Chapter 1

INTRODUCTION

The **Vehicle Insurance Database** is a stand-alone application. It provides a user-friendly, interactive Menu Driven Interface (MDI). All data is stored in files for persistence. The application uses 2 files: An Index file, to store the primary index, and, a Data file, to store records pertaining to the vehicle's chassis number, insurance number.

1.1 Introduction to File Structure

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

1.1.1 History

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

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

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 thee tree was not a single record, as in a binary tree, but a file block containing dozens, perhaps even hundreds, of records

.

It took 10 years until a solution emerged in the form of a *B-Tree*. Whereas AVL trees grow from the top down as records were added, B-Trees grew from the bottom up. B-Trees provided excellent access performance, but there was a cost: no longer could a file be accessed sequentially with efficiency. The problem was solved by adding a linked list structure at the bottom level of the B-Tree. The combination of a B-Tree and a sequential linked list is called a *B+ tree*.

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

1.1.3 Various Kinds of storage of Fields and Records

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

Field Structures

The four most common methods as shown in Fig. 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 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 as shown in Fig 1.2 and Fig 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 ofbytes 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.

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*.

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

Method 4: Use an Index to Keep Track of Addresses

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

up the position of a record in the index, then seek to the record in the data file.

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

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

1.1.4 Application of File Structure

Relative to other parts of a computer, disks are slow. 1 can pack thousands of megabytes on a disk that fits into a noteboook , disks are *very* slow compared to memory. On the other hand, disks provide enormous capacity at much less cost than memory. They also keep the information stored on them when they are turned off.

Tension between a disk's relatively slow access time and its enormous, nonvolatile capacity is the driving force behind file structure design. Good file structure design will give us access to all the capacity without making our applications spend a lot of time waiting for the disk.

Chapter 2

SYSTEM ANALYSIS

2.1 Analysis of Application

Vehicle Insurance Database is used by the Insurance Company to view whether a given vehicle is insured or not. The system is initially used to add vehicle details containing the chassis no, vehicle brand, vehicle model etc. file corresponding to the vehicle details. The system can then be used to search, delete, modify or display existing records of all vehicles, and, to view a current vehicle whether it is insured or not.

2.2 Structure used to Store the Fields and Records

Storing Fields

Fixing the Length of Fields:

In the vehicle insurance database, the chassis no. field is a character array that can hold a string value of some maximum size. The size of the array is larger than the longest string it can hold. The vehicle model are strings while the price is a floating-point number, each of the numbers, converted into ASCII before writing to the data file. Simple arithmetic is sufficient to recover data from the original fields.

Separating the Fields with Delimiters:

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

Storing Records

Making Records a Predictable Number of Fields:

In this system, we have a fixed number of fields, each with a maximum length, that combine to make a data record. 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. We have 18 contiguous fields and we can recognize fields simply by counting the fields *modulo 18*.

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. We look up the position of a record in the B-Tree, and then seek to the record in the data file.

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

The system is initially used to add vehicle records containing the chassis number, model name, vehicle brand into the file corresponding to the vehicle. In the insertion section user is prompted with the various options starting with the Chassis number of the vehicle which should be in the proper sequence and if the correct pattern is not followed then it again asks to enter the details. After that vehicle name is asked followed by vehicle price and brand.

- Display

The system can then be used to display existing records of all vehicles. The records are displayed based on the ordering of chassis number maintained in the B-Tree of indexes, which here is, an ascending order. Only records with references in the B-Tree of indexes are displayed. This prevents records marked as deleted (using '$') from being displayed. There is also an option to display the nodes of the B-Tree level-wise.

- Search

The system can then be used to search for existing records of all semesters. The user is prompted for a chassis number, 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 semester 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 all record. The reference to a deleted record is removed from index while the deleted record persists in the data file. A '$' is placed in the first byte of the first field (CN) of the record, to help distinguish it from records that should be displayed. 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.

- Modify

The reference to a deleted record is removed from index while the deleted record persists in the data file. A '$' is placed in the first byte of the first field (CN) of the record, to help distinguish it from records that should be displayed. 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. If present, after deletion, the system is used to insert the modified vehicle record containing, possibly, again by prompting the menu.

