File Structures

File Structures is the Organization of Data in Secondary Storage devices in such a way that minimize the access time and the storage space. 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.

Data Representation in Memory

Record:

A subdivision of a file, containing data related to a single entity.

Field :

A subdivision of a record containing a single attribute of the entity which the record describes.

Stream of bytes:

A file which is regarded as being without structure beyond separation into a sequential set of bytes.

1. Within a program, data is temporarily stored in variables.
2. Individual values can be aggregated into structures, which can be treated as a single variable with parts.
3. In C++, classes are typically used as as an aggregate structure.
4. C++ Person class (version 0.1):

```
class Person {
  public:
    char FirstName [11];
    char LastName[11];
    char Address [21];
    char City [21];
    char State [3];
    char ZIP [5];
};
```

With this class declaration, variables can be declared to be of type Person. The individual fields within a Person can be referred to as the name of the variable and the name of the field, separated by a period (.).

EX : C++ Program:

```
#include<iostream.h>
#include<string.h>
class Person {
  public:
    char FirstName [11];
    char LastName[11];
    char Address [31];
```

```
    char City [21];
    char State [3];
    char ZIP [5];
};
void Display (Person);
int main ()
{
        Person Clerk;
        strcpy (Clerk.FirstName, "Fred");  strcpy (Clerk.LastName, "Flintstone");
        strcpy (Clerk.Address, "4444 Granite Place");
        strcpy (Clerk.City, "Rockville");
        strcpy (Clerk.State, "MD");
        strcpy (Clerk.ZIP, "00001");

        Display (Clerk);
}

void Display (Person Someone)
 {
       cout << Someone.FirstName << Someone.LastName<< Someone.Address <<
       Someone.City<< Someone.State << Someone.ZIP;
  }
```

In memory, each Person will appear as an aggregate, with the individual values being parts of the aggregate

Person					
Clerk					
FirstName	**LastName**	**Address**	**City**	**State**	**ZIP**
Fred	Flintstone	4444 Granite Place	Rockville	MD	0001

The output of this program will be:

FredFlintstone4444 Granite PlaceRockvilleMD00001LilyMunster1313 Mockingbird LaneHollywoodCA90210

Obviously, this output could be improved. It is marginally readable by people, and it would be difficult to program a computer to read and correctly interpret this output.

A Stream File

- In the Windows, DOS, UNIX, and LINUX operating systems, files are not internally structured; they are streams of individual bytes.

F	r	E	d		F	l	i	n	t	s	t	o	n	e	4	4	4	4		G	r	a	n	...

- The only file structure recognized by these operating systems is the separation of a text file into lines.

- For Windows and DOS, two characters are used between lines, a carriage return (ASCII 13) and a line feed (ASCII 10);
- For UNIX and LINUX, one character is used between lines, a line feed (ASCII 10);

The code in applications programs can, however, impose internal organization on stream files. File processing in C++ is performed using the fstream class. Unlike the FILE structure, fstream is a complete C++ class with constructors, a destructor and overloaded operators.

To perform file processing, you can declare an instance of an fstream object. If you do not yet know the name of the file you want to process, you can use the default constructor.

Unlike the FILE structure, the fstream class provides two distinct classes for file processing. One is used to write to a file and the other is used to read from a file.

Opening a File

In C program , the type FILE is used for a file variable and is defined in the stdio.h file. It is used to define a file pointer for use in file operations. Before we can write to a file, we must open it. What this really means is that we must tell the system that we want to write to a file and what the file name is. We do this with the fopen() function illustrated in the first line of the program. The file pointer, fp in our case, points to the file and two arguments are required in the parentheses, the file name first, followed by the file type.

In C++ , two way we can open file by using **constructor** or by using member function **open().**

1.Open a file by constructor:

You can first declare an instance of the **stream** class using one of its constructors from the following syntaxes to open a file :

- **ofstream:** Stream class to write on files
- **ifstream:** Stream class to read from files
- **fstream:** Stream class to both read and write from/to files.

Syntaxes :

ofstream obj(const char* FileName, int FileMode);
or
ifstream obj(const char* *FileName*, int *FileMode*);
or
fstream obj(const char* *FileName*, int *FileMode*);

These classes are derived directly or indirectly from the classes istream, and ostream. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files.

The first argument of the constructor, *FileName*, is a constant string that represents the file that you want to open. The *FileMode* argument is a natural number that follows the table of modes as we described below.

Mode	Description
ios::app	If *FileName* is a new file, data is written to it. If *FileName* already exists and contains data, then it is opened, the compiler goes to the end of the file and adds the new data to it.
ios::ate	If *FileName* is a new file, data is written to it and subsequently added to the end of the file. If *FileName* already exists and contains data, then it is opened and data is written in the current position.
ios::in	If *FileName* is a new file, then it gets created fine as an empty file. If *FileName* already exists, then it is opened and its content is made available for processing
ios::out	If *FileName* is a new file, then it gets created fine as an empty file. Once/Since it gets created empty, you can write data to it. If *FileName* already exists, then it is opened, its content is destroyed, and the file becomes as new. Therefore you can create new data to write to it. Then, if you save the file, which is the main purpose of this mode, the new content is saved it.*This operation is typically used when you want to save a file
ios::trunk	If *FileName* already exists, its content is destroyed and the file becomes as new
ios::nocreate	If *FileName* is a new file, the operation fails because it cannot create a new file. If *FileName* already exists, then it is opened and its content is made available for processing
ios::noreplace	If *FileName* is a new file, then it gets created fine. If *FileName* already exists and you try to open it, this operation would fail because it cannot create a file of the same name in the same location.

Let's see an example:

```
 1 // basic file operations
 2 #include <iostream>
 3 #include <fstream>
 4 using namespace std;
 5
 6 int main () {
 7   ofstream myfile("example.txt");
 8   myfile << "Writing this to a file.\n";
 9   myfile.close();
10   return 0;
11 }
12
```

```
[file example.txt]
Writing this to a file.
```

This code creates a file called example.txt and inserts a sentence into it in the same way we are used to do without, but using the file stream myfile instead.

2. Member function open().

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to *open a file*. An open file is represented within a program by a stream object (an instantiation of one of these classes, in the previous example this was myfile) and any input or output operation performed on this stream object will be applied to the physical file associated to it.

In order to open a file with a stream object we use its member function open():

open (filename, mode);

Where filename is a null-terminated character sequence of type const char * (the same type that string literals have) representing the name of the file to be opened, and mode is an optional parameter with a combination of the flags. flags can be combined using the bitwise operator OR (|). For example, if we want to open the fileexample.bin in binary mode to add data we could do it by the following call to member function open():

```
1 ofstream myfile;
2 myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

Each one of the open() member functions of the classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

For ifstream and ofstream classes, ios::in and ios::out are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the open() member function.

The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as *text files*, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

Since the first task that is performed on a file stream object is generally to open a file, these three classes include a constructor that automatically calls the open() member function and has the exact same parameters as this member. Therefore, we could also have declared the previous myfile object and conducted the same opening operation in our previous example by

writing:

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.

To check if a file stream was successful opening a file, you can do it by calling to member is_open() with no arguments. This member function returns a bool value of true in the case that indeed the stream object is associated with an open file, or false otherwise:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```

Closing a file

When we are finished with our input and output operations on a file we shall close it so that its resources become available again. In order to do that we have to call the stream's member function close(). This member function takes no parameters, and what it does is to flush the associated buffers and close the file:

```
myfile.close();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function close().

Text files

Text file streams are those where we do not include the ios::binary flag in their opening mode. These files are designed to store text and thus all values that we input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

Data output operations on text files are performed in the same way we operated with cout:

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;
int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open())
  {
    myfile << "This is a line.\n";
    myfile << "This is another line.\n";
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

```
[file example.txt]
This is a line.
This is another line.
```

Data input from a file can also be performed in the same way that we did with cin:

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())
  {
    while ( myfile.good() )
    {
      getline (myfile,line);
      cout << line << endl;
    }
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

```
This is a line.
This is another line.
```

This last example reads a text file and prints out its content on the screen. Notice how we have used a new member function, called good() that returns true in the case that the stream is ready for input/output operations. We have created a while loop that finishes when indeed myfile.good() is no longer true, which will happen either if the end of the file has been reached or if some other error occurred.

Checking state flags

In addition to good(), which checks whether the stream is ready for input/output operations, other member functions exist to check for specific states of a stream (all of them return a bool value):

bad()
: Returns true if a reading or writing operation fails. For example in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.

fail()
: Returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.

eof()
: Returns true if a file open for reading has reached the end.

good()
: It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true.

In order to reset the state flags checked by any of these member functions we have just seen we can use the member function clear(), which takes no parameters.

get and put stream pointers

All i/o streams objects have, at least, one internal stream pointer:

ifstream, like istream, has a pointer known as the *get pointer* that points to the element to be read in the next input operation.

ofstream, like ostream, has a pointer known as the *put pointer* that points to the location where the next element has to be written.

Finally, fstream, inherits both, the get and the put pointers, from iostream (which is itself derived from bothistream and ostream).

These internal stream pointers that point to the reading or writing locations within a stream can be manipulated using the following member functions:

tellg() and tellp()

These two member functions have no parameters and return a value of the member type pos_type, which is an integer data type representing the current position of the get stream pointer (in the case of tellg) or the put stream pointer (in the case of tellp).

seekg() and seekp()

These functions allow us to change the position of the get and put stream pointers. Both functions are overloaded with two different prototypes. The first prototype is:

seekg (position);
seekp (position);

Using this prototype the stream pointer is changed to the absolute position position (counting from the beginning of the file). The type for this parameter is the same as the one returned by functions tellg and tellp: the member type pos_type, which is an integer value.

