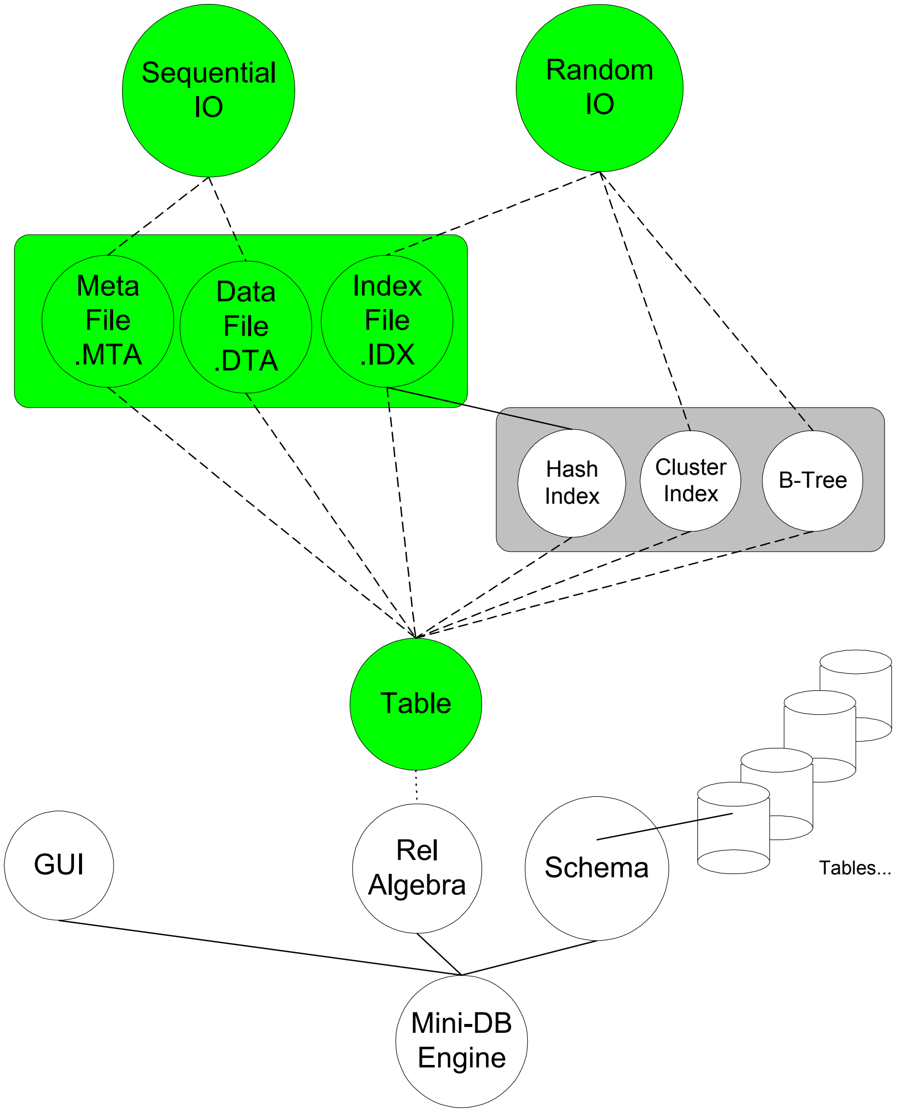
Assignment-2	**MINI DB-ENGINE (Phase I)** **Building the Base Classes**	

Our goal in Phase I of this project is to first build two simple classes for handling sequential and random access files. Once these base classes are developed, we will construct Three additional classes which will serve as the bases for constructing a database table. These classes are: Meta_File, Data_File, Index_File.   Review the following technical report http://www.cs.iusb.edu/technical_reports/TR-20071222-2.pdf section 3.1 Access Mechanism.

