

Chapter 1

Introduction

The Shares and Stock details mini project provides a user-friendly, interactive Menu Driven Interface (MDI). All data is stored in files for persistence. The system uses an Index file to store the primary index and a Data file to store records pertaining to each customer.

1.1 Introduction to File Structures

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

1.1.1 History

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

The indexes made it possible to keep a list of keys and pointers in a smaller file that could be searched more quickly. With key and pointer, the user had direct access to the large, primary file.

But simple indexes had some of the same sequential flaws as the data file, and as the indexes grew, they too became difficult to manage, especially for dynamic files in which the set of keys changes.

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 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-Trees 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 program relies on the OS to take care of the details of the telephone switching system. It could be that bytes coming down the line into the program originate from a physical file they come from the keyboard or some other input device. Similarly, bytes the program sends down the line might end up in a file, or they could appear on the terminal screen or some other output device. Although the program doesn't know where the bytes are coming from or where they are going, it does know which line it is using. This line is usually referred to as the logical file, to distinguish it from the physical files on the disk or tape.

1.1.3 Various Kinds of storage of Fields and Records

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

➤ Field Structures

The four most common methods as shown in Figure. 1.1 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 this method, 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

a)

Ames	Mary	123 Maple	Stillwater	OK74075
Mason	Alan	90 Eastgate	Ada	OK74820

b)

04Ames04Mary09123 Maple10Stillwater02OK0574075
05Mason04Alan1190 Eastgate03Ada02OK0574820

c)

Ames Mary 123 Maple Stillwater OK74075
Mason Alan 90 Eastgate Ada OK 74820

d)

Last=Ames first=Mary address=123 Maple city=Stillwater state=OK zip=74075

Figure 1.1 Four methods for Field Structure

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 as shown in Figure 1.2 and Figure 1.3 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, 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 1.2 below, we have 6 contiguous fields and we can recognize fields simply by counting the fields modulo 6.

a)

Ames	Mary	123 Maple	Stillwater	OK74075
Mason	Alang	90 Eastgate	Ada	OK74820

b)

Ames Mary 123 Maple Stillwater OK74075 ←unused space→
Mason Alang 90 Eastgate Ada OK 74820 ← unused space →

c)

Ames Mary 123 Maple Stillwater OK 74075 Mason Alang 90 Eastgate Ada OK. . .
--

Figure 1.2 Making Records Predictable number of Bytes and Fields

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. It is shown in figure 1.3.

Method 4: Use an Index to Keep Track of Addresses

We can use an index to keep a byte offset for each record in the original file. The byte offset allows us to find the beginning of each successive record and compute the length of each record. We look up the position of a record in the index, then seek to the record in the data file. It is shown in figure 1.3.

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. It is shown in figure 1.3.

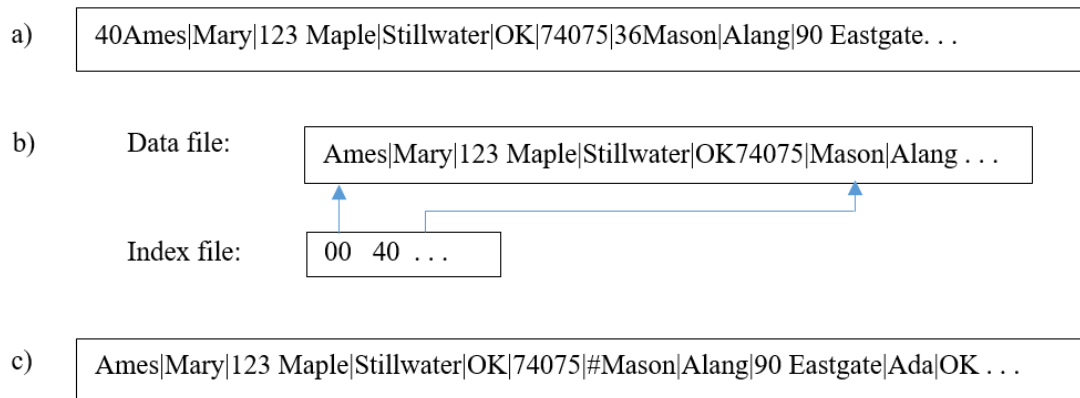


Figure 1.3 Using Length Indicator, Index and Record Delimiters

1.1.4 Application of File Structure

Relative to other parts of a computer, disks are slow. One can pack thousands of megabytes on a disk that fits into a notebook computer.

The time it takes to get information from even relatively slow electronic random-access memory (RAM) is about 120 nanoseconds. Getting the same information from a typical disk takes 30 milliseconds. So, the disk access is a quarter of a million times longer than a memory access. Hence, disks are very slow compared to memory. On the other hand, disks provide enormous capacity at much less cost than memory. They also keep the information stored on them when they are turned off.

Chapter 2

System Analysis

2.1 Analysis of Program

This mini project is used by the users to view current details of shares held by customers. The system is initially used to add records containing the details by entering the required data. This system can then be used to search, delete, modify or display existing records. We can also see the arrangement of the record by viewing the tree structure of the record

2.2 Structure used to Store the Fields and Records

Various types of structures are used to store fields and records in a file. The structures used in this mini project are explained below.

2.2.1 Storing Fields

Fixing the Length of Fields

In this System, the share_id field is a character array that can hold a string value of fixed size. Other fields are also character arrays. Here, share_id is the primary key which can be used to retrieve the entire record.

Separating the Fields with Delimiters

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

2.2.2 Storing Records

Making Records a Predictable Number of Fields

In this system, we have a fixed number of fields, each with a maximum length, that combine to make a data record. 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.