The other prototype for these functions is:

seekg (offset, direction);
seekp (offset, direction);

Using this prototype, the position of the get or put pointer is set to an offset value relative to some specific point determined by the parameter direction. offset is of the member type off_type, which is also an integer type. Anddirection is of type seekdir, which is an enumerated type (enum) that determines the point from where offset is counted from, and that can take any of the following values:

ios::beg	offset counted from the beginning of the stream
ios::cur	offset counted from the current position of the stream pointer
ios::end	offset counted from the end of the stream

The following example uses the member functions we have just seen to obtain the size of a file:

```
// obtaining file size
#include <iostream>
#include <fstream>
int main () {
  long begin,end;
  ifstream myfile ("example.txt");
  begin = myfile.tellg();
  myfile.seekg (0, ios::end);
  end = myfile.tellg();
  myfile.close();
  cout << "size is: " << (end-begin) << " bytes.\n";
  return 0;
}
```

```
size is: 40 bytes.
```

Buffers and Synchronization

When we operate with file streams, these are associated to an internal buffer of type streambuf. This buffer is a memory block that acts as an intermediary between the stream and the physical file. For example, with anofstream, each time the member function put (which writes a single character) is called, the character is not written directly to the physical file with which the stream is associated. Instead of that, the character is inserted in that stream's intermediate buffer.

When the buffer is flushed, all the data contained in it is written to the physical medium (if it is an output stream) or simply freed (if it is an input stream). This process is called *synchronization* and takes place under any of the following circumstances:

When the file is closed: before closing a file all buffers that have not yet been flushed are synchronized and all pending data is written or read to the physical medium.

- **When the buffer is full:** Buffers have a certain size. When the buffer is full it is automatically synchronized.
- **Explicitly, with manipulators:** When certain manipulators are used on streams, an explicit synchronization takes place. These manipulators are: flush and endl.
- **Explicitly, with member function sync():** Calling stream's member function sync(), which takes no parameters, causes an immediate synchronization. This function returns an int value equal to -1 if the stream has no associated buffer or in case of failure. Otherwise (if the stream buffer was successfully synchronized) it returns 0.

1. Delineation of Records in a File

Fixed Length Records

A record which is predetermined to be the same length as the other records in the file.

Record 1	Record 2	Record 3	Record 4	Record 5

- The file is divided into records of equal size.
- All records within a file have the same size.
- Different files can have different length records.
- Programs which access the file must know the record length.
- Offset, or position, of the nth record of a file can be calculated.
- There is no external overhead for record separation.
- There may be internal fragmentation (unused space within records.)
- There will be no external fragmentation (unused space outside of records) except for deleted records.
- Individual records can always be updated in place.
- Algorithms for Accessing Fixed Length Records
- Code for Accessing Fixed Length Records
- Code for Accessing String Records

Delimited Variable Length Records

T E R M S

1. **variable length record**
   A record which can differ in length from the other records of the file.
2. **delimited record**
   A variable length record which is terminated by a special character or sequence of characters.
3. **delimiter**
   A special character or group of characters stored after a field or record, which indicates the end of the preceding unit.

Record 1	#	Record 2	#	Record 3	#	Record 4	#	Record 5	#

- The records within a file are followed by a delimiting byte or series of bytes.
- The delimiter cannot occur within the records.
- Records within a file can have different sizes.
- Different files can have different length records.
- Programs which access the file must know the delimiter.
- Offset, or position, of the nth record of a file cannot be calculated.
- There is external overhead for record separation equal to the size of the delimiter per record.
- There should be no internal fragmentation (unused space within records.)
- There may be no external fragmentation (unused space outside of records) after file updating.

- Individual records cannot always be updated in place.
- Algorithms for Accessing Delimited Variable Length Records
- Code for Accessing Delimited Variable Length Records
- Code for Accessing Variable Length Line Records

Length Prefixed Variable Length Records

110	Record 1	40	Record 2	100	Record 3	80	Record 4	70	Record 5

- The records within a file are prefixed by a length byte or bytes.
- Records within a file can have different sizes.
- Different files can have different length records.
- Programs which access the file must know the size and format of the length prefix.
- Offset, or position, of the nth record of a file cannot be calculated.
- There is external overhead for record separation equal to the size of the length prefix per record.
- There should be no internal fragmentation (unused space within records.)
- There may be no external fragmentation (unused space outside of records) after file updating.
- Individual records cannot always be updated in place.
- Algorithms for Accessing Prefixed Variable Length Records
- Code for Accessing PreFixed Variable Length Records

Delineation of Fields in a Record

Fixed Length Fields

Field 1	Field 2	Field 3	Field 4	Field 5

- Each record is divided into fields of correspondingly equal size.
- Different fields within a record have different sizes.
- Different records can have different length fields.
- Programs which access the record must know the field lengths.
- There is no external overhead for field separation.
- There may be internal fragmentation (unused space within fields.)

Delimited Variable Length Fields

Field 1	!	Field 2	!	Field 3	!	Field 4	!	Field 5	!

- The fields within a record are followed by a delimiting byte or series of bytes.
- Fields within a record can have different sizes.
- Different records can have different length fields.
- Programs which access the record must know the delimiter.
- The delimiter cannot occur within the data.
- If used with delimited records, the field delimiter must be different from the record delimiter.
- There is external overhead for field separation equal to the size of the delimiter per field.
- There should be no internal fragmentation (unused space within fields.)

Length Prefixed Variable Length Fields

12	Field 1	4	Field 2	10	Field 3	8	Field 4	7	Field 5

- The fields within a record are prefixed by a length byte or bytes.
- Fields within a record can have different sizes.
- Different records can have different length fields.
- Programs which access the record must know the size and format of the length prefix.
- There is external overhead for field separation equal to the size of the length prefix per field.
- There should be no internal fragmentation (unused space within fields.)

Representing Record or Field Length

Record or field length can be represented in either binary or character form.

- The length can be considered as another hidden field within the record.
- This length field can be either fixed length or delimited.
- When character form is used, a space can be used to delimit the length field.
- A two byte fixed length field could be used to hold lengths of 0 to 65535 bytes in binary form.
- A two byte fixed length field could be used to hold lengths of 0 to 99 bytes in decimal character form.
- A variable length field delimited by a space could be used to hold effectively any length.
- In some languages, such as strict Pascal, it is difficult to mix binary values and character values in the same file.
- The C++ language is flexible enough so that the use of either binary or character format is easy.

Tagged Fields

- Tags, in the form "Keyword=Value", can be used in fields.
- Use of tags does not in itself allow separation of fields, which must be done with another method.
- Use of tags adds significant space overhead to the file.
- Use of tags does add flexibility to the file structure.
- Fields can be added without affecting the basic structure of the file.
- Tags can be useful when records have sparse fields - that is, when a significant number of the possible attributes are absent.

Index :

A structure containing a set of entries, each consisting of a key field and a reference field, which is used to locate records in a data file.

Key field :

The part of an index which contains keys.

Reference field:

The part of an index which contains information to locate records.

- An index imposes order on a file without rearranging the file.
- Indexing works by indirection.

A Simple Index for Entry-Sequenced Files

Simple index

An index in which the entries are a key ordered linear list.

- Simple indexing can be useful when the entire index can be held in memory.
- Changes (additions and deletions) require both the index and the data file to be changed.
- Updates affect the index if the key field is changed, or if the record is moved.
- An update which moves a record can be handled as a deletion followed by an addition.

Direct access

Accessing data from a file by record position with the file, without accessing intervening records.

Relative record number

An ordinal number indicating the position of a record within a file.

Primary key

A key which uniquely identifies the records within a file.

Secondary key

A search key other than the primary key.

Secondary index

An index built on a secondary key.

- Secondary indexes can be built on any field of the data file, or on combinations of fields.
- Secondary indexes will typically have multiple locations for a single key.
- Changes to the data may now affect multiple indexes.
- The reference field of a secondary index can be a direct reference to the location of the entry in the data file.
- The reference field of a secondary index can also be an indirect reference to the location of the entry in the data file, through the primary key.
- Indirect secondary key references simplify updating of the file set.
- Indirect secondary key references increase access time.

Cosequential Algorithms

- Initialize (open the input files.)
- Get the first item from each list
- While there is more to do:
    - Compare the current items from each list
    - Based on the comparison, appropriately process one or all items.
    - Get the next item or items from the appropriate list or lists.
    - Based on the whether there were more items, determine if there is more to do.
- Finalize (close the files.)

Match :

The process of forming a list containing all items common to two or more lists.

Cosequential Match Algorithm

- Initialize (open the input and output files.)
- Get the first item from each list.
- While there is more to do:
    - Compare the current items from each list.
    - If the items are equal,
        - Process the item.
        - Get the next item from each list.
        - Set *more* to true iff none of this lists is at end of file.
    - If the item from list *A* is less than the item from list *B*,
        - Get the next item from list *A*.
        - Set *more* to true iff list *A* is not at end-of-file.
    - If the item from list *A* is more than the item from list *B*,
        - Get the next item from list *B*.
        - Set *more* to true iff list *B* is not at end-of-file.
- Finalize (close the files.)

Merge

The process of forming a list containing all items in any of two or more lists.

Cosequential Merge Algorithm

- Initialize (open the input and output files.)
- Get the first item from each list.
- While there is more to do:
    - Compare the current items from each list.
    - If the items are equal,
        - Process the item.
        - Get the next item from each list.
    - If the item from list *A* is less than the item from list *B*,
        - Process the item from list *A*.
        - Get the next item from list *A*.
    - If the item from list *A* is more than the item from list *B*,
        - Process the item from list *B*.

