ADVANCE DATABASE ORGANIZATION

PROGRAMMING ASSIGNMENT III: RECORD MANAGER

CWID	Name	Contribution (description)	Percent Contribution
A20594489	Kori Kamoljonov	Designed and implemented record insertion, deletion, and updating functionalities. Integrated the record manager with the buffer manager and ensured data consistency. Contributed to final testing and documentation.	33.3%
A20568393	Chaitanya Datta Maddukuri	Developed scanning functionality with search conditions to filter records efficiently. Created and optimized indexing strategies for faster retrieval. Assisted in debugging and performance tuning.	33.3%
A20582646	Vamshi Krishna Cheeti	Implemented table creation and schema management. Developed storage handling mechanisms to manage tables in separate page files and ensured efficient access via the buffer manager. Worked on tracking performance metrics and optimizations.	33.3%

1.TASK

The goal of this assignment is to create a simple record manager that manages tables with a predefined schema. Users should be able to add, delete, update, and scan records in a table. The scan operation is linked to a search condition, which returns only records that meet the specified criteria. Each table must be stored in its own page file, which the record manager should access using the buffer manager created in the previous assignment.

Tips for Implementing Record Manager

This assignment is significantly more complex than the previous ones, and if you don't have a clear plan for structuring your solution and selecting appropriate data structures, you might get stuck. Before you begin implementing, spend some time designing your data structures and architecture on paper.

Key Considerations:
The assignment uses fixed-length data types, so the size of each record is determined by its schema.

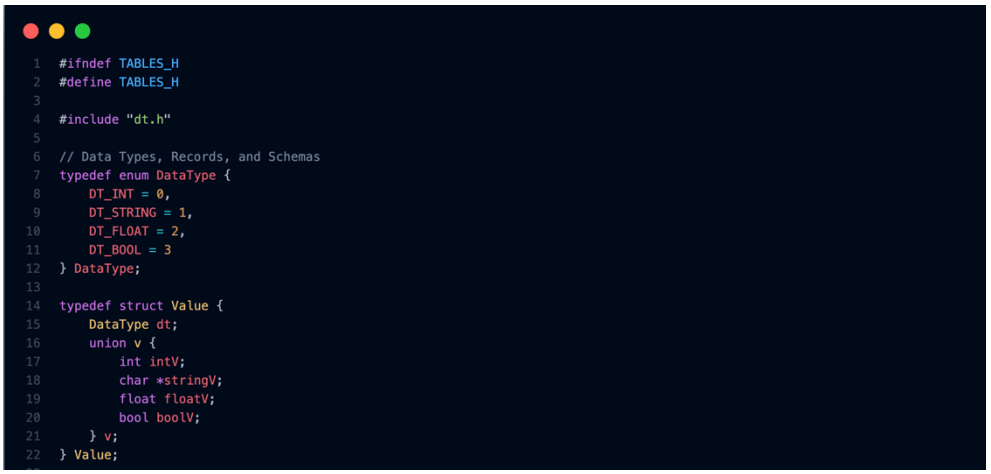
1. Page Layout: Specify how records will be arranged on pages, leaving space for managing page entries. Review the page layout concepts discussed in class, such as how to represent record slots and efficiently manage available space.
2. Allocate one or more pages in the page file to store table metadata, like the schema. Record IDs: Each record requires a unique identifier consisting of a page number and a slot number within that page.

   To support record deletion, the record manager must track available space. A simple approach is to link pages with free space by using a reserved pointer on each page that points to the next free space. Alternatively, a directory spanning multiple pages could be used to keep track of free space on each page.

2.TABLES.H

The 'tables.h' header file defines the basic data structures needed to manage schemas, tables, records, record IDs (RIDs), and values in a record manager. It also contains functions for serializing these structures to strings, which are implemented in 'rm_serializer.c'. The record manager can handle four fixed-length data types: integers ('DT_INT'), floating-point numbers ('DT_FLOAT'), fixed-length strings ('DT_STRING'), and booleans ('DT_BOOL'). Each table follows a predefined schema that specifies the number of attributes, their names, data types, and, for string attributes, fixed lengths. A schema can also define a primary key, which is an array of integers that correspond to attribute positions in the record.