Using an Index to Keep Track of Addresses

We use a B-Tree of indexes to keep byte offsets for each record in the original file. The byte offsets allow us to find the beginning of each successive record and compute the length of each record.

Placing a Delimiter at the End of Each Record

Our choice of a record delimiter for the data files is the end-of-line (new-line) character (`\n`).

2.3 Operations Performed on a File Insertion

Insert

The system is initially used to add records containing the `share_id`, `cust_name`, `comp_name`, `no_of_shares` and `share_value`. During the insertion of the record if the entered `share_id` is already used and if the person trying to use it again, the system is designed to prompt the user that duplicate `share_id` cannot be used. The length of the `share_id` is checked to see whether it contains only 10 characters. If not, an error message will be displayed.

Display

The system can then be used to display all existing records in the file. The records can be displayed, either based on the ordering of `share_id` maintained in the B-Tree of indexes, which here is, an ascending order or in the order of their entries. Only records with references in the B-Tree of indexes are displayed.

Search

The system can then be used to search for existing records in the file. The user is prompted for a `share_id`, which is used as the key in searching for records in the B-Tree of indexes. The B-Tree is searched to obtain the desired starting byte address, which is then used to seek to the desired

data record in any of the files.

The details of the requested record, if found, are displayed, with suitable headings on the user's screen. If absent, a "record not found" message is displayed to the user.

Delete

The system can then be used to delete existing records from the file. The reference to a deleted record is removed from index file as well. The requested record, if found, is marked for deletion, a "record deleted" message is displayed, and the reference to it is removed from the B-Tree. If absent, a "record not found" message is displayed to the user.

Update

The user can also update existing records. First, share_id is to be entered which is used to search for the record. If found, the reference to a deleted record is removed from index file as well as the data file. Then the user is prompted to enter new values for the record to be updated. If absent, a "record not found" message is displayed to the user. After insertion of the record it will store in the sorted manner in the B-tree.

Tree structure of the record

This is the sorted formats of the records, here the system is able to show how records are arranged in the B-tree. It also gives the details about the leaf node in which record is stored. In this system share_id is used as the primary key to store the record in the B-tree. During insertion or deletion of the records we can notice the changes in the tree structure.

2.4 Indexing Used

B-Tree

A B-Tree of simple indexes on the primary key is used to provide direct access to data records. Each node in the B-Tree consists of a primary key with the reference to record. The primary key field is the share_id field while the reference field is the starting byte offset of the matching

record in the data file. Each B-Tree node can have max of 2 child node.

The share_id is stored in the B-Tree, and hence written to an index file. On retrieval the matched share_id is used, before it is used to seek to a data record, as in the case of requests for a search, delete, modify operation. As records are added, nodes of the B-Tree undergo splitting (on detection of overflow), merging (on detection of underflow) or redistribution (to improve storage utilization), in order to maintain a balanced tree structure.

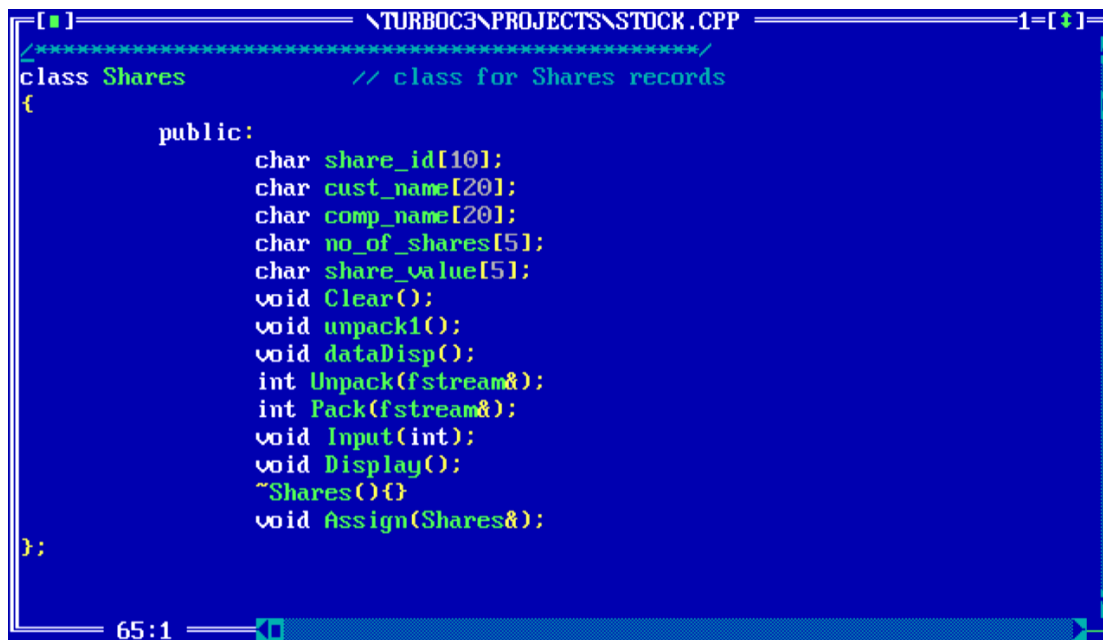
The data files are entry-sequenced, that is, records occur in the order they are entered into the file. The contents of the index files are loaded into their respective B-Trees, prior to use of the system, each time. Each B-Tree is updated as requests for the available operations are performed. Finally, the B-Trees are written back to their index files, after every insert, delete and modify operation.