        - Get the next item from list *B*.
    - Set *more* to false iff all of this lists are at end of file.
- Finalize (close the files.)

Hashing

- Key driven file access should be *O*(1) - that is, the time to access a record should be a constant which does not vary with the size of the dataset.
- Indexing can be regarded as a table driven function which translates a key to a numeric location.
- Hashing can be regarded as a computation driven function which translates a key to a numeric location.

Hashing
The transformation of a search key into a number by means of mathematical calculations.

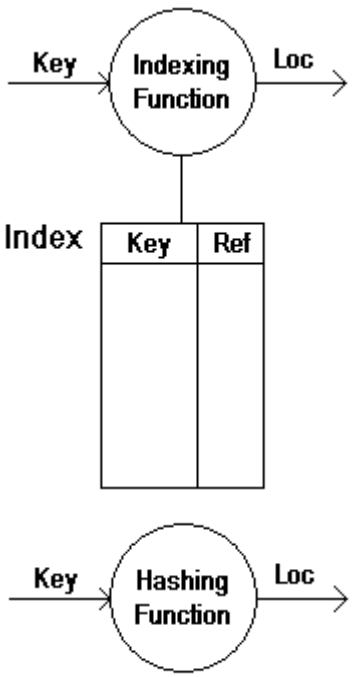
Randomize
To transform in an apparently random way.

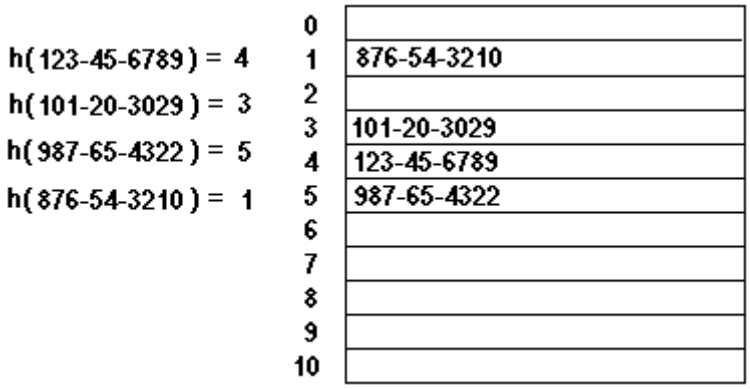
- Hashing uses a repeatable pseudorandom function.
- The hashing function should produce a uniform distribution of hash values.

Uniform distribution
A randomization in which each value in a range has an equal probability.

- For each key, the result of the hashing function is used as the home address of the record.
- home address
  The address produced by the hashing of a record key.

- Under ideal conditions, hashing provides *O*(1) key driven file access**.**





Hashing Algorithms

Modulus

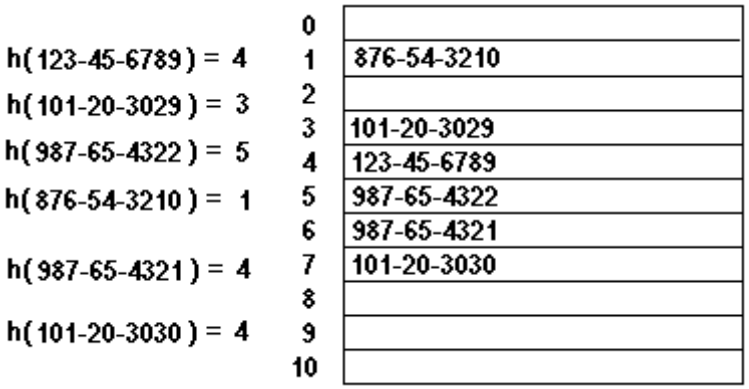
- Modulus - the key is divided by the size of the table, and the remainder is used as the hash function.
- Example:
  Key = 123-45-6789
  123456789 % 11 = 5
  h(123-45-6789) = 5
- Modulus functions work better when the divisor is a prime number, or at least not a composite of small numbers.

Collision Resolution by Progressive Overflow

Progressive overflow:

A collision resolution technique which places overflow records at the first empty address after the home address

- With progressive overflow, a sequential search is performed beginning at the home address.
- The search is continued until the desired key or a blank record is found.
- Progressive overflow is also referred to as *linear probing.*



Storing more than One Record per Address: Buckets

Bucket :

An area of a hash table with a single hash address which has room for more than one record.

- When using buckets, an entire bucket is read or written as a unit. (Records are not read individually.)
- The use of buckets will reduce the average number of probes required to find a record.

	0				
h(123-45-6789) = 4	1	876-54-3210			
h(101-20-3029) = 3	2				
	3	101-20-3029			
h(987-65-4322) = 5	4	123-45-6789	987-65-4321	101-20-3030	
h(876-54-3210) = 1	5	987-65-4322			
	6				
h(987-65-4321) = 4	7				
	8				
h(101-20-3030) = 4	9				
	10				

Introduction to Btrees

Tree structures support various basic dynamic set operations including Search, Predecessor, *Successor*, *Minimum*, *Maximum*, *Insert*, and *Delete* in time proportional to the height of the tree. Ideally, a tree will be balanced and the height will be *log n* where *n* is the number of nodes in the tree. To ensure that the height of the tree is as small as possible and therefore provide the best running time, a balanced tree structure like a red-black tree, AVL tree, or b-tree must be used.

When working with large sets of data, it is often not possible or desirable to maintain the entire structure in primary storage (RAM). Instead, a relatively small portion of the data structure is maintained in primary storage, and additional data is read from secondary storage as needed. Unfortunately, a magnetic disk, the most common form of secondary storage, is significantly slower than random access memory (RAM). In fact, the system often spends more time retrieving data than actually processing data.

B-trees are balanced trees that are optimized for situations when part or all of the tree must be maintained in secondary storage such as a magnetic disk. Since disk accesses are expensive (time consuming) operations, a b-tree tries to minimize the number of disk accesses. For example, a b-tree with a height of 2 and a branching factor of 1001 can store over one billion keys but requires at most two disk accesses to search for any node .

The Structure of B-Trees

Unlike a binary-tree, each node of a b-tree may have a variable number of keys and children. The keys are stored in non-decreasing order. Each key has an associated child that is the root of a subtree containing all nodes with keys less than or equal to the key but greater than the preceeding key. A node also has an additional rightmost child that is the root for a subtree containing all keys greater than any keys in the node.

A b-tree has a minumum number of allowable children for each node known as the *minimization factor*. If *t* is this *minimization factor*, every node must have at least *t - 1* keys. Under certain circumstances, the root node is allowed to violate this property by having fewer than *t - 1* keys. Every node may have at most *2t - 1* keys or, equivalently, *2t* children.

Since each node tends to have a large branching factor (a large number of children), it is typically neccessary to traverse relatively few nodes before locating the desired key. If access to each node requires a disk access, then a b-tree will minimize the number of disk accesses required. The minimzation factor is usually chosen so that the total size of each node corresponds to a multiple of the block size of the underlying storage device. This choice simplifies and optimizes disk access. Consequently, a b-tree is an ideal data structure for situations where all data cannot reside in primary storage and accesses to secondary storage are comparatively expensive (or time consuming).

Height of B-Trees

For *n* greater than or equal to one, the height of an *n*-key b-tree T of height *h* with a minimum degree *t* greater than or equal to 2,

$$h \leq \log_t \frac{n+1}{2}$$

For a proof of the above inequality, refer to Cormen, Leiserson, and Rivest pages 383-384.

The worst case height is *O(log n)*. Since the "branchiness" of a b-tree can be large compared to many other balanced tree structures, the base of the logarithm tends to be large; therefore, the number of nodes visited during a search tends to be smaller than required by other tree structures. Although this does not affect the asymptotic worst case height, b-trees tend to have smaller heights than other trees with the same asymptotic height.

Operations on B-Trees

The algorithms for the *search*, *create*, and *insert* operations are shown below. Note that these algorithms are single pass; in other words, they do not traverse back up the tree. Since b-trees strive to minimize disk accesses and the nodes are usually stored on disk, this single-pass approach will reduce the number of node visits and thus the number of disk accesses. Simpler double-pass approaches that move back up the tree to fix violations are possible.

Since all nodes are assumed to be stored in secondary storage (disk) rather than primary storage (memory), all references to a given node be be preceeded by a read operation denoted by *Disk-Read*. Similarly, once a node is modified and it is no longer needed, it must be written out to secondary storage with a write operation denoted by *Disk-Write*. The algorithms below assume that all nodes referenced in parameters have already had a corresponding *Disk-Read* operation. New nodes are created and assigned storage with the *Allocate-Node* call. The implementation details of the *Disk-Read*, *Disk-Write*, and *Allocate-Node* functions are operating system and implementation dependent.

B-Tree-Search(x, k)

```
i <- 1
while i <= n[x] and k > key_i[x]
    do i <- i + 1
if i <= n[x] and k = key_i[x]
    then return (x, i)
if leaf[x]
    then return NIL
    else Disk-Read(c_i[x])
        return B-Tree-Search(c_i[x], k)
```

The search operation on a b-tree is analogous to a search on a binary tree. Instead of choosing between a left and a right child as in a binary tree, a b-tree search must make an n-way choice. The correct child is chosen by performing a linear search of the values in the node. After finding the value greater than or equal to the desired value, the child pointer to the immediate left of that value is followed. If all values are less than the desired value, the rightmost child pointer is followed. Of course, the search can be terminated as soon as the desired node is found. Since the running time of the search operation depends upon the height of the tree, *B-Tree-Search* is $O(\log_t n)$.