- Vehicle Record

The user is prompted for a CN, which is used as the key in searching for records in the B-Tree of indexes. The record with the entered CN is searched for in all vehicle data files. The details retrieved from each file in which a matching record is found, are used to display the details of vehicle bearing the entered CN. A "Record Not Found", prefixed with the vehicle, is returned for each vehicle data file in which a matching record is not found while a "CN=", prefixed with the vehicle, is returned for each vehicle data file in which a matching record is found.

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 and reference pair of fields. The primary key field is the CN field while the reference field is the starting byte offset of the matching record in the data file, with one B-Tree of indexes per semester data file. Each B-Tree node can have a maximum of 4 and a minimum of 2 index entries.

The integer byte offset is stored in the B-Tree, and hence written to an index file, in ASCII form. On retrieval, the byte offset is converted into an integer value, before it is used to seek to a data record, as in the case of requests for a search, delete, modify or vehicle record, 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 the Fields and Records

The CN is declared as a character array that can hold a maximum of 10 characters. Checks are done to ensure the CN if of exactly 10 characters during input. The VName & Vbrand of each vehicle is declared as string, each having a predetermined range, only within which it is accepted during input (0 to 10 for CN and Hyundai, Nissan etc. for brand). The price field is declared as a floating-point integer.

Hence, a typical vehicle data file record can have up to 46 bytes of information to be stored, hence occupying a maximum of 65 bytes, in the data file. This includes the 18 bytes taken up by the field delimiters (|) and the 1 byte taken up by the record delimiter (\n).

The class declaration of a typical semester file record is as shown in Fig 3.1:

```
public:
char cn[25];
char vname[30];
char vaddr[40];
char price[10];
char brand[ZO] ;
void Clear();
int Unpack(fstreame();   int Pack(fstream&);
;int Pack(fstream&);
void Input(int);
void Display();
nvi(){I
void Assign(vi&);
```

Fig. 3.1 class vinsurance

3.2 User Interface

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

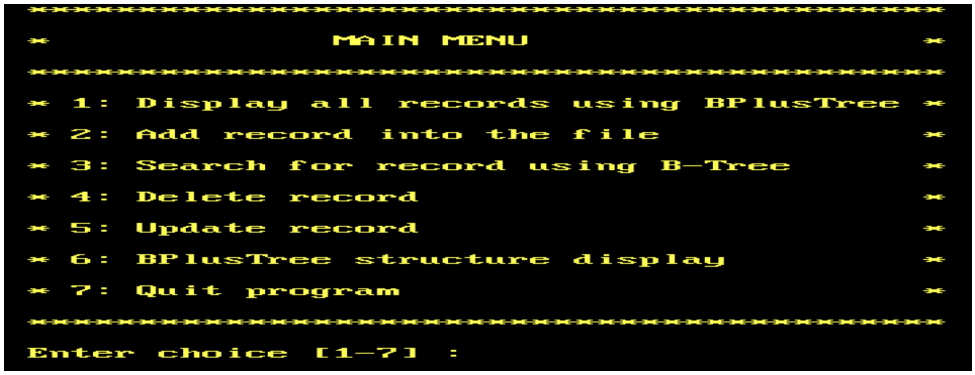


Fig. 3.2 User Menu Screen

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 CN of the vehicle. Finally, the user must enter the CN, followed Vname, Vaddress, price, brand. 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 CN, until the user enters one of exactly 10 characters.
- Price is in floating type.
- Brand is from the options-vok, hyn, Bugatti, Nissan, lambo.

After all the values are accepted, a "done…" message is displayed and the user is prompted to press any key to return back to the menu screen.