Chapter 3

System Design

3.1 Design of Fields and Records

The share_id is declared as a character array that can hold a maximum of 10 characters. Checks are done to ensure the share_id is of exactly 10 characters during input. Cust_name, comp_name, no_of_shares and share_value are declared as character arrays. Each field is separated by a vertical bar character (|) and finally by a new line character to show the end of each record. The class declaration of a typical Stock file record is as shown in Figure 3.1.



```
class Shares // class for Shares records
{
public:
    char share_id[10];
    char cust_name[20];
    char comp_name[20];
    char no_of_shares[5];
    char share_value[5];
    void Clear();
    void unpack1();
    void dataDisp();
    int Unpack(fstream&);
    int Pack(fstream&);
    void Input(int);
    void Display();
    ~Shares(){}
    void Assign(Shares&);
};
```

Figure 3.1 Class Shares

3.2 User Interface

The User Interface or UI refers to the interface between the program and the user. Here, the UI is menu-driven i.e., a list of options (menu) is displayed to the user and the user is prompted to enter an integer corresponding to the choice, that represents the desired operation to be performed as shown in Figure 3.2.



Figure 3.2 User Menu Screen

3.2.1 Insertion of a Record

If the operation desired is Insertion, the user is required to enter 2 as his/her choice, from the menu displayed, after which a new screen is displayed. Next, the user is prompted to enter the share_id in which the record is to be inserted followed by the cust_name, comp_name, no_of_shares and share_value. The user is prompted for each input until the value entered meets the required criterion for each value. This means that, the user is prompted for the share_id, until the user enters 10 characters of the format.

3.2.2 Display of a Record

If the operation desired is Display of Records, the user is required to enter 7 as his/her choice, from the menu displayed, after which a new screen is displayed. If there are no records in the file “no records found” message is displayed. For the files with at least 1 record, each share_id, followed by the details of each record within the file, with suitable headings, is displayed. In each case, the user is then prompted to press any key to return back to the menu screen. There is also an option for Display of the nodes for which the user is required to enter 6 as his/her choice.

3.2.3 Deletion of a Record

If the operation desired is Deletion, the user is required to enter 4 as his/her choice, from the menu displayed, after which a new screen is displayed. Next, the user is prompted to enter the share_id of the record to be deleted. If there are no records in the file, a “no records to delete” message is displayed, and the user is prompted to press any key to return back to the menu screen. The share_id entered is used as a key to search for a matching record. If none is found, a “record not found” message is displayed. If one is found, a “record deleted” message is displayed. In each case, the user is then prompted to press any key to return back to the menu screen.

3.2.4 Searching of a Record

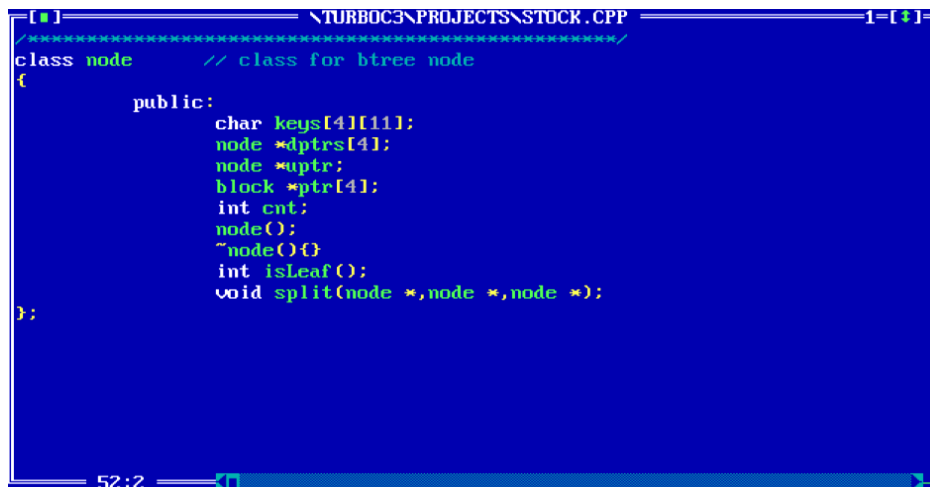
If the operation desired is Search, the user is required to enter 3 as his/her choice, from the menu displayed, after which a new screen is displayed. Next, the user is prompted to enter the share_id of the record that is to be searched for. The share_id entered is used as a key to search for a matching record. If none is found, a “record not found” message is displayed. If one is found, the details of the record, with suitable headings, are displayed. In each case, the user is then prompted to press any key to return back to the menu screen.

3.2.5 Modification of a Record

If the operation desired is Update, the user is required to enter 5 as his/her choice, from the menu displayed, after which a new screen is displayed. Next, the user is prompted to enter the share_id of the record that is to be updated. The Update operation is implemented as a deletion followed by an insertion. The share_id entered is used as a key to search for a matching record. If none is found, a “record not found” message is displayed and the user is then prompted to press any key to return back to the menu screen. If one is found, a “record deleted” message is displayed, after which a new screen is displayed. Next, the user is prompted to enter the share_id, followed by all the other fields.

3.2.6 Design of Index

The B-Tree is declared as a class, an object of which represents a node in the B-Tree. Each node contains a count of the number of entries in the node and a reference to each descendant. The maximum number of descendants is 4 while the minimum is 2. Each node has an array of objects, each an instance of the class type index as shown in Figure. 3.3. Each object of type “index” contains a share_id field and an address field. The contents of the B-Tree are written to the index file on disk after each insert, delete and modify operation. To ensure efficient space utilization, and, to handle the conditions of underflow and overflow, nodes may be merged with their siblings, or split into 2 descendants, thereby creating a new root node for the new nodes created, or, entries within nodes maybe shifted to save space.