B-Tree-Create(T)

```
x <- Allocate-Node()
leaf[x] <- TRUE
n[x] <- 0
Disk-Write(x)
root[T] <- x
```

The *B-Tree-Create* operation creates an empty b-tree by allocating a new root node that has no keys and is a leaf node. Only the root node is permitted to have these properties; all other nodes must meet the criteria outlined previously. The *B-Tree-Create* operation runs in time $O(1)$.

B-Tree-Split-Child(x, i, y)

```
z <- Allocate-Node()
leaf[z] <- leaf[y]
n[z] <- t - 1
for j <- 1 to t - 1
    do key_j[z] <- key_j+t[y]
if not leaf[y]
    then for j <- 1 to t
        do c_j[z] <- c_j+t[y]
n[y] <- t - 1
for j <- n[x] + 1 downto i + 1
    do c_j+1[x] <- c_j[x]
c_i+1 <- z
for j <- n[x] downto i
    do key_j+1[x] <- key_j[x]
key_i[x] <- key_t[y]
n[x] <- n[x] + 1
Disk-Write(y)
Disk-Write(z)
Disk-Write(x)
```

If is node becomes "too full," it is necessary to perform a split operation. The split operation moves the median key of node *x* into its parent *y* where *x* is the i^{th} child of *y*. A new node, *z*, is allocated, and all keys in *x* right of the median key are moved to *z*. The keys left of the median key remain in the original node *x*. The new node, *z*, becomes the child immediately to the right of the median key that was moved to the parent *y*, and the original node, *x*, becomes the child immediately to the left of the median key that was moved into the parent *y*.

The split operation transforms a full node with *2t - 1* keys into two nodes with *t - 1* keys each. Note that one key is moved into the parent node. The *B-Tree-Split-Child* algorithm will run in time *O(t)* where *t* is constant.

B-Tree-Insert(T, k)

```
r <- root[T]
if n[r] = 2t - 1
    then s <- Allocate-Node()
        root[T] <- s
            leaf[s] <- FALSE
            n[s] <- 0
            c1 <- r
            B-Tree-Split-Child(s, 1, r)
            B-Tree-Insert-Nonfull(s, k)
    else B-Tree-Insert-Nonfull(r, k)
```

B-Tree-Insert-Nonfull(x, k)

```
i <- n[x]
if leaf[x]
    then while i >= 1 and k < keyi[x]
         do keyi+1[x] <- keyi[x]
                i <- i - 1
        keyi+1[x] <- k
            n[x] <- n[x] + 1
            Disk-Write(x)
    else while i >= and k < keyi[x]
         do i <- i - 1
            i <- i + 1
            Disk-Read(ci[x])
            if n[ci[x]] = 2t - 1
                then B-Tree-Split-Child(x, i, ci[x])
                    if k > keyi[x]
                            then i <- i + 1
        B-Tree-Insert-Nonfull(ci[x], k)
```

To perform an insertion on a b-tree, the appropriate node for the key must be located using an algorithm similiar to *B-Tree-Search*. Next, the key must be inserted into the node. If the node is not full prior to the insertion, no special action is required; however, if the node is full, the node must be split to make room for the new key. Since splitting the node results in moving one key to the parent node, the parent node must not be full or another split operation is required. This process may repeat all the way up to the root and may require splitting the root node. This approach requires two passes. The first pass locates the node where the key should be inserted; the second pass performs any required splits on the ancestor nodes.

Since each access to a node may correspond to a costly disk access, it is desirable to avoid the second pass by ensuring that the parent node is never full. To accomplish this, the presented algorithm splits any full nodes encountered while descending the tree. Although this approach may result in unecessary split operations, it guarantees that the parent never needs

to be split and eliminates the need for a second pass up the tree. Since a split runs in linear time, it has little effect on the $O(t \log_t n)$ running time of *B-Tree-Insert*.

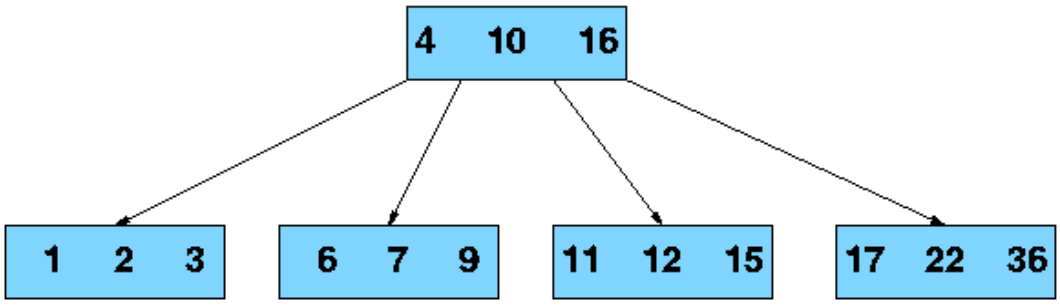
Splitting the root node is handled as a special case since a new root must be created to contain the median key of the old root. Observe that a b-tree will grow from the top.

B-Tree-Delete

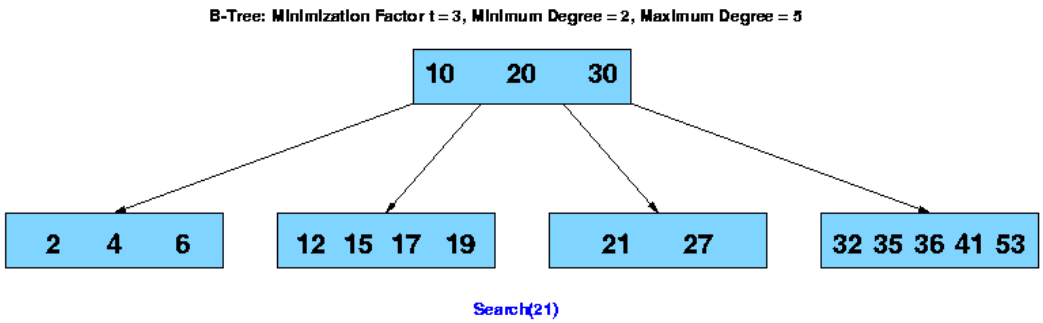
Deletion of a key from a b-tree is possible; however, special care must be taken to ensure that the properties of a b-tree are maintained. Several cases must be considered. If the deletion reduces the number of keys in a node below the minimum degree of the tree, this violation must be corrected by combining several nodes and possibly reducing the height of the tree. If the key has children, the children must be rearranged. For a detailed discussion of deleting from a b-tree, refer to Section 19.3, pages 395-397, of Cormen, Leiserson, and Rivest or to another reference listed below.

Examples

Sample B-Tree



Searching a B-Tree for Key 21



Inserting Key 33 into a B-Tree (w/ Split)

1. Write a C++ program to read series of names, one per line, from standard input and write these names spelled in reverse order to the standard output using I/O redirection and pipes. Repeat the exercise using an input file specified by the user instead of the standard input and using an output file specified by the user instead of the standard output.

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
#include<string.h>
void main()
{
char buf[20][20],infile[20],outfile[20],buf1[20];
char line[200];
int n, i,j,indx,x,y,ch;
fstream file,file1;
clrscr();
cout<<"Enter your choice\n1->file I/O\n2->standard I/O";
cin>>ch;
switch(ch)
{
case 1:
        cout<<"Enter the input file:"<<flush;
        cin>>infile;
        cout<<"Enter the output file:"<<flush;
        cin>>outfile;
        file.open(infile,ios::in);

        file.unsetf(ios::skipws);
        file1.open(outfile,ios::out);
        while(1)
        {

        if(!file.eof())
        {
                file.getline(line,'\n');
                strrev(line);
                file1<<line;
                file1<<"\n";
        }
        else
        break;

        }
        file.close();
        file1.close();
        getch();

        break;
case 2:
        cout<<"Enter the number of names ";
        cin>>n;
```

```
        for(i=1;i<=n;i++)
        {
                cout<<"enter name:"<<i<<"\t";
                cin>>buf[i];

        }
        for(i=1;i<=n;i++)
        {

               strrev(buf[i]);
                cout<<"reversed name:"<<buf[i]<<endl;
        }
        getch();
        }
        break;
}
```

Input/Output :

```
Enter your choice\n1->file I/O\n2->standard I/O
2
Emter the number of names
3
enter the name
CIT
AIT
SIT
The reversed name are:
TIC
TIA
TIS

C:\TC\BIN>edit file1.txt
Computer
Laptop


Enter your choice\n1->file I/O\n2->standard I/O
1
enter input file name: file1.txt
enter output filename: file2.txt

C:\TC\BIN>edit file1.txt file2.txt
retupmoC
potpaL
```

Output:

2. Write a C++ program to read and write student objects with fixed-length records and the fields delimited by "|". Implement pack ( ), unpack ( ), modify ( ) and search ( ) methods.