MINI-DB Engine



Functional Specification for MiniDB BASE classes:

Sequential IO

```
class Seq_IO
{
        char    FileName[256];
        fstream Seqfp;
        int     State;            // File state (OPEN for READ  WRITE, APPEND, or CLOSED)
        int     Verbose;          // Verbose flag (ON /OFF)
  public:
        Seq_IO(char *filename);
        ~Seq_IO();
        void EraseFile(void);
        int  OpenForWrite(void);
        int  OpenForAppend(void);
        int  OpenForRead(void);
        int  CloseFile();
        int  WriteData(char *a_record);
        int  ReadData(char *a_record, char rec_sep);
        int  ReadData(char *a_record, unsigned long record_location,char rec_sep);
        unsigned long GetCurrentFileLocation(void);
        unsigned long SetCurrentFileLocation(unsigned long location);
        unsigned long FileSize(void);
};
```

Random IO

```
class Random_IO {
        fstream RandomFile;
        int     Status;                 // OPEN or CLOSED
        int     Verbose;                // Verbose flag (ON /OFF)
 public:
        Random_IO(void);
        ~Random_IO(void);

        void Initialize(long start_address, long size, char filler_char);

        int  OpenRandom(char *file_name);
        void CloseRandom(void);

        void ReadRandom(long start_address, unsigned size, char *in_buffer);
        void WriteRandom(long start_address, unsigned size, char *out_buffer);
        void AppendRandom(unsigned size, char *out_buffer);

        unsigned long FileSize();
        void DisplayRandom(long start_address, unsigned size);
};
```

Data File (.DTA)

The data file is a sequentially organized, randomly access file. In other words, the record lengths are variable not fixed (just like a sequential file) and fields and records are separated using the "^" and "~" characters respectively. However, the data in the file should be accessed using a direct (random) access method.   This file will be used to maintain the actual data.

```
class Data_File
{
        char          FileName[256];               // *.dta
        char          RecordSeparator;             // defaulted to '~'
        char          FieldSeparator;              // defaulted to '^'
        unsigned long CurrentFileLocation;
        unsigned long EOFLocation;
        int           Verbose;                     // Verbose flag (ON / OFF)

  public:
        Data_File(char *filename,
                  char field_separator,
                  char record_separator);          //Constructor
        ~Data_File();                              // Destructor
        void SetRecordSeparator(char separator);   // Default separator = '~'
        void SetFieldSeparator(char separator);    // Default separator = '^'

        void Initialize(void);
        unsigned long GetCurrentFileLocation();
        unsigned long FileSize();

        int  WriteRecord(char *a_record);          // Write/Append a record
        int  ReadRecord(char *a_record, unsigned long record_location);
        int  ReadNextRecord(char *a_record);       // Read next record or 0 if error

        int  UnPackRecord(char *a_record, int &number_of_fields, char *fields[]);
        int  PackRecord(char *a_record, char *fields[]);
        void DumpDataFile(void);

};
```

Index File (.IDX)

The index file is direct-access (random) file.   Records in this file are fixed size and would have the following format:

```
struct idx_record {
         unsigned long Key;           // Key to search for
         unsigned long Address;       // Physical location of the record in the .DTA file
         char          Flag;          // A = ACTIVE, D=DELETED)
};

//-----------------------------------------------------------------------//
class Index_File
{
         char       FileName[256];                       // *.idx
         Random_IO  RandomFile;
         int        CurrentMaxRecords;                   // Max number of index records in the file
         int        Verbose;                             // ON / OFF

  public:
         Index_File(char *filename);                     // Constructor
         ~Index_File();                                  // Destructor, also closes the file

         void Initialize(int max_records);               // set the max records and open the file
         void Expand(int highest_record);                // Expand the size of the Index_File

         void InsertIndexRecord(idx_record *idx);        // Insert an Index record
         int  SearchIndexRecord(unsigned long Key,       // return idx record, -1 if not found
                                idx_record *idx);

         int  MaximumIndexRecords();                     // Return max number of index records
         void DumpIndexFile(void);
};
```

Meta File (.MTA)

This is a sequential file which will maintain information about our database / table/ data file. Each record in this file has the following structure:

Tag Name=	field separator (^)	filed information	field terminator(~)

```
TABLE_NM=^Table Name~
NUM_FILDS=^Number_of_Fields_In_Table~
FN=^Field Name~
FS=^Field Size~
FT=^Field Type~                               // for now don't worry about fields other
                                              than character strings.
FN=^Field Name~
FS=^Field Size~
FT=^Field Type~
```

To designate a filed as a primary key of the table, you should add a new "PK" tag and include that at the end of the table definition.

```
PK=^Dept_ID~
FS=^4~
FT=^Char~
```

Note that we can add a new tag such as "DATABASE_NM=" at the beginning of the meta-file and then proceed to maintain information about multiple tables in the same meta-file.

Hacker's Corner:

Similar to Primary keys, if a field is to be designated as the foreign key (FK) to another table, then you should add a new FK tag and include it at the end of the table definition. Note that the FK, must have a link to a foreign table, therefore, it must include extra tags for (FTN=^Foreign Table Name~ and FFN=^Foreign Field Name~) as well.

Here is an example of a department table with 3 fields (Dept_ID, Dept_Name, Dept_Mgr). The Dept_Mgr is a foreign key to the employee table.

```
TABLE_NM=^Department~
NUM_FILDS=^3~

FN=^Dept_ID~
FS=^4~
FT=^Char~

FN=^Dept_Name~
FS=^25~
FT=^Char~

FN=^Dept_Mgr~
FS=^25~
FT=^Char~

PK=^Dept_ID~
```

FS=^4~
FT=^Char~

FK=^Dept_Mgr~ **// keep it simple, the FK and PK should have the same field name**
FFN=^EmpolyeeID~
FS=^25~
FT=^Char~
FTN=^Employee~

The overall goal is to design a simple, general purpose access mechanism for a small Relational-Algebra based database. The above is the first phase of the implementation. The relational algebra part comes later!!

```
class Meta_File
{
          char          FileName[256];          // *.dta
          char          RecordSeparator;        // defaulted to '~'
          char          FieldSeparator;         // defaulted to '^'
          char          TagSeparator;
          unsigned long CurrentFileLocation;
          unsigned long EOFLocation;
          int           Verbose;                // Verbose flag (ON / OFF)



  public:
          Meta_File(char *filename,                         //Constructor
                    char field_separator,
                    char record_separator,
                    char tag_separator);
          ~Meta_File();                                     // Destructor

          void SetRecordSeparator(char separator);        // Default separator = '~'
          void SetFieldSeparator(char separator);         // Default separator = '^'
          void SetTagSeparator(char separator);           // Default separator = '='

          void Initialize(void);

          unsigned long SetCurrentFileLocation(unsigned long );
          unsigned long FileSize();

          int  WriteMetaTag(char *a_tag);                 // Write/Append a meta-tag
          int  WriteMetaRecord(char *a_record);           // Write/Append a record

          int  WriteXMLMetaTag(char *a_tag);              // Write/Append a meta-tag
          int  WriteXMLMetaRecord(char *a_record);        // Write/Append a record

          int  ReadMetaTag(char *a_tag);
          int  ReadMetaRecord(char *a_record);

          int  UnPackRecord(char *a_record, char *fields[]);  // UnPack the record into fields
          int  PackRecord(char *a_record, char *fields[]);    // Pack the fields into a record
          void DumpMetaFile(void);
};
```