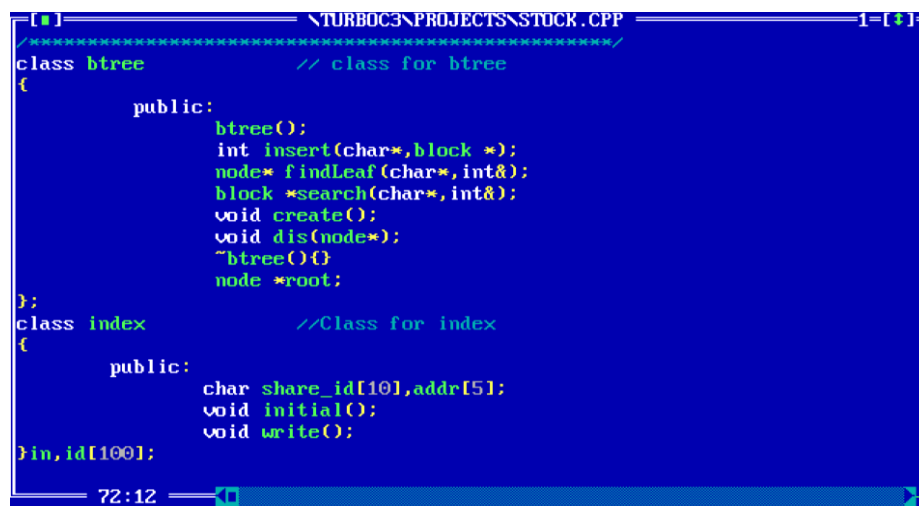


```

\\TURBOC3\\PROJECTS\\STOCK.CPP
//=====
class node      // class for btree node
{
public:
    char keys[4][11];
    node *dptrs[4];
    node *uptr;
    block *ptr[4];
    int cnt;
    node();
    ~node();
    int isLeaf();
    void split(node *, node *, node *);
};
52:2

```

Figure 3.3 Class Node



```

\\TURBOC3\\PROJECTS\\STOCK.CPP
//=====
class btree     // class for btree
{
public:
    btree();
    int insert(char*, block *);
    node* findLeaf(char*, int&);
    block* search(char*, int&);
    void create();
    void dis(node*);
    ~btree();
    static node* root;
};

class index     //Class for index
{
public:
    char share_id[10], addr[5];
    void initial();
    void write();
};
72:12

```

Figure 3.4 Class B Tree and Index

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++

4.1.1 Classes and Objects

A book file is declared as an object with the `share_id`, `cust_name`, `comp_name`, `no_of_shares` and `share_value` as its data members. An object of this class type is used to store the values entered by the user, for the fields represented by the above data members, to be written to the data file. The B-Tree is declared as the class `b-tree node`, an object of which represents a node in the B-Tree. Each node also has an array of objects, each an instance of the class type `index`. Each object of type `index` contains a `share_id` field and an `address` field, which stores the address at which the corresponding data record is stored in the data file. Class objects are created and allocated memory only at runtime.

4.1.2 Dynamic Memory Allocation and Pointers

Memory is allocated for nodes of the B-Tree dynamically, using the method `malloc()`, which returns a pointer (or reference), to the allocated block. Pointers are also data members of objects of type `b-tree node`, and, an object's pointers are used to point to the descendants of the node represented by it. File handles used to store and retrieve data from files, act as pointers. The `free()` function is used to release memory, that had once been allocated dynamically. `malloc()` and `free()` are defined in the header file `alloc.h`.

4.1.3 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 two 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 two parameters, the filename and the mode of access. `close()` takes no parameters. The two types of files used are data files and index files.

4.1.4 Character Arrays and Character functions

Character arrays are used to store the BOOKID fields to be written to data files and stored in a B-Tree index object. Character functions are defined in the header file `cctype.h`. Some of the functions used include:

- `itoa()` – to store ASCII representations of integers in character arrays
- `isdigit()` – to check if a character is a decimal digit (returns non-zero value) or not (returns zero value)
- `isalpha()` - to check if a character is an alphabet (returns non-zero value) or not (returns zero value)
- `atoi()` – to convert an ASCII value into an integer.

4.2 Pseudocode

4.2.1 Insertion Module

The insertion operation is implemented by `append()` function which adds index objects to the B-Tree if the `share_id` entered is not a duplicate. Values are inserted into a node until it is full, after which the node is split and a new root node is created for the 2 child nodes formed. It makes calls to other recursive and non-recursive functions. Pseudo code for insertion of records is given below.

```
void append()                                //Function to add record
{
    shares std;

    int flag=1, pos;

    fstream file("shares.txt",ios::app);

    std.Input();                             //Transfers control to function that reads inputs

    file.seekp(0,ios::end);

    pos=file.tellp();

    flag=s.Insert(std.share_ID,pos);         //Inserts record into position pos

    if(flag && std.Pack(file)){

        for(i=indsize;i>0;i--)

        {

            if(strcmp(std.share_ID,id[i-1].ishare_ID)<0)

                id[i]=id[i-1];

            else

                break;

        }

        strcpy(id[i].ishare_ID,std.share_ID);

        itoa(pos,id[i].addr,10);

        indsize++;

        cout << "\n\t Done...\n";          //Return this if insertion is successful

    }

    else
```

```
        cout << "\n\t Failure.";           //Return this if record is not inserted successfully

    file.clear();

    file.close();

}
```