3.2.2 Display of a Record

If the operation desired is Display of Records, the user is required to enter 1 as his/her choice, from the menu displayed, after which a new screen is displayed. If there are no records in any file, the "no records found" message is displayed. For the vehicle files with at least 1 record 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 of the B-Tree level-wise, 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 CN, whose file from which a record is 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. If there is at least 1 record in the file, the user is prompted for the CN, whose matching record is to be deleted. The U 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 Search 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 CN, whose file the record is to be searched for. If there are no records in the file, a "no records to search"

message is displayed, and the user is prompted to press any key to return back to the menu screen. If there is at least 1 record in the file, the user is prompted for the CN, whose matching record is to be searched for. The CN 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 Modify of a Record

If the operation desired is Modify, 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 CN, whose file from which a record is to be modified. The Modify operation is implemented as a deletion followed by an insertion. 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. If there is at least 1 record in the file, the user is prompted for the CN, whose matching record is to be deleted.

The CN 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 CN in which the modified record is to be inserted. Finally, the user must enter the CN, followed by Vname, Vaddress, price, brand. 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 CN, until the user enters one of exactly 10 characters.
- Price is in floating type.
- Brand is from the options-vok, hyn, Bugatti, Nissan, lambo.

After all the values are accepted, a "done…" message is displayed and the user is prompted to press any key to return back to the menu screen.

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 Fig. 3.3 and Fig. 3.4. Each object of type "index" contains a CN filed and an address field. One object, representing the root of the B-Tree, is created for each vehicle file. 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.

The above operations are used to maintain a balanced tree-structure after each insertion and deletion. The links of a node, beginning at the root, are traversed recursively, while displaying, and, writing to the index file, in order to maintain an ascending order among index entries. The same links are traversed when there is a request to search for a record, if present. The address fields of each node contain the ASCII form of the actual integer byte-offset, obtained from the data files' get pointers. They are converted into integers, before seeking to a record in any data file, as in the case of retrieving details of a record found using a search operation. The retrieved details are either displayed, with suitable headings, as is (for Display and Search.

```
class node // class for btree node
{
        public:
        char eus[4][11];
        node *dptrs[41];
        node *uptr;
        block *ptr[4];
        int cnt;
        node();
        nnodeati
        int isLeaf();
        void split(node *,node *,node *);

}
```

```
class btree
{          // class for btree
public:
btree();
int insert(char*,block *);
node* findLeaf(char*,inttl);
block *search(char*,int&):
void create();
void dis(node*);
nbtrel()11
node *root;
```

Figure 3.3 btree class

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 vehicle file is declared as a *vinsurance* object with the CN, Vanme, Vaddress, price, brand 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 *btreenode*, 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 also has an array of objects, each an instance of the class type *index*. Each object of type *index* contains a CN filed and an address field, which stores the ASCII representation of the starting byte 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 *btreenode*, 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 2 types of files used are data files and index files. *open()* and *close()* are invoked on the file handle of the file to be opened/closed. *open()* takes 2 parameters- the filename and the mode of access. *close()* takes no parameters.

4.1.4 Character Arrays and Character functions

Character arrays are used to store the CN fields to be written to data files and stored in a B-Tree index object. They are also used to store the ASCII representations of the integer and floating-point fields (SG), that are written to the data file, and, the starting byte offsets of data records, to be written to the index file. Character functions are defined in the header file *ctype.h*. Some of the functions used include:

- *toupper( )* – used to convert lowercase characters to uppercase characters.
- *itoa( )* – to store ASCII representations of integers in character arrays
- *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.
- atof( ) – converts an ASCII value into a floating-point number.

4.2 Pseudocode

4.2.1 Insertion Module Pseudocode

The *insertion( )* function (Fig. 4.1) adds index objects to the B-Tree if the USN 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.

```
/* insert val in B-Tree */
void insertion(index val) {
        int flag;
        index i;
        btreeNode *child;
    //  cout<<"start\n";
        flag = setValueInNode(val, &i, root, &child);
        if (flag)
                root = createNode(i, child);
}
```

Fig. 4.1 *insertion( )* function

4.2.2 Display Module Pseudocode

The *traversal()* function (Fig. 4.2) traverses the B-Tree in ascending order accessing each index object stored at each node. The byte offsets in the objects are used to retrieve and display the corresponding data record fields. It is a recursive function.

```
block *SSET :: FindNode(char *val)      // function to find the block
      {
       BK t=head;
           int flag=0;

       node *temp=bt.root;
       int flg;
       temp = bt.findLeaf(val,flg);
       for(int i=0;i<temp->cnt;i++)
        if((strcmpi(temp->keys[i],val)>=0))
        {
         t=temp->ptr[i];
         flag=1;
         break;

        }
       else if((temp->cnt!=0)&& (flag!=1) &&(!(strcmpi(temp->keys[i],val)==0)))
         t=temp->ptr[temp->cnt-1];
       return t;

      }
```