```
#include<iostream.h>
#include<stdio.h>
#include<fstream.h>
#include<stdlib.h>
#include<string.h>
# include<conio.h>

class Person
{
        char usn[30];
        char name[30];
        char address[30];
        char branch[30];
        char college[30];
        char buffer[50];
public:
        void input();
        void output();
        void search();
        void modify();
        void pack();
        void unpack();
        void Write();

};


void Person :: input()
{
        cout<<"Enter Usn"<<endl;
        cin>>usn;
        cout<<"Enter Name"<<endl;
        cin>>name;
        cout<<"Enter Address"<<endl;
        cin>>address;
        cout<<"Enter Branch"<<endl;
        cin>>branch;
        cout<<"Enter College"<<endl;
        cin>>college;
}

void Person :: output()
{
        cout<<"Usn :";
        puts(usn);
        cout<<"Name :";
        puts(name);
        cout<<"Address :";
```

```
        puts(address);
        cout<<"Branch :";
        puts(branch);
        cout<<"College :";
        puts(college);
}

void Person::pack()
{
        strcpy(buffer,usn); strcat(buffer,"|");
        strcat(buffer,name); strcat(buffer,"|");
        strcat(buffer,address); strcat(buffer,"|");
        strcat(buffer,branch); strcat(buffer,"|");
        strcat(buffer,college); strcat(buffer,"|");
        while(strlen(buffer)<50)
        strcat(buffer,"*");
}

void Person::unpack()
{
        char *ptr = buffer;
        while(*ptr!='*')
        {
                if(*ptr == '|')
                *ptr = '\0';
                ptr++;
        }
        ptr = buffer;
        strcpy(usn,ptr);

        ptr = ptr +strlen(ptr)+1;
        strcpy(name,ptr);

        ptr = ptr +strlen(ptr)+1;
        strcpy(address,ptr);

        ptr = ptr +strlen(ptr)+1;
        strcpy(branch,ptr);

        ptr = ptr+strlen(ptr)+1;
        strcpy(college,ptr);
}


void Person:: Write()
{
        fstream os("p.txt",ios::out|ios::app);
        os.write(buffer,sizeof(buffer));
        os.close();
}
```

```
void Person :: search()
{
        int found = 0;
        char key[30];
        fstream is("p.txt",ios::in);

        cout<<"Enter The Usn Of The Record To Be Searched "<<endl;
        cin>>key;

        while(!is.eof() && !found)
        {
            is.read(buffer,sizeof(buffer));
            unpack();
            if(strcmp(usn,key) == 0)
                {

                        cout<<"Record Found!!! "<<endl;
                        output();
                        found = 1;
                }
        }
        if(!found)
        cout<<"Record Not Found!!!"<<endl;
        is.close();
}


void Person :: modify()
{
        char key[30];
        char del='$';
        cout<<"Enter The USN Of The Record To Be Modified"<<endl;
        cin>>key;


        int found = 0;

        fstream is;
        is.open("p.txt",ios::in|ios::out);
        while(!is.eof())
        {
            is.read(buffer,sizeof(buffer));
            unpack();
            if(strcmp(usn,key) == 0)
            {
                        int pos=is.tellg();
                        pos=pos-50;
                        is.seekg(pos,ios::beg);
                        is<<del;
                        cout<<"ENTER
                        1:NAME\n2:ADDRESS\n3:BRANCH\n4:COLLEGE\n";
```

```
                        cout<<"Enter What to modify ? ";
                        int ch;
                        cin>>ch;
                        switch(ch)
                        {

                        case 1 :
                        cout<<"\n NAME :";
                        cin>>name;
                        break;

                        case 2:
                        cout<<"\n ADDRESS :";
                        cin>>address;
                        break;

                        case 3:
                        cout<<"\n BRANCH :";
                        cin>>branch;
                        break;

                        case 4:
                        cout<<"\n COLLEGE :";
                        cin>>college;
                        break;

                        default :
                        cout<<"wrong choice !!!";
                        exit(0);
                        }
                        found = 1;
                        pack();
                        Write();
                }
        }
        if(!found)
        cout<<"The Record with the given usn does not exist "<<endl;
        is.close();
}

void main()
{
        int choice = 1;
        clrscr();
        Person ob;

        while(choice < 4)
        {
                cout<<"1> Insert A Record "<<endl;
                cout<<"2> Search For A Record "<<endl;
                cout<<"3> Modify A Record "<<endl;
```

```
                cout<<"4> Exit "<<endl;
                cin>> choice;
                switch(choice)
                {
                case 1: ob.input();
                        ob.pack();
                        ob.Write();
                        break;

                case 2: ob.search();
                        break;

                case 3: ob.modify();
                        break;
                }
        }   getch();
}
```

Input/Output :

```
1> Insert A Record
2> Search For A Record
3> Modify A Record
4> Exit

1
Enter Usn
10is020
Enter Name
raju
Enter Address
gubbi
Enter Branch
ise
Enter College
Cit

1> Insert A Record
2> Search For A Record
3> Modify A Record
4> Exit
1
Enter Usn
10is021
Enter Name
usha
Enter Address
tumkur
Enter Branch
ise
Enter College
```

```
Cit

1> Insert A Record
2> Search For A Record
3> Modify A Record
4> Exit
1
Enter Usn
10cs040
Enter Name
risha
Enter Address
tumkur
Enter Branch
ise
Enter College
cit

1> Insert A Record
2> Search For A Record
3> Modify A Record
4> Exit
2
Enter The Usn Of The Record To Be Searched
10is021
Record Found!!!
Usn :10is021
Name :usha
Address :tumkur
Branch :ise
College :cit

1> Insert A Record
2> Search For A Record
3> Modify A Record
4> Exit
3
Enter The USN Of The Record To Be Modified
10is020
ENTER 1:NAME
2:ADDRESS
3:BRANCH
4:COLLEGE
Enter What to modify ? 2
 ADDRESS :Thirthahalli

1> Insert A Record
2> Search For A Record
3> Modify A Record
4> Exit
2
```

Enter The Usn Of The Record To Be Searched
10is020
Record Found!!!
Usn :10is020
Name :raju
Address :Thirthahalli
Branch :ise
College :cit
1> Insert A Record
2> Search For A Record
3> Modify A Record
4> Exit : 4

3.

Output:

3. Write a C++ program to read and write student objects with Variable - Length records using any suitable record structure. Implement pack ( ), unpack ( ), modify ( ) and search ( ) methods.

```
#include<iostream.h>
#include<stdio.h>
#include<fstream.h>
#include<stdlib.h>
#include<string.h>
# include<conio.h>

class Person
{
        char usn[30];
        char name[30];
        char address[30];
        char branch[30];
        char college[30];
        char buffer[100];
public:
        void input();
        void output();
        void search();
        void modify();
        void pack();
        void unpack();
        void Write();
};

void Person :: input()
{
        cout<<"Enter Usn"<<endl;
        cin>>usn;
        cout<<"Enter Name"<<endl;
        cin>>name;
        cout<<"Enter Address"<<endl;
        cin>>address;
        cout<<"Enter Branch"<<endl;
        cin>>branch;
        cout<<"Enter College"<<endl;
        cin>>college;
}

void Person :: output()
{
        istream& flush();
        cout<<"Usn :";
        puts(usn);
        cout<<"Name :";
        puts(name);
     cout<<"Address :";
```

```
        puts(address);
    cout<<"Branch :";
        puts(branch);
    cout<<"College :";
        puts(college);
}

void Person::pack()
{
        strcpy(buffer,usn); strcat(buffer,"|");
        strcat(buffer,name); strcat(buffer,"|");
        strcat(buffer,address); strcat(buffer,"|");
        strcat(buffer,branch); strcat(buffer,"|");
        strcat(buffer,college); strcat(buffer,"|");
        strcat(buffer,"#");
}

void Person::unpack()
{
        char *ptr = buffer;
        while(*ptr!='#')
        {
                if(*ptr == '|')
                *ptr = '\0';
                ptr++;
        }
        ptr = buffer;
        strcpy(usn,ptr);

        ptr = ptr +strlen(ptr)+1;
        strcpy(name,ptr);

        ptr = ptr +strlen(ptr)+1;
        strcpy(address,ptr);

        ptr = ptr +strlen(ptr)+1;
        strcpy(branch,ptr);

        ptr = ptr+strlen(ptr)+1;
        strcpy(college,ptr);
}

void Person:: Write()
{
        fstream os("p.txt",ios::out|ios::app);
        os.write(buffer,strlen(buffer));
        os<<endl;
        os.close();
}
void Person :: search()
{
```

```
        int found = 0;
        char key[30];
        fstream is("p.txt",ios::in);

         cout<<"Enter The Usn Of The Record To Be Searched "<<endl;
        cin>>key;

        while(!is.eof() && !found)
        {
                is.getline(buffer,'#');
                unpack();
                if(strcmp(usn,key) == 0)
                {

                        cout<<"Record Found!!! "<<endl;
                        output();
                        found = 1;
                }
        }
        if(!found)
        cout<<"Record Not Found!!!"<<endl;
        is.close();
}

void Person :: modify()
{
        char key[30];
        char del='$';
        cout<<"Enter The USN Of The Record To Be Modified"<<endl;
        cin>>key;

        int found = 0;
        fstream is;
        is.open("p.txt",ios::in|ios::out);
        while(!is.eof() && !found)
        {
                is.getline(buffer,'#');
                 int len=strlen(buffer);
                unpack();
                if(strcmp(usn,key) == 0)
                {
                        int pos=is.tellg();
                        pos=pos-len-2;
                        is.seekg(pos,ios::beg);
                        is<<del;
                        cout<<"ENTER 1:USN\n2:NAME\n3:ADDRESS \n4:BRANCH   \n
                        5:COLLEGE\n";
                        cout<<"Enter What to modify ? ";
                        int ch;
                        cin>>ch;
                        switch(ch)
```

```
                        {
                        case 1:
                        cout<<"USN :";
                        cin>>usn;
                        break;

                        case 2 :
                        cout<<"\n NAME :";
                        cin>>name;
                        break;