4.2.2 Display Module

The seqdisp() function retrieves the records stored in data file and displays them in entry sequence order. Pseudo code for this function is shown below.

```
void shares::seqdisp(){                               //Function to display in entry sequence order

    shares std;

    fstream file("shares.txt",ios::in);               //Open file in output mode

    file.seekg(0,ios::beg);

    cout<<setiosflags(ios::left)<<setw(15)<<"ShareID"<<setw(15)<<"Customer"<<setw(15);

    cout<<"Company"<<setw(15)<<"Share Value"<<setw(15)<<"No_of_shares";

    cout<<"-----"<<endl;

    while(!file.eof()){

        std.Unpack(file);                             //Retrieve data from file to memory

        cout<<setw(15)<<std.share_ID<<setw(15)<<std.cust_name<<setw(15)<<std.comp_name;

        cout<<setw(15)<<std.share_value<<setw(15)<<std.no_of_shares<<endl;

    }

    file.close();

}
```

4.2.3 Deletion Module Pseudocode

The delrec() function deletes records from the data file and reflects the same in the B tree. It makes calls to other recursive and non-recursive functions. The pseudo code for deletion module is shown below.

```
void delrec(char *key)                                //Delete function to delete based on key
{
    int r=0,found=0,s;
    char del='N';
    shares stds[100],std;
    fstream file("shares.txt",ios::in);
    file.seekg(0,ios::beg);
    while(!file.fail())
        if(std.Unpack(file))
            if(strcmpi(std.share_ID,key)==0)
            {
                found=1;
                cout<<" \n Record :";
                std.Display(); //Display the record to be deleted
                cout<<"\n\n Confirm permanent deletion:[Y/N]";
                cin>>del;    //Confirm deletion
                if(!(del=='Y' || del=='y'))
                {
                    stds[r].Clear();
                    stds[r++].Assign(std);
                }
            }
            else            //If deletion is confirmed
            {
                int pos=ind_search(std.share_ID);
                for(i=pos;i<indsize;i++)
```

```
                id[i]=id[i+1];
                indsize--;
                cout<<"\n\n\t Deleted : ";
            }
        }
    Else                //If key is not found
    {
        stds[r].Clear();
        stds[r++].Assign(std);
    }
    file.clear();
    if(found==0)
        cout<<"\n\n\t Record not found.";
    else
    {
        file.close();
        file.open("shares.txt",ios::out);
        file.seekp(0,ios::beg);
        for(s=0;s<r;s++)
            if(!(stds[s].Pack(file)))
                continue;
        file.clear();
    }
    file.close();
}
```

4.2.4 Search Module Pseudocode

The search() function traverses the B-Tree, based on the values of objects in the root node according to the key specified. It returns the records that matches the key. Pseudo code for search function is given below.

```
void search(char *key)                                //Search function based on key
{
    shares std;
    int found=0,i;
    block *dp;
    fstream file("shares.txt",ios::in);
    file.seekg(ios::beg);
    dp=bt.search(key,found);                          //Search B tree for record
    if(found==0)
        cout<<"\n\n\t Record not found...\n";
    else                                              //If record is found
    {
        found=0;
        for(i=0;i<dp->cnt;i++)
            if(strcmpi(key,dp->keys[i])==0)
            {
                found = 1;
                file.seekg(dp->disp[i],ios::beg);
                std.Unpack(file);
                cout<<"\n\n\t Record found : ";
                std.Display(); //Display the searched record
            }
        if(found==0) cout<<"\n\n\t Record not found ";
    }
    file.clear();
    file.close();
}
```

4.2.5 Modify Module Pseudocode

The modify operation is implemented as a call to the delrec() function followed by a call to the append(). The update() function updates records based on their key. Pseudo code for this function is shown below.

```
void update(char *key)                //Update function based on key
{
    shares stds[100],std;
    int f=0,found=0,g;
    char upd='n';
    fstream file("shares.txt",ios::in);
    file.seekg(0,ios::beg);
    while(!file.fail())
        if(std.Unpack(file))
            if(strcmpi(std.share_ID,key)==0)
            {
                found=1;
                cout<<"\n\tRecord:";
                std.Display(); //Display record to be modified
                cout<<"\n\n Confirm permanent updation:[Y/N] ";
                cin>>upd;
                if(!(upd=='Y' || upd=='y'))
                {
                    stds[f].Clear();
                    stds[f++].Assign(std);
                }
            }
            Else                //If modification is confirmed
            {
                int pos=ind_search(std.share_ID);
                for(i=pos;i<indsize;i++)
```

```
        id[i]=id[i+1];
        indsize--;
        file.seekp(0,ios::end);
        pos=file.tellp();
        cout << "\n\t Enter the new record :\n";
        std.Input();    //Read new details
        for(i=indsize;i>0;i--)
        {
            if(strcmp(std.share_ID,id[i-1].ishare_ID)<0)
                id[i]=id[i-1];
            else
                break;
        }
        strcpy(id[i].ishare_ID,std.share_ID);
        itoa(pos,id[i].addr,10);
        indsize++;

        stds[f].Clear();
        stds[f++].Assign(std);
    }
}
else
{
    stds[f].Clear();
    stds[f++].Assign(std);
}
file.clear();
if(found==0)
    cout<<"\n\n\t Record not found...\n";
else
{
```



```

        file.close();
        file.open("shares.txt",ios::out);
        file.seekp(0,ios::beg);
        for(g=0;g<f;g++)
            if(!(stds[g].Pack(file))) continue;
            file.clear();
    }
    file.close();
    delete head;
    delete bt.root;
    head = new block;
    bt.root = new node;
    s.create();
}