Fig. 4.2 *traversal( )* function

There is also an option to display the nodes of a B-Tree level-wise using the *dispbtree()* function (Fig. 4.3). It is a recursive function.

```
void SSET :: display()           // frunction to display nodes
           {
                 fstream file;
                 file.open("index.txt",ios::out);
                  int j=0;
        BK t;
        t=head;
        getch();
        cout<<"*********************************************";
        cout<<"\n Block Structure \n";
        cout<<"*********************************************";
           while(t != 0)
        {
         cout<<"\n Node :"<<j;
         file<<"\n Node :"<<j;
         for(int i=0;i<t->cnt;i++)
         {
          cout<<"\n keys["<<i<<"] : " <<t->keys[i];
          //    <<"\t disp["<<i<<"] : "<<t->disp[i]
//        cout<<"\n";
         file<<"\n keys["<<i<<"] : " <<t->keys[i];
         }
         t=t->link;
         j++;
        }
           }
```

Fig. 4.3 *dispbtree( )* function

4.2.3 Deletion Module Pseudocode

The *deletion()* function (Fig. 4.4) deletes values from nodes and balances the B-Tree after, by merging or redistributing objects in the nodes. It makes calls to other recursive and non-recursive functions.

```
/* delete val from B-tree */
void deletion(index val, btreeNode *myNode) {
        btreeNode *tmp;
        z=1;
        if (!delValFromNode(val, myNode)) {
        x=0;
                cout<<"\t\tRecord not found\n";
                return;
        } else {
                if (myNode->count == 0) {
                        tmp = myNode;
                        myNode = myNode->link[0];
                        free(tmp);
                }
        }
        root = myNode;
        return;
}
```

Fig. 4.4 deletion( ) function

4.2.4 Search Module Pseudocode

The *searching()* function (Fig. 4.5) traverses the B-Tree, based on the values of objects in the root node. It is a recursive function.

```
block *btree::search(char *key,int &fnd)
{
   int i,flg=0;
   fnd=0;
   node *x=findLeaf(key,flg),*a1,*a2;
   for(i=0;i<x->cnt;i++)
    if(strcmp(key,x->keys[i])<=0)
    {
          fnd=1;
          return(x->ptr[i]);
    }
   return head;
}
```

Fig. 4.5 search( ) function

4.2.5 Modify Module Pseudocode

The modify operation is implemented as a call to the *deletion()* function followed by a call to the *insertion()* function (Fig. 4.6).

```
void SSET :: del(char *val)      // function to delete key from block
      {
       int z,flg=0,i;
       BK x=FindNode(val);
       for(i=0;i<x->cnt;i++)
        if(strcmpi(x->keys[i],val)==0)
        {
         flg=1;
         z=i;
         break;
        }
       if(flg==1)
          {
        if(x->cnt-1 < 2)
        {
         for(int j=i;i<x->cnt;i++)
         {
          strcpy(x->keys[j],x->keys[i+1]);
          strcpy(x->keys[i+1],"");
          x->disp[j]=x->disp[i+1];
          x->disp[i+1]=-1;
          j++;
         }
         x->cnt--;
         x->merge();
         delete bt.root;
         //bt.create();
        }
        else
        {
         for(int j=i;i<x->cnt;i++)
         {
          strcpy(x->keys[j],x->keys[i+1]);
          strcpy(x->keys[i+1],"");
          x->disp[j]=x->disp[i+1];
          x->disp[i+1]=-1;
          j++;
         }
         x->cnt--;
         if(z==x->cnt)
         {
         delete bt.root;
          bt.root = new node;
          //bt.create();
         }
        }
       }
       else
        cout<<"\n\nKey "<<val<<" not found\n";
      }
```