                        case 3:
                        cout<<"\n BRANCH :";
                        cin>>branch;
                        break;

                        case 4:
                        cout<<"\n ADDRESS :";
                        cin>>address;
                        break;

                        case 5:
                        cout<<"\n COLLEGE :";
                        cin>>college;
                        break;

                        default :
                        cout<<"wrong choice !!!";
                        break;
                        }
                        found = 1;
                        pack();
                        Write();
                }
        }
        if(!found)
        cout<<"The Record with the given usn does not exist "<<endl;
        is.close();
}

void main()
{
        int choice = 1;
        clrscr();
        Person ob;
        //istream& flush();
        //ostream& flush();
        //char filename[] = "p.txt";
        while(choice < 4)
        {
                cout<<"1> Insert A Record "<<endl;
```

```
            cout<<"2> Search For A Record "<<endl;
            cout<<"3> Modify A Record "<<endl;
            cout<<"4> Exit "<<endl;
                cin>> choice;
                switch(choice)
                {
                case 1: ob.input();
                        ob.pack();
                        ob.Write();
                        break;

                case 2: ob.search();
                        break;

                case 3: ob.modify();
                        break;
                }
        }   getch();
}
```

Input/Output :

```
1> Insert A Record
2> Search For A Record
3> Modify A Record
4> Exit
1
Enter Usn
10is020
Enter Name
raju
Enter Address
gubbi
Enter Branch
ise
Enter College
Cit

1> Insert A Record
2> Search For A Record
3> Modify A Record
4> Exit
1
Enter Usn
10is021
Enter Name
usha
Enter Address
tumkur
Enter Branch
ise
```

```
Enter College
Cit

1> Insert A Record
2> Search For A Record
3> Modify A Record
4> Exit
1
Enter Usn
10cs040
Enter Name
risha
Enter Address
tumkur
Enter Branch
ise
Enter College
cit

1> Insert A Record
2> Search For A Record
3> Modify A Record
4> Exit
2
Enter The Usn Of The Record To Be Searched
10is021
Record Found!!!
Usn :10is021
Name :usha
Address :tumkur
Branch :ise
College :cit

1> Insert A Record
2> Search For A Record
3> Modify A Record
4> Exit
3
Enter The USN Of The Record To Be Modified
10is020
ENTER 1:NAME
2:ADDRESS
3:BRANCH
4:COLLEGE
Enter What to modify ? 2
 ADDRESS :Thirthahalli

1> Insert A Record
2> Search For A Record
3> Modify A Record
4> Exit
```

```
2
Enter The Usn Of The Record To Be Searched
10is020
Record Found!!!
Usn :10is020
Name :raju
Address :Thirthahalli
Branch :ise
College :cit
1> Insert A Record
2> Search For A Record
3> Modify A Record
4> Exit
4.
```

Output:

4. Write a C++ program to write student objects with Variable - Length records using any suitable record structure and to read from this file a student record using RRN.

```
#include<iostream.h>
#include<stdio.h>
#include<fstream.h>
#include<stdlib.h>
#include<string.h>
#include<conio.h>

class Person
{
        int rrn[10];
        char usn[30];
        char name[30];
        char address[30];
        char branch[30];
        char college[30];
        char buffer[100];
        int count;

public:
        void input();
      void output();
        void searchrrn();
        void creatrrn();
        void pack();
      void unpack();
        void Write();
};

void Person :: input()
{
        cout<<"Enter Usn"<<endl;
        cin>>usn;
        cout<<"Enter Name"<<endl;
        cin>>name;
        cout<<"Enter Address"<<endl;
        cin>>address;
        cout<<"Enter Branch"<<endl;
        cin>>branch;
        cout<<"Enter College"<<endl;
        cin>>college;
}

void Person :: output()
{
        istream& flush();
        cout<<"Usn :";
        puts(usn);
        cout<<"Name :";
```

```
        puts(name);
        cout<<"Address :";
        puts(address);
        cout<<"Branch :";
        puts(branch);
        cout<<"College :";
        puts(college);
}

void Person::pack()
{
        strcpy(buffer,usn); strcat(buffer,"|");
        strcat(buffer,name); strcat(buffer,"|");
        strcat(buffer,address); strcat(buffer,"|");
        strcat(buffer,branch); strcat(buffer,"|");
        strcat(buffer,college); strcat(buffer,"|");
        strcat(buffer,"#");

}

void Person::unpack()
{
        char *ptr = buffer;
        while(*ptr!='#')
        {
                if(*ptr == '|')
                *ptr = '\0';
                ptr++;
        }
        ptr = buffer;
        strcpy(usn,ptr);

        ptr = ptr +strlen(ptr)+1;
        strcpy(name,ptr);

        ptr = ptr +strlen(ptr)+1;
        strcpy(address,ptr);

        ptr = ptr +strlen(ptr)+1;
        strcpy(branch,ptr);

        ptr = ptr+strlen(ptr)+1;
        strcpy(college,ptr);
}

void Person:: Write()
{
        fstream os("p.txt",ios::out|ios::app);
        os.write(buffer,strlen(buffer));
        os<<endl;
        os.close();
```

```
}

void Person :: searchrrn()
{
        int pos=-1;
        int key;
        cout<<"\n ENTER THE RRN:";
        cin>>key;

        if(key>count)
        cout<<"\n FILE IS NOT FOUND";
        else
        {
                fstream is("p.txt",ios::in);
                pos=rrn[key];
                is.seekp(pos,ios::beg);
                is.getline(buffer,'#');
                unpack();
                output();
                cout<<endl;
                cout<<buffer;
                cout<<endl<<endl;
                is.close();
        }
}

void Person::creatrrn()
{
        fstream fs;
        int pos;
        count=-1;
        fs.open("p.txt",ios::in);
        while(fs)
        {
                pos=fs.tellg();
                fs.getline(buffer,'#');
                if(fs.eof())
                break;
                rrn[++count]=pos;
        }
        fs.close();
}

void main()
{
        int choice = 1;
        clrscr();
        Person ob;

        while(choice < 3)
        {
```

```
                cout<<"1> Insert A Record "<<endl;
                cout<<"2> Search For A Record "<<endl;
                cout<<"3> Exit "<<endl;
                cin>> choice;
                switch(choice)
                {
                case 1: ob.input();
                        ob.pack();
                        ob.Write();
                        break;

                case 2: ob.creatrrn();
                        ob.searchrrn();
                        break;
                }
        }   getch();
}
```

Input/Output :

```
1> Insert A Record
2> Search For A Record
3> Exit
1
Enter Usn
10cs010
Enter Name
shruthi
Enter Address
tumkur
Enter Branch
cse
Enter College
Cit

1> Insert A Record
2> Search For A Record
3> Exit
1
Enter Usn
10is025
Enter Name
revanth
Enter Address
gubbi
Enter Branch
ise
Enter College
sit

1> Insert A Record
```

```
2> Search For A Record
3> Exit
1
Enter Usn
10cv030
Enter Name
sonali
Enter Address
bangaluru
Enter Branch
civil
Enter College
ssit

1> Insert A Record
2> Search For A Record
3> Exit
2

 ENTER THE RRN:1
Usn :10is025
Name :revanth
Address :gubbi
Branch :ise
College :sit

1> Insert A Record
2> Search For A Record
3> Exit
2
 ENTER THE RRN:4
 FILE IS NOT FOUND
1> Insert A Record
2> Search For A Record
3> Exit
```

Output:

5. Write a C++ program to implement simple index on primary key for a file of student objects. Implement add ( ), search ( ), delete ( ) using the index.

```
#include<iostream.h>
#include<stdio.h>
#include<fstream.h>
#include<stdlib.h>
#include<string.h>
# include<conio.h>

class Person
{
        int add[10],count;
        char usnl[10][10];
        char usn[30];
        char name[30];
        char address[30];
        char branch[30];
        char college[30];
        char buffer[100];

public:
        void input();
        void creatind();
        void search();
        void remove();
        void pack();
        void sort();
        int searchusn(char[20]);
        void Write();
};

void Person :: input()
{
        cout<<"Enter Usn"<<endl;
        cin>>usn;
        cout<<"Enter Name"<<endl;
        cin>>name;
        cout<<"Enter Address"<<endl;
        cin>>address;
        cout<<"Enter Branch"<<endl;
        cin>>branch;
        cout<<"Enter College"<<endl;
        cin>>college;
}

void Person::pack()
{
        strcpy(buffer,usn); strcat(buffer,"|");
        strcat(buffer,name); strcat(buffer,"|");
        strcat(buffer,address); strcat(buffer,"|");
```

```
        strcat(buffer,branch); strcat(buffer,"|");
        strcat(buffer,college); strcat(buffer,"|");
        strcat(buffer,"#");
}

void Person:: Write()
{
        fstream os("pt.txt",ios::out|ios::app);
        os.write(buffer,strlen(buffer));
        os<<endl;
        os.close();
}

void Person::creatind()
{
        int pos;
        count=-1;
        fstream file;
        file.open("pt.txt",ios::in);
        while(file)
        {
                pos=file.tellg();
                file.getline(buffer,'#');
                if(*buffer=='$')
                continue;
                if(file.eof())
                break;

                char *ptr=buffer;
                while(*ptr!='|')
                ptr++;
                *ptr='\0';
                strcpy(usnl[++count],buffer);
                add[count]=pos;
        }
        file.close();
        sort();
}

void Person::sort()
{

        int i,j,addlist;
        char temp[20];
        for(i=0;i<=count;i++)
        {
                for(j=i+1;j<=count;j++)
                {
                        if(strcmp(usnl[i],usnl[j])>0)
                        {
                                strcpy(temp,usnl[i]);
```