```

4.2.6 Indexing Pseudocode

The B-Tree of indexes is written to the index file after every insertion, deletion, and modification operation, using the write() and initial() functions to keep index file up-to-date and consistent. Pseudo code for these functions is shown below.

```

void index::initial()                //Function to initialize index file contents
{
    indfile.open(indexfile,ios::in);
    if(!indfile)                    //If there is no file
    {
        indsize=0;
        return;
    }
    for(indsize=0;;indsize++)        //If file already exists
    {
        indfile.getline(id[indsize].ishare_ID,15,'');
    }
}

```

```

        indfile.getline(id[indsize].addr,5,'\n');
        if(indfile.eof())
            break;
    }
    indfile.close();
}

void index::write()                //Function to write to index file
{
    indfile.open(indexfile,ios::out);
    for(i=0;i<indsize;i++)
        indfile<<id[i].ishare_ID<<"|"<<id[i].addr<<"\n";
    indfile.close();
}

```

4.3 Testing

4.3.1 Unit Testing

The unit testing is the process of testing the part of the program to verify whether the program is working correct or not. In this part the main intention is to check the each and every 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.

Testing for share_id:

Table 4.1 Unit Test Case for share_id input check

Sl no. of test case	1
Name of test case	Check test
Item/Feature being tested	Input for share_id field
Sample Input	Share_id = 'TDS98'
Expected Output	Display a message saying "Invalid share_id"
Actual Output	Invalid share_id
Status	Pass

Testing for Customer_Name:

Table 4.2 Unit Test Case for Customer_Name input check

Sl no. of test case	2
Name of test case	Check test
Item/Feature being tested	Input for customer_ame field
Sample Input	Name = 'Ri45ya'
Expected Output	Display a message saying "Invalid customer_name"
Actual Output	Invalid customer_name
Status	Pass

4.3.2 Functional Testing

Functional testing is a software testing process used within software development in which software is tested to ensure that it conforms to all requirements. Functional testing is a way of checking software to ensure that it has all the required functionality that's specified within its functional requirements. It is not concerned with how processing occurs, but rather, with the results of processing. During functional testing, Black Box Testing technique is used in which the internal logic of the system being tested is not known to the tester.

Table 4.3 Functional testing Test Cases for Shares and Stocks management

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 error message	File is opened in append mode	Pass
2.	Validating share_id	AS12	Accept the share_id and prompt for customer_name will be displayed	"Enter Customer_Name :"	Pass

Case_id	Description	Input Data	Expected Output	Actual Output	Status
3.	Validating share_id	1234	Invalid share_id and the message “Re-enter share_id:” will be displayed	“Re-enter share_id:”	Pass
4.	Enter share_id: Enter cust_name: Enter comp_name: Enter share_val: Enter no_of_shares:	AS12 Tejas Amazon 2500 100	It should accept the fields and return to main menu	“Press any key to return to main menu”	Pass
5.	File Updation	-	It should pack all the fields and append them to the data file	Data will be updated both in data and index file	Pass
6.	Closing the file	-	File should be closed without any error	File is closed	Pass

4.3.3 Integration Testing

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

Table 4.4 Integration testing Test Case for Shares and Stocks management

Case_id	Description	Input data	Expected Output	Actual Output	Status
1.	To display the entered records of the data file	Enter the option 7 in the menu	Display the record one after the other	Display all records	Pass
2.	To add the new records into the data file	Enter the option 2 In the menu	Display the record entry form	Display the record entry form	Pass
3.	To search for a particular record in the file	Enter the option 3 in the menu and should enter invalid share_id	Record not found.	Record not found.	Pass
4.	To search for a particular record in the file	Enter the option 3 in the menu and should enter the share_id	'Record is found' and it will display the contents of the searched record.	Same as expected output.	Pass
5.	To delete a particular record in the file	Enter the option 4 in the menu and should enter the invalid share_id	Record not found.	Record not found.	Pass

Case_id	Description	Input data	Expected Output	Actual Output	Status
6.	To delete a particular record in the file	Enter the option 4 in the menu and should enter the share_id	Delete the record and both data file and index file will updated.	Same as expected output.	Pass
7.	To update a particular record in the file	Enter the option 5 in the menu and should enter the invalid share_id	Record not found	Record not found	Pass
8.	To update a particular record in the file	Enter the option 5 in the menu and should enter the share_id	'Record is found' and delete the record and allow the users to re-enter the record.	Same as expected output.	Pass
9.	To display the all entered records in the data file	Enter the option 1 in the menu	Display all the saved records	Display all the saved records	Pass
10.	To display the tree structure of the entered records in the data file	Enter the option 6 in the menu	Display the tree structure of the records	Display the tree structure of the records	Pass
11.	To quit the program	Enter the option 8	Exit the program	Exit the program	Pass

4.3.4 System Testing

System testing is defined as testing of a complete and fully integrated software product. System testing is done after integration testing is complete.

Table 4.5 System testing Test Case for Shares and Stocks management

Case_id	Description	Input data	Expected output	Actual output	Status
1.	To display the records	Display records for valid share_id (present) and invalid share_id (not present)	Record is displayed for valid and record not found for invalid	Record is displayed for valid and record not found for invalid	Pass
2.	To search the record	Search records for valid share_id (present) and invalid share_id (not present)	Record is displayed for valid and record not found for invalid	Record is displayed for valid and record not found for invalid	Pass
3.	To delete the records	Delete records for valid share_id (present) and invalid share_id (not present)	Record is deleted for valid and record not found for invalid	Record is deleted for valid and record not found for invalid	Pass
4.	To update the records	Update records for valid share_id (present) and invalid share_id (not present)	Record is updated for valid and record not found for invalid	Record is updated for valid and record not found for invalid	Pass

4.3.5 Acceptance Testing

Acceptance testing, a testing technique performed to determine whether or not the software system has met the requirement specifications. The main purpose of this test is to evaluate the system's compliance with the business requirements and verify if it has met the required criteria for delivery to end users.