Fig. 4.6 modify( ) module

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()* function (Fig. 4.8), to keep index file up-to-date and consistent.

```
int varlen::Write(fstream &file)
{
  if(file.write(Buffer,BufSiz))
  {
        file.write("#",1);
        return 1;
  }
  return 0;
}
```

Fig. 4.7 *write( )* function

4.3 Testing

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

4.3.1 Unit Testing

1. CN input it checked to see if it is of 10 characters of the form:
   - A digit for the first character
   - Characters (uppercase or lowercase) for the next 2 characters
   - Digits for the next 2 characters
   - Characters (uppercase or lowercase) for the next 2 characters
   - Digits for the last 3 characters.

Table 4.1 Unit test case for CN Input Check

Sl No. of test case	1
Name of test	Check test
Item / Feature being tested	Input for CN field
Sample Input	CN=’1RN15ISA0A’. Upon press of ENTER key.
Expected output	Prompt for CN with ‘Enter CN:’ message.
Actual output	‘Enter CN:’ message displayed.
Remarks	Test succeeded

4.3.2 Integration Testing

1. The insertion function is checked to see if a duplicate record is attempted to be inserted and that the data files and index files are correctly updated with the appropriate records. It is also checked to see if validation of input values is performed correctly.

Table 4.2 Integration test case for Insertion module

Sl No. of test case	1
Name of test	Check test
Item / Feature being tested	Insertion module
Sample Input	CN='1AA15IS800'. Upon press of ENTER key.
Expected output	CN and marks inputs accepted, followed by 'Record inserted' message and a 'Press any key to go back to Menu' message displayed.
Actual output	CN and marks inputs accepted, followed by 'Record inserted' message and a 'Press any key to go back to Menu' message is displayed.
Remarks	Test succeeded

Fig 4.8 Integration test cases for insertion module

Table 4.3 Integration test case for Insertion module

Sl No. of test case	2
Name of test	Check test
Item / Feature being tested	Insertion module

Sample Input	CN='1aa1588561'. Upon press of ENTER key.
Expected output	"Duplicates not allowed" message followed by "Press any key to go back to Menu" message displayed.
Actual output	"Duplicates not allowed" message followed by "Press any key to go back to Menu" message is displayed.
Remarks	Test succeeded

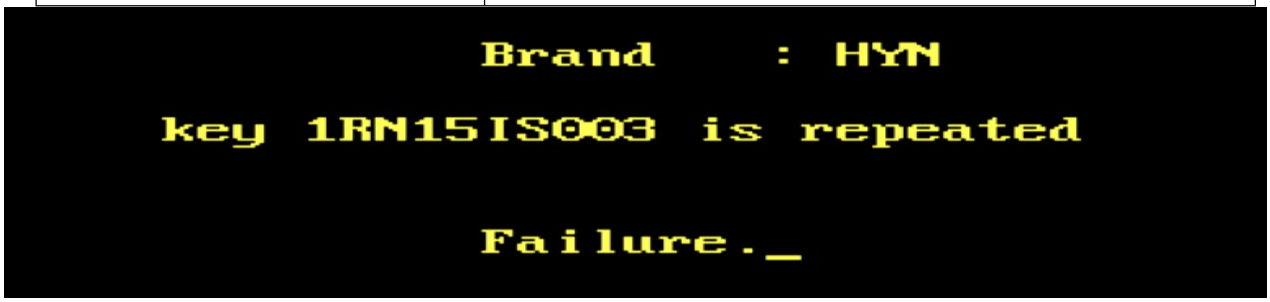


Fig. 4.9 Integration test case for insertion

2. The deletion function is checked to see if validation of input values is performed correctly, the correct results are returned based on whether a file contains records or not and whether desired record is present in a file or not, only existing matching records are deleted and that the result of the deletion is reflected in the data and index files.

Table 4.4 Integration test case for Deletion module

Sl No. of test case	3
Name of test	Check test
Item / Feature being tested	Deletion module
Sample Input	CN='1AA458900'. Upon press of ENTER key.
Expected output	"Record Deleted" message followed by "Press any key to go back to Menu" message displayed.
Actual output	"Record Deleted" message followed by "Press any key to go back to Menu" message is displayed.
Remarks	Test succeeded

Fig. 4.10 Integration test case for Deletion

3. The search function is checked to see if validation of input values is performed correctly, correct results are returned based on whether a file contains records or not and whether desired record is present in a file or not.