```
                                strcpy(usnl[i],usnl[j]);
                                strcpy(usnl[j],temp);
                                addlist=add[i];
                                add[i]=add[j];
                                add[j]=addlist;
                        }
                }
        }
}

int Person :: searchusn(char key[20])
{
        int low=0,high=count,mid=0,flag=0,pos;
        while(low<=high)
        {
                mid=(low+high)/2;
                if(strcmp(usnl[mid],key)==0)
                {
                        flag=1;
                        break;
                }
                if(strcmp(usnl[mid],key)>0)
                high=mid-1;
                else
                low=mid+1;
        }
        if(flag)
        return mid;
        else
        return -1;
}

void Person ::remove()
{
        char key[30];
        char del='$';
        fstream is;
        cout<<"ENTER THE USN :: "<<endl;
        cin>>key;
        int pos=searchusn(key);
        if(pos>=0)
        {
                is.open("pt.txt",ios::in|ios::out);
                int addl=add[pos];
                is.seekp(addl,ios::beg);
                is<<del;
                cout<<"Record DELETED !!! "<<endl;
                is.close();
                count--;
        }
        else
```

```
        cout<<"Record Not Found!!! "<<endl;
}

void Person::search()
{
        int pos=0;
        char key[20];
        fstream file;
        cout<<"\n ENTER THE KEY TO BE SEARCH : " ;
        cin>>key;
        pos=searchusn(key);
        if(pos>=0)
        {
                file.open("pt.txt",ios::in);
                int addl=add[pos];
                file.seekp(addl,ios::beg);
                file.getline(buffer,'#');
                cout<<"\n RECORD FOUND !!! "<<buffer;
                file.close();
        }
        else
        cout<<"Record Not Found!!! "<<endl;
}

void main()
{
        int choice = 1;
        clrscr();
        Person ob;
        while(choice < 4)
        {
             ostream&flush();
                cout<<"1> Insert A Record "<<endl;
             cout<<"2> Search For A Record "<<endl;
            cout<<"3> Delete A Record "<<endl;
            cout<<"4> Exit "<<endl;
                cin>> choice;
                switch(choice)
                {
                case 1: ob.input();
                        ob.pack();
                        ob.Write();
                        break;

                case 2: ob.creatind();
                        ob.search();
                        break;

                case 3: ob.creatind();
                        ob.remove();
                        break;
```

```
                }
        }   getch();
}
```

Input/Output :

```
1> Insert A Record
2> Search For A Record
3> Delete a record
4> Exit
1
Enter Usn
10cs010
Enter Name
shruthi
Enter Address
tumkur
Enter Branch
cse
Enter College
Cit

1> Insert A Record
2> Search For A Record
3> Delete a record
4> Exit
1
Enter Usn
10is025
Enter Name
revanth
Enter Address
gubbi
Enter Branch
ise
Enter College
sit

1> Insert A Record
2> Search For A Record
3> Delete a record
4> Exit
1
Enter Usn
10cv030
Enter Name
sonali
Enter Address
bangaluru
Enter Branch
civil
```

```
Enter College
ssit

1> Insert A Record
2> Search For A Record
3> Delete A Record
4> Exit
2

 ENTER THE KEY TO BE SEARCH : 10is025

 RECORD FOUND !!!
10is025|revanth|gubbi|ise|sit|#
1> Insert A Record
2> Search For A Record
3> Delete A Record
4> Exit
3
ENTER THE USN ::
10cv030
Record DELETED !!!
1> Insert A Record
2> Search For A Record
3> Delete A Record
4> Exit
2

 ENTER THE KEY TO BE SEARCH : 10cv030
Record Not Found!!!
```

Output:

6. Write a C++ program to implement index on secondary key, the name, for a file of student objects. Implement add ( ), search ( ), delete ( ) using the secondary index.

```
#include<iostream.h>
#include<stdio.h>
#include<fstream.h>
#include<stdlib.h>
#include<string.h>
# include<conio.h>

class Person
{
        int add[10],count;
        char sk[10][10];
        char usn[30];
        char name[30];
        char address[30];
        char branch[30];
        char college[30];
        char buffer[100];
public:
        void input();
        void creatind();
        void search();
        void remove();
        void delfile(int);
        void pack();

        void sort();
        void readf(int);
        int searchn(char[20]);
        void Write();
};

void Person :: input()
{
        cout<<"Enter Usn"<<endl;
        cin>>usn;
        cout<<"Enter Name"<<endl;
        cin>>name;
        cout<<"Enter Address"<<endl;
        cin>>address;
        cout<<"Enter Branch"<<endl;
        cin>>branch;
        cout<<"Enter College"<<endl;
        cin>>college;
}
```

```
void Person::pack()
{
        strcpy(buffer,usn); strcat(buffer,"|");
        strcat(buffer,name); strcat(buffer,"|");
        strcat(buffer,address); strcat(buffer,"|");
        strcat(buffer,branch); strcat(buffer,"|");
        strcat(buffer,college); strcat(buffer,"|");
        strcat(buffer,"#");
}
void Person:: Write()
{
        fstream os("p.txt",ios::in | ios::app);
        os.write(buffer,strlen(buffer));
        os<<endl;
        os.close();
        creatind();
}

void Person::creatind()
{
        int pos;
        count=-1;
        fstream file;
        file.open("p.txt",ios::in);
        while(file)
        {
                pos=file.tellg();
                file.getline(buffer,'#');
                if(*buffer=='$')
                continue;
                if(file.eof())
                break;

                char *ptr=buffer;
                while(*ptr!='#')
                {
                        if(*ptr=='|')
                        *ptr='\0';
                        ptr++;
                }
                ptr=buffer;
                ptr=ptr+strlen(ptr)+1;
                strcpy(sk[++count],ptr);
                add[count]=pos;
        }
        file.close();
        sort();
}
```

```
void Person::sort()
{
        int i,j,addlist;
        char temp[20];
        for(i=0;i<=count;i++)
        {
                for(j=i+1;j<=count;j++)
                {
                        if(strcmp(sk[i],sk[j])>0)
                        {
                                strcpy(temp,sk[i]);
                                strcpy(sk[i],sk[j]);
                                strcpy(sk[j],temp);
                                addlist=add[i];
                                add[i]=add[j];
                                add[j]=addlist;
                        }
                }
        }
}
int Person :: searchn(char key[20])
{
        int low=0,high=count,mid=0,flag=0;
        while(low<=high)
        {
                mid=(low+high)/2;
                if(strcmp(sk[mid],key)==0)
                {
                        flag=1;
                        break;
                }
                if(strcmp(key,sk[mid])<0)
                high=mid-1;
                else
                low=mid+1;
        }
        if(flag)
        return mid;
        else
        return -1;
}

void Person::delfile(int pos)
{
        char del='$';
        int i;
        fstream is;
        if(pos>=0)
        {
                is.open("p.txt",ios::in|ios::out);
                int addl=add[pos];
```

```
                is.seekp(addl,ios::beg);
                is<<del;
                cout<<"Record DELETED !!! "<<endl;
                is.close();

        }
}
void Person ::remove()
{
        char key[30];

        fstream is;
        cout<<"ENTER THE NAME :: "<<endl;
        cin>>key;
        int pos=searchn(key);
        if(pos>=0)
        {
                readf(pos);
                delfile(pos);
                int t=pos;
                while(strcmp(sk[++t],key)==0 && t<=count)
                {
                        readf(t);
                        delfile(t);
                 }
                 t=pos;
                 while(strcmp(sk[--t],key)==0)
                {
                        readf(t);
                        delfile(t);
                 }
        }
        else
        cout<<"Record Not Found!!! "<<endl;
}

void Person::readf(int pos)
{
        fstream file;
        file.open("p.txt",ios::in);
        int addl=add[pos];
        file.seekp(addl,ios::beg);
        file.getline(buffer,'#');
        cout<<"\n RECORD FOUND !!! "<<buffer;
        file.close();
}
```

```
void Person::search()
{
        int pos=0,t;
        char key[20];
        fstream file;
        cout<<"\n ENTER THE NAME TO BE SEARCH : " ;
        cin>>key;
        pos=searchn(key);
        if(pos>=0)
        {
                readf(pos);
                t=pos;
                while(strcmp(sk[++t],key)==0 && t<=count)
                readf(t);

                t=pos;
                while(strcmp(sk[--t],key)==0 && t<=count)
                readf(t);
        }
        else
        cout<<"Record Not Found!!! "<<endl;
}
void main()
{
        int choice = 1;
        clrscr();
        Person ob;
        while(choice < 4)
        {
                ostream&flush();
                cout<<"1> Insert A Record "<<endl;
                cout<<"2> Search For A Record "<<endl;
                cout<<"3> Delete A Record "<<endl;
                cout<<"4> Exit "<<endl;
                cin>> choice;
                switch(choice)
                {
                case 1: ob.input();
                        ob.pack();
                        ob.Write();
                        break;

                case 2:
                        ob.creatind();
                        ob.search();
                        break;

                case 3:
                        ob.creatind();
                        ob.remove();
                        break;
```

```
            }
        }   getch();
}
```

Input/Output :

```
1> Insert A Record
2> Search For A Record
3> Delete a record
4> Exit
1
Enter Usn
10cs010
Enter Name
shruthi
Enter Address
tumkur
Enter Branch
cse
Enter College
Cit

1> Insert A Record
2> Search For A Record
3> Delete a record
4> Exit
1
Enter Usn
10is025
Enter Name
revanth
Enter Address
gubbi
Enter Branch
ise
Enter College
sit

1> Insert A Record
2> Search For A Record
3> Delete a record
4> Exit
1
Enter Usn
10cv030
Enter Name
sonali
Enter Address
bangaluru
Enter Branch
civil
```

```
Enter College
ssit

1> Insert A Record
2> Search For A Record
3> Delete a record
4> Exit
1
Enter Usn
10ec040
Enter Name
revanth
Enter Address
shimoga
Enter Branch
ece
Enter College
cit

1> Insert A Record
2> Search For A Record
3> Delete A Record
4> Exit
2

 ENTER THE NAME TO BE SEARCH : revanth

 RECORD FOUND !!!  10ec040|revanth|shimoga|ece|cit|#
 RECORD FOUND !!!  10is025|revanth|gubbi|ise|sit|#
1> Insert A Record
2> Search For A Record
3> Delete A Record
4> Exit
3
ENTER THE NAME ::
revanth

 RECORD FOUND !!!  10ec040|revanth|shimoga|ece|cit|#    Record DELETED !!!