There are three important forms of acceptance testing: **Alpha testing**, **Beta testing** and **User acceptance testing**.

Alpha testing is a type of acceptance testing performed to identify all possible issues or bugs before releasing the product to everyday users or the public. The main aim is to carry out tasks that a typical user might perform.

Beta testing is where a beta version of the software is released to a limited number of end users of the product to obtain feedback of the product quality. Beta testing reduces the risk of product failure and provides increased quality of the product through customer validation.

A Sample report on User acceptance testing is shown below:

Owners and their contacts:

Owner Name	Email	Phone	Role
Shreyas DL	dlshtreyas30@gmail.com	+91 8105769691	Program Manager
Tejas DS	tdsleo98@gmail.com	+91 8884448824	System Test Lead

Sign Offs

Phase	Name of the Testers	Date	Signature with Status
User Acceptance Test Release	Kishore Shivaji, User Manager	20/05/2019	
	Scott Wilson, Customers Handler	21/05/2019	

Revision History

Date	Reason for change(s)	Author(s)
18/05/2019	Expecting more User friendliness	Kishore Shivaji

4.4 Discussion of Results

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

4.4.1 Menu Options

Figure 4.1 is a snapshot of the menu screen which has the list of options that give suggestion to the user of the system.



Figure 4.1 User Menu Screen

4.4.2 Insertion

Figure 4.2 is a snapshot of the insertion screen which gives the data insertion format to the user of the system to insert new records.

```
*****
*  ADD RECORD INTO THE FILE  *
*****

Share ID      : AK12
Customer Name  : Akash Shukla
Company Name   : ITC
Share_value   : 300.75
No_of_shares   : 3000
Done...
```

Figure 4.2 Insertion of a Record

4.4.3 Deletion

Figure 4.3 is a snapshot of the deletion screen to delete the unwanted records. To delete the record, user has to specify valid share_id.

```
*****
*  DELETE RECORD  *
*****

Enter the Share_ID to delete : UN12
Record :
Share ID      : UN12
Customer Name  : Venkatesh
Company Name   : TATA MOTORS
Share Value   : 168.75
No_of_shares   : 3000
Confirm permanent deletion:[Y/N]Y
Deleted :
```

Figure 4.3 Deletion of a Record

4.4.4 Searching a Record

Figure 4.4 is a snapshot of the search screen to search for a record in the data file. To search for a particular record, user has to provide the valid share_id.

```
*****
*  SEARCH FOR RECORD USING B-TREE  *
*****

Enter the share_ID to search : AK12

Record found :
Share ID      : AK12
Customer Name  : Akash Shukla
Company Name   : ITC
Share Value    : 300.75
No_of_shares   : 3000
```

Figure 4.4 Search for a Record

4.4.5 Modification

Figure 4.5 and Figure 4.6 are snapshots of the modification screen to update the existing record. To update the record, we have to provide valid share_id of the record.

```
*****
*  UPDATE RECORD  *
*****

Enter the Share ID to update : RS12

Record:
Share ID      : RS12
Customer Name  : Rakesh Sharma
Company Name   : TATA STEEL
Share Value    : 523.50
No_of_shares   : 40000
Confirm permanent updation:[Y/N] Y_
```

Figure 4.5 Modification of a Record

```

Confirm permanent updation:[Y/N] Y

Enter the new record :
Share ID      : RS12

Customer Name  : Rakesh Sharma

Company Name   : TATA STEEL

Share_value   : 423.12

No_of_shares  : 40000_

```

Figure 4.6 Modification of a Record

4.4.6 Display

Figure 4.7 is a snapshot of the display screen which shows the inserted records in entry sequenced order.

```

*****
*  DISPLAY ALL RECORDS USING B-TREE  *
*****

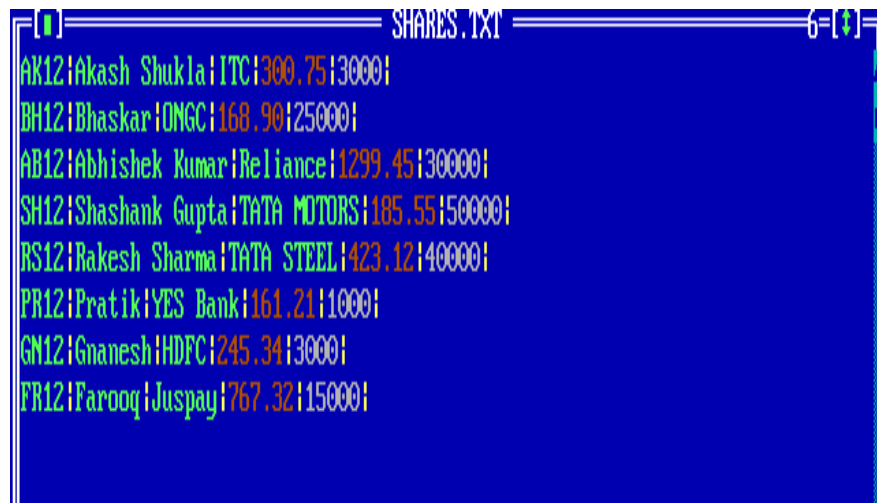
```