Table 4.5 Integration test case for Search module

Sl No. of test case	5
Name of test	Check test
Item / Feature being tested	Search module
Sample Input	CN='1ma88IS800'. Upon press of ENTER key.
Expected output	"Record found" message followed by details of the record and a "Press any key to go back to Menu" message displayed.
Actual output	"Record found" message followed by details of the record and a "Press any key to go back to Menu" message is displayed.
Remarks	Test succeeded

Table 4.6 Integration test case for Search module

Sl No. of test case	6
Name of test	Check test
Item / Feature being tested	Search module
Sample Input	CN='1MS78CC300'. Upon press of ENTER key.
Expected output	"Record not found" message followed by "Press any key to go back to Menu" message displayed.
Actual output	"Record not found" message followed by "Press any key to go back to Menu" message is displayed.
Remarks	Test succeeded

Fig. 4.11 Integration test cases for search module

4. The modify function is checked to see if the correct results are returned based on whether a file contains records or not and whether desired record is present in a file or not, only existing records are deleted, the deletion followed by insertion happens in the proper way, validation of input values is performed correctly, and that the result of the modification is reflected in the data and index files.

5. The modify function is checked to see if the correct results are returned based on whether a file contains records or not and whether desired record is present in a file or not, only existing records are deleted, the deletion followed by insertion happens in the proper way, validation of input values is performed correctly.

Table 4.7 Integration test case for Modify module

Sl No. of test case:	7
Name of test:	Check test

Item / Feature being tested	Modify module
Sample Input	CN='1RN15IS004'. Upon press of ENTER key.
Expected output	'Record deleted' message displayed. USN and marks inputs accepted, followed by 'Record inserted' message and a 'Press any key to go back to Menu' message displayed.
Actual output	'Record deleted' message is displayed. CN inputs accepted, followed by 'Record inserted' message and a 'Press any key to go back to Menu' message is displayed.
Remarks	Test succeeded

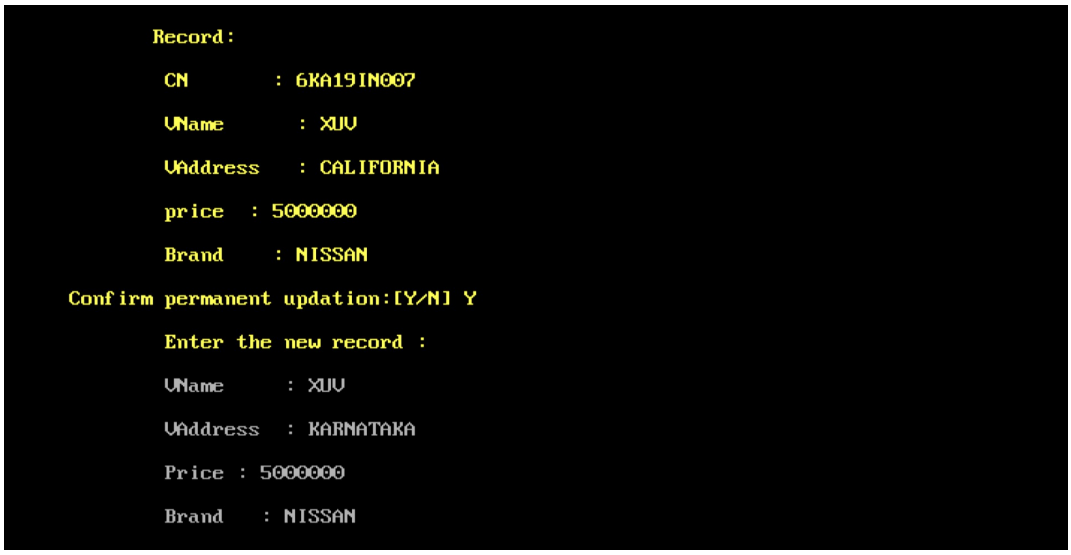


Fig.4.12 Integration test case for modify module

Table 4.8 Integration test case for Modify module

Sl No. of test case	8
Name of test	Check test
Item / Feature being tested	Modify module
Sample Input	CN='1RN15IS003'. Upon press of ENTER key.
Actual output	"Record not found" message followed by "Press any key to go back to Menu" message is displayed.