 RECORD FOUND !!!  10is025|revanth|gubbi|ise|sit|#     Record DELETED !!!

1> Insert A Record
2> Search For A Record
3> Delete A Record
4> Exit
```

Output:

7. Write a C++ program to read two lists of names and then match the names in the two lists using Cosequential Match based on a single loop. Output the names common to both the lists.

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
#include<string.h>
#include<stdio.h>

class coseq
{
        char list1[20][20],list2[20][20];
        int  count1,count2;

        public:
        void load();
        void sort();
        coseq();
        void match();
};

coseq::coseq()
{
        int n;
        char name[20];
        fstream fs;
        fs.open("name1.txt",ios::out|ios::app);
        cout<<"Enter hoW many name for list one: ";
        cin>>n;
        for(int i=0;i<n;i++)
        {
                fflush(stdin);
                cin>>name;
                fs<<name<<endl;
        }
        fs.close();

        fs.open("name2.txt",ios::out|ios::app);
        cout<<"Enter hoW many name for list two: ";
        cin>>n;
        for(i=0;i<n;i++)
        {
                fflush(stdin);
                cin>>name;
                fs<<name<<endl;
        }
        fs.close();
}
```

```
void coseq::load()
{
	fstream file;
	char name[20];
	 count1=-1;
	 count2=-1;
	file.open("name1.txt",ios::in);
	while(!file.eof())
	{
		file.getline(name,'\n');
		strcpy(list1[++count1],name);
	}
	file.close();

	file.open("name2.txt",ios::in);
	while(!file.eof())
	{
		file.getline(name,'\n');
		strcpy(list2[++count2],name);
	}
	file.close();
}

void coseq::sort()
{
	int i,j;
	char temp[20];
	for(i=0;i<=count1;i++)
	{
		for(j=i+1;j<=count1;j++)
		{

			if(strcmp(list1[i],list1[j])>0)
			{
				strcpy(temp,list1[i]);
				strcpy(list1[i],list1[j]);
				strcpy(list1[j],temp);
			}
		}
	 }
	 for(i=0;i<=count2;i++)
	 {
		for(j=i+1;j<=count2;j++)
		{
			if(strcmp(list2[i],list2[j])>0)
			{
				strcpy(temp,list2[i]);
				strcpy(list2[i],list2[j]);
				strcpy(list2[j],temp);
			}
		}
```

```
        }
}

void coseq::match()
{
        int i=0,j=0;
        cout.flush();
        while(i<=count1 &&j<=count2)
        {
                if(strcmp(list1[i],list2[j])==0)
                {
                        cout<<endl<<list1[i];
                        i++;
                        j++;
                }
                if(strcmp(list1[i],list2[j])<0)
                i++;
                if(strcmp(list1[i],list2[j])>0)
                j++;
        }
}
void main()
{       clrscr();
          coseq c;
          c.load();
          c.sort();
          cout<<"\nList of  Names common in both list are : ";
          c.match();
          getch();
}
```

Input/Output :

```
Enter hoW many name for list one: 4
ramu
risha
raj
revanth

Enter hoW many name for list two: 5
arun
savanth
revanth
risha
raj
List of  Names common in both list are :
raj
revanth
risha
```

Output:

8. Write a C++ program to read k Lists of names and merge them using k-way merge algorithm with k = 8.

```
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<fstream.h>
#include<stdlib.h>
#include<string.h>

class Merge
{
        char list[9][30][20];
        char outputlist[270][20];
        int k,count;
        int size[9];
        int index[9];
public:
        void sort(char l[30][20],int s);
        void merge();
        void read(char temp[30]);
        void load(char filename[20]);
        char* minValue();
        void display();
};

void Merge::read(char temp[30])
{
        int n;
        char name[20];
        fstream fs;
        fs.open(temp,ios::out|ios::app);
        cout<<"Enter hoW many name for list one: ";
        cin>>n;
        for(int i=0;i<n;i++)
        {
                fflush(stdin);
                cin>>name;
                fs<<name<<endl;
        }
        fs.close();
}

void Merge :: load(char filename[20])
{
        k = 8;
        char temp[30],buf[10];
        fstream file;
        memset(size,0,sizeof(size));
        memset(index,0,sizeof(index));
```

```
        for(int i=1; i<=k;i++)
        {
                strcpy(temp,filename);
                sprintf(buf,"%d",i);
                strcat(temp,buf);
                strcat(temp,".txt");
                read(temp);
                file.open(temp,ios::in);
                while(!file.eof())
                {
                        file.getline(temp,'\n');
                        if(file.eof())
                        break;
                        strcpy(list[i][size[i]],temp);
                        size[i]++;
                }
                file.close();
                sort(list[i],size[i]);
        }
}

void Merge :: merge()
{
        count = 0;
        char *value = minValue();
        while(value != NULL)
        {
                for(int i = 1; i<= k; i++)
                {
                        if(index[i]>= size[i])
                        continue;

                        if(strcmp(value,list[i][index[i]]) == 0)
                        index[i]++;
                }
                strcpy(outputlist[count],value);
                count++;
                value = minValue();
        }
}

char* Merge :: minValue()

{
        int t = 1;
        char *value = NULL;

        while(index[t] >= size[t] && t <= k)
        t++;

        if( t <= k)
```

```
        {
                value = list[t][index[t]];
                for(int i = t+1; i <= k; i++)
                if(strcmp(value,list[i][index[i]]) > 0 && index[i]<size[i])
                value = list[i][index[i]];
        }
        return value;
}

void Merge :: sort(char l[30][20],int s)
{
        char temp[20];
         for(int i = 0; i < s; i++)
        {
                for(int j=i+1; j<s;j++)
                {
                        if(strcmp(l[i],l[j]) > 0)
                        {
                                strcpy(temp,l[i]);
                                strcpy(l[i],l[j]);
                                strcpy(l[j],temp);

                        }
                }
        }
}

void Merge ::display()
{
        for(int i= 0; i < count; i++)
        cout<<outputlist[i]<<"   ";
}

void main()
{
        Merge m;
        char filename[]="list";
        clrscr();
        m.load(filename);
        m.merge();
        cout<<"\nNames in sorted order is :";
        m.display();
        getch();
}
```

Input/Output

```
Enter hoW many name for list : 2
MMM
 NNN
Enter hoW many name for list : 3
CCC
 DDD
AAA
 Enter hoW many name for list : 1
ZZZ
Enter hoW many name for list : 2
WWW
AAA
Enter hoW many name for list : 3
KKK
 MMM
NNN
Enter how many name for list : 2
RRR
SSS
Enter how many names for list : 2
AAA
OOO
Enter how many names for list : 2
 KK
KKKK
Names in sorted order is :
  AAA  CCC  DDD  KK  KKK  KKKK  MMM  NNN  OOO  RRR  SSS  WWW
ZZZ
```

Output:

VIVA QUESTIONS

1. What is File Structure?
2. What is a File?
3. What is a field?
4. What is a Record?
5. What is fixed length record?
6. What is RRN?
7. What is Variable length record?
8. What are the different modes of opening a file?
9. What is ifstream()?
10. What is ofstream()?
11. What is the difference between read() and getline()?
12. How to close a file? What happens if a file is not closed?
13. What is Hashing? What is its use?
14. Explain any one collision resolution technique.
15. What is Btree? What is B+tree?
16. Differentiate between Logical and Physical file
17. What is the use of seekg() and seekp()?
18. Explain the different way of write data to a file.
19. Explain the different way of write data to a file.
20. What is meant by Indexing?
21. What is multilevel indexing?
22. What is File descriptor?
23. What is Fragmentation? What is internal fragmentation?
24. What is DMA?
25. What is a delimeter?
26. Define direct access.
27. Define sequential access.
28. What is the need of packing the record before writing into the file?
29. Explain ios::trunk and ios::nocreate
30. What is the use of End-of-file (EOF)?
31. What are stdin, stdout and stderr?

32. What is Fragmentation?
33. What is data compression?
34. What are the properties of B tree?
35. How do we delete fixed length records?
36. How can we reclaim the deleted space dynamically?
37. What are the advantages and disadvantages of indexes that are too large to hold in memory?
38. What is an AVL tree?
39. H M L B Q S T N A Z P E G C V J K D I U Show B tree creation, insertion, splitting, deletion, merging and redistribution.
40. What is memset() ? Explain its parameters.
41. What is sprintf() ? Explain its parameters.
42. What is the use of tellg() and tellp()?
43. What is Boeing tree?