Table Class

```
class Table
{
            char        TableName[256];
            Data_File   *dta;
            Meta_File   *mta;
            Index_File *idx;

            int         TotalRecords;                           // Total number of records in the data file
            int         DeletedRecords;                         // Deleted records

            int         Verbose;                                  // Verbose flag (ON / OFF)
  public:

            Table(char *tablename);                             // Constructor
            ~Table();                                           // Destructor

            void EraseTable(void);                              // Erase the table (Meta, Data, and Index files)
            int  CreateTable(char *schema);                     // Create the schema for the table
                                                                // create empty data and index files

            void OpenTable(void);                               // The table has already been created, just open it.
            void CloseTable(void);

            int  Insert(char *a_record, unsigned long key);
            int  Delete(unsigned long key);
            int  Update(char *a_new_record, unsigned long key);

            int  SearchByKey(unsigned long key);
            int  SearchByField(char *field_name, char *value);

            void Print(unsigned long key);                       // Print the record represented by the KEY.
            void Print();                                        // Print the active records in the entire table.
            void PrintSchema(void);                              // Format and Print the meta file

            void Sort();                                         // Same as reorganize
            void Reorganize();                                   // Only keep the ACTIVE records (also sorts the data)

            int    GetTotalRecords(void);                        // Get the total number of records in the data file.
            int    GetDeletedRecords(void);                      // Get the number of deleted records in the table.
            double GarbageRatio(void);                           // return DeletedRecords / TotalRecords
            void   CalculateTotalAndDeletedRecords(void);
};
```

- EraseTable

  Initialize (erase) the Meta, Data and Index files and start over.

- CreateTable

  Build the meta file by asking the user for the proper information. (See suggestions below regarding the creation of Insert_Field, Delete_Field, Update_Field functions.)

- Insert a Record

  Using the **ID,** insert the record in both the **Index** as well as the **Data** file. Note: In the index file this would be the same as random insertion. In the data file it would be the same as sequential insert. (Note that this would be a good time to check your insert information against your meta data about each field!!)

- Delete a Record

  Ask the user to provide you with the ID of the person to be deleted. Find the record in the Index file and delete it by marking the flag as DELETED. If no such record exists, provide an appropriate message to the user. (Note that the record is not actually deleted from the DAT file.

- Update a Record

  Ask the user to provide you with the ID of the person. Find the record in the Index file, display its content, allow the user to modify the content, insert the new record at the end of the data file and place the new address in the index file. Note that this causes the old record to be left in the data file with no reference to it (This will be handled later in the reorganize function). If no such record exists, provide an appropriate message to the user.

- Search_BY_KEY_ID

  Ask the user to provide you with the ID of the person. Find the record in the index file, seek to that location in the data file, sequentially read the rest of the record and display it. If no such record exists, provide an appropriate message to the user.

- Search_BY_FILEDNAME=VALUE

  Let the user to search for a value within a given field of your database!!. This is were the META file comes into the picture. Function specification may look something like..

  ```
  Search(char *table_name, char *field_name, char *field_value);
  Search("Department", "Dept_Name", "Computer Science");
  ```

- Print (id)

  Ask the user to provide you with the ID of the person. Find the record in the index file, if the record is ACTIVE seek to the appropriate location in the data file, read and print it. If no such record exists, provide an appropriate message to the user.

- Print (overloaded)

  Print the ACTIVE records in the file. Read the index file one record at a time, check the FLAG for being ACTIVE, seek to the appropriate location in the data file and print the record.

- Print Schema)

  Print the information in the meta-data file.

- Sort

  See Reorganize.

- Reorganize

  Rewrite or regenerate both the index as well as the data file, discarding the deleted records. Note: if you are using a random access file to implement your index, the reorganization will also exhibit an interesting side effect. The data file will be sorted in the process of reorganization.

Yet Another Hacker's Corner:

Create the following DDL functions to allow the user to create, delete and update the schema of your table or database.

Insert_Field should allow you to insert a new field in the meta file of a give table. Note that this operation will at least have two parts to it. First you need to modify the meta file (rewrite it!!) and, then you need to reorganize the data file to account for the fact that a new field has been added.

Delete_Field should allow the user to delete one of the fields from a given table in the meta file. Note that this is also non-trivial. You have rewrite the metafile, also you need to make sure you are not dealing with a Primary key.

Update_Field would be Delete_Field followed by an Insert_Field

What to hand in:

- **Cover page** with proper title, your name, course # and name, assignment #, date, etc....
- **Source code** (documented)
- **Sample runs** (annotated if necessary)

For this assignment you may form a team of up to 2 individuals.