4.3.3 System Testing

System Testing is a level of the software testing where a complete and integrated software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements. The application is run to check if all the modules (functions) can be executed concurrently.

Table 4.9 System test case for Vehicle Insurance Details

Sl No. of test case	1
Name of test	Check test
Item / Feature being tested	Vehicle Insurance database
Sample Input	Choices entered in the order: 1 2 4 2 5 2 6 2 7 3 8
Expected output	Screens displayed (except menu screen) in order for: Insertion of a record Display of records before deletion of a record. Deletion of a record. Display of records after deletion of a record. Search for a record. Display of records before modification of a record. Modification of a record. Display of records after modification of a record. Display of B-Trees
Actual output	Screens are displayed in order
Remarks	Test succeeded

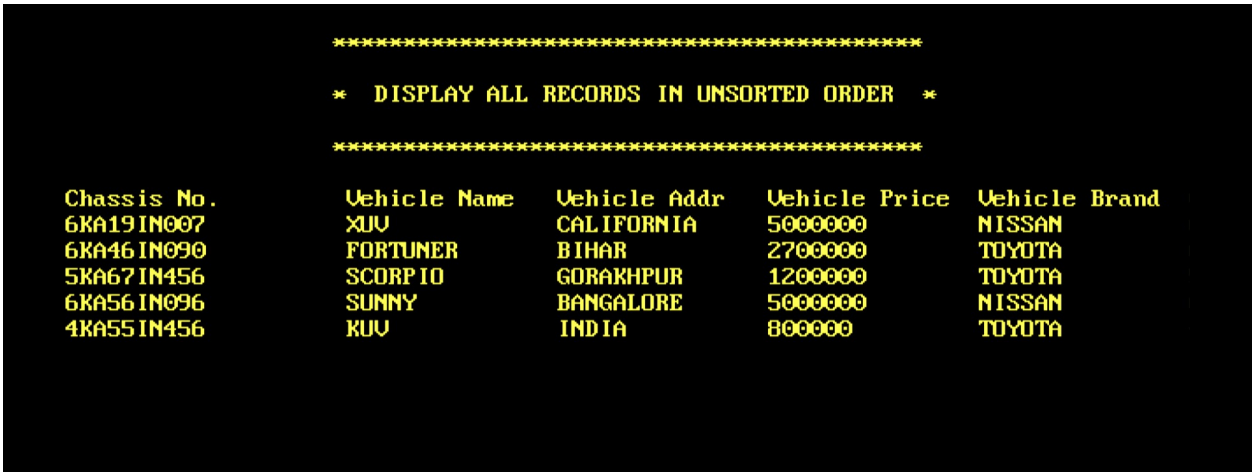


Fig. 4.13 Display of unsorted data

4.4 Discussion of Results

All the menu options provided in the application and its operations have been presented in as screen shots from Fig 4.23 to 4.34

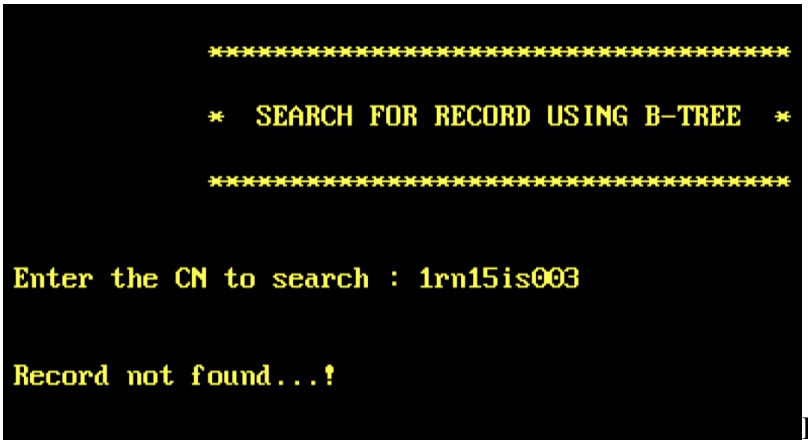
4.4.1 Menu Options

Fig. 4.14 User Menu Screen

4.4.2 Insertion

Fig. 4.15 Insertion Screen

4.4.2 Searching a Record



F

Fig. 4.16 Search for a record.