Share ID	Customer	Company	Share Value	No_of_shares
AK12	Akash Shukla	ITC	300.75	3000
BH12	Bhaskar	ONGC	168.90	25000
AB12	Abhishek Kumar	Reliance	1299.45	30000
SH12	Shashank Gupta	TATA MOTORS	185.55	50000
RS12	Rakesh Sharma	TATA STEEL	423.12	40000
PR12	Pratik	YES Bank	161.21	1000
GN12	Gnanesh	HDFC	245.34	3000
FR12	Farooq	Juspay	767.32	15000

Figure 4.7 Display

4.4.7 Data File Contents

Figure 4.8 is a snapshot of the data file which stores all the records inserted by the user in entry sequenced order.



```

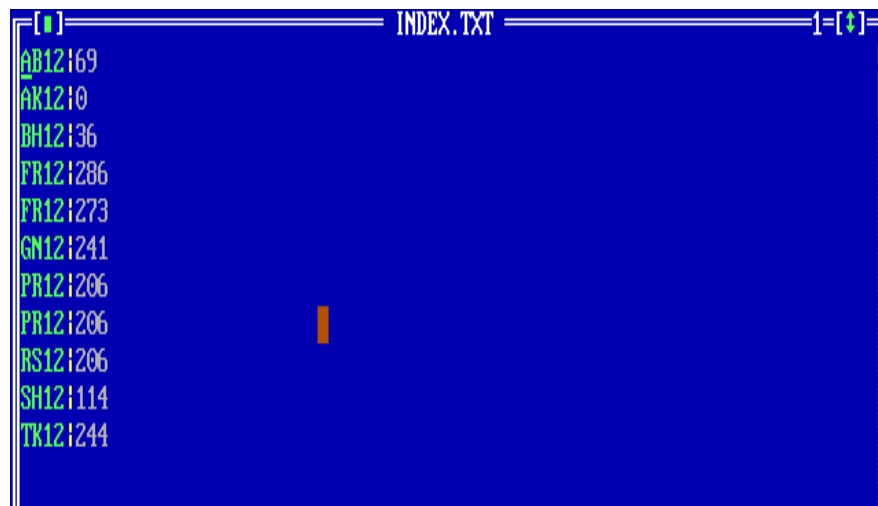
SHARES.TXT
AK12|Akash Shukla|ITC|300.75|3000|
BH12|Bhaskar|ONGC|168.90|25000|
AB12|Abhishek Kumar|Reliance|1299.45|30000|
SH12|Shashank Gupta|TATA MOTORS|185.55|50000|
RS12|Rakesh Sharma|TATA STEEL|423.12|40000|
PR12|Pratik|YES Bank|161.21|1000|
GM12|Gnanesh|HDFC|245.34|3000|
FR12|Farooq|Juspay|767.32|15000|

```

Figure 4.8 Data File Contents

4.4.8 Index File Contents

Figure 4.9 is a snapshot of the index file which stores the primary key i.e., share_id and byte offset of the primary key in ascending order of the primary keys.



```

INDEX.TXT
AB12|69
AK12|0
BH12|36
FR12|286
FR12|273
GM12|241
PR12|206
PR12|206
RS12|206
SH12|114
TK12|244

```

Figure 4.9 Index File Contents

4.4.9 B Tree structure Display

Figure 4.10 is a snapshot of the structure of B-tree that contains primary keys of the record stored in nodes. Figure 4.11 is a snapshot that shows the structure of a single record using the B Tree.

```

*****
*   B-TREE STRUCTURE DISPLAY   *
*****

Level-1:   AK12   PR12   SH12
-----
Block Structure
*****
Node :0
keys[0] : AB12
keys[1] : AK12
Node :1
keys[0] : BH12
keys[1] : FR12
keys[2] : GN12
keys[3] : PR12
Node :2
keys[0] : RS12
keys[1] : SH12

```

Figure 4.10 B Tree Structure

```

*****
*   DISPLAY ALL RECORDS USING B-TREE   *
*****

COUNT : 1
*****
Share ID      : BF21
Customer Name : Arun
Company Name  : Amazon
Share Value   : 2500
No_of_shares  : 5000
Press any key ...

*****_

```

Figure 4.11 B Tree Node

Chapter 5

Conclusion and Future Enhancement

This mini project is a simple program which is used to insert, retrieve, delete and update the shares and stock details. As seen earlier, the concept used in the development of the mini project is B tree. It has its set of advantages and disadvantages.

The mini project is designed to store the details of companies, customers and their shares. This greatly helps in reducing the muddling of huge amounts of data and also minimizes the risk of errors while trying to store or retrieve the data. It is a totally secure system where the records can be accessed only by the verified user who knows the value of primary key. The Shares and stock details is a mini project developed to assist both the companies and customers in supervising their shares.

The mini project can be improved by implemented it using B+ tree. In B+ tree, records can be accessed sequentially using keys. The same project can also be implemented using hashing for small data and easy accessing. Secondary key such as cust_name along with share_id as the primary key can be used for simplified and more efficient access of records.

References

- [1] File Structures: An Object-Oriented Approach in C++, PEARSON, 3rd Edition.
- [2] K.R. Venugopal, K.G. Srinivas, P.M. Krishnaraj: File Structures Using C++, Tata McGraw-Hill, 2008.
- [3] Scot Robert Ladd: C++ Components and Algorithms, BPB Publications, 1993
- [4] www.geeksforgeeks.com
- [5] www.includehelp.com