index file contents

Chapter 5

CONCLUSIONS AND FUTURE ENHANCEMENTS

5.1 Conclusion

Instrument store management system is specially designed for the purpose of adding daily Instrument records in an Instrument store. This system makes easy for storing records as it is not time consuming. This file management provides efficient handling of the available Instrument records in the store which otherwise would be tedious to maintain a manual records at the store. This project focuses on providing the user the ability to view the details of the available Instrument collection, manipulate the records and also add or delete any record as per the requirements.

5.2 Future Enhancements

- Multilevel indexing can be enhanced by using B+ tree data structure.
- Furthermore, hashing can be used to access data by primary key in $O(1)$ time, whereas tree algorithms can take up to $O(\log n)$ time.
- Secondary indexing can be implemented as well to access Instrument records by their type, model, price, etc.

REFERENCES

- [1] Michael J. Folk, Bill Zoellick, Greg Riccardi: File Structures: An Object-Oriented Approach with C++, 3rd Edition, Pearson Education, 1998
- [2] www.cplusplus.com/reference/
- [3] en.cppreference.com/w/
- [4] www.programiz.com
- [5] www.stackoverflow.com
- [6] www.geeksforgeeks.org