4.4.3 Before and After Modification

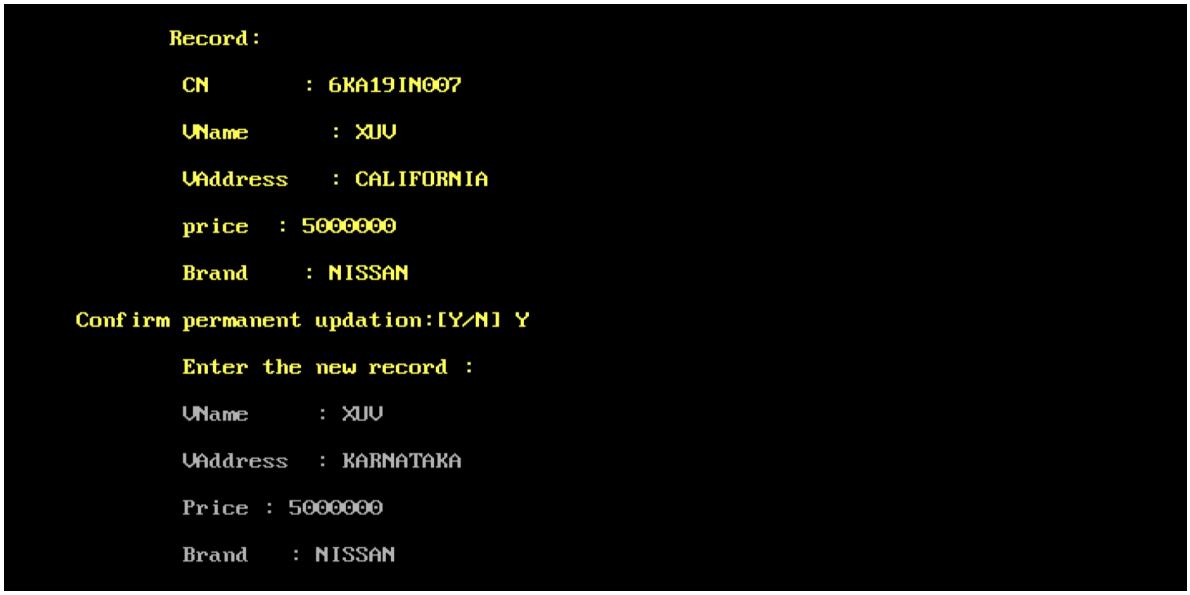


Fig. 4.17 Display of records after modification of a record.

4,45 Vehicle Record File Contents

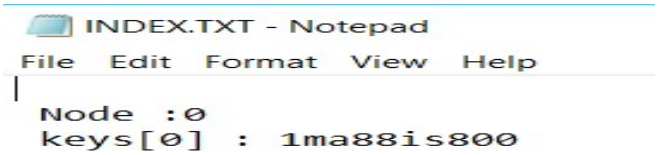


Fig. 4.18 Index File Contents.

Chapter 5

CONCLUSION AND FUTURE ENHANCEMENTS

The Vehicle Insurance database focuses on providing the users, the ability to view whether a given vehicle is insured or not. We implemented using a B-Tree of simple indexes, to allow viewing of the details related to the vehicle. The system can be used to add vehicle records containing the Chassis number, Vname, Vaddress, price, brand.

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

More improvisations can be incorporated like having vehicle database.

Different record structures and field structures can be used to increase efficiency and performance.

REFERENCES

1. File Structures: An Object-Oriented Approach in C++, PEARSON, 3rd Edition.
2. www.geeksforgeeks.org
3. uxmankabir.wordpress.com