2.1 Schema and binary representation. A record is made up of a **Record ID (RID)**, which includes a **page number** and a **slot number**, as well as the schema-compliant **binary representation** of its attribute values. The values are stored in a compact format using standard C data types, with an exception for string attributes. While C strings are typically null-terminated, records store fixed-length strings **without the extra null byte**, resulting in efficient space utilization. This structure enables efficient record storage, retrieval, and management while ensureing data consistency across the table's schema.

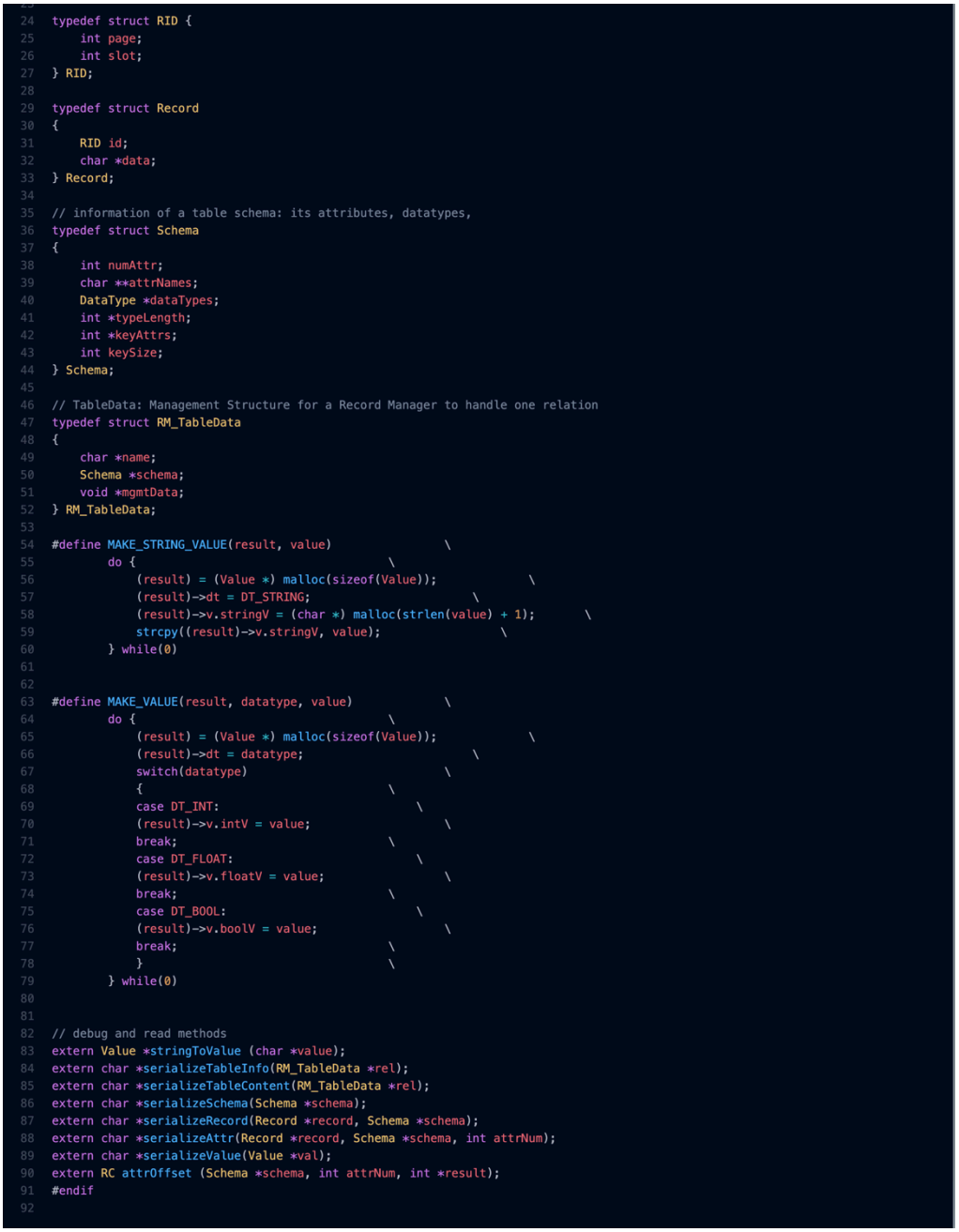


```c
#ifndef TABLES_H
#define TABLES_H

#include "dt.h"

// Data Types, Records, and Schemas
typedef enum DataType {
    DT_INT = 0,
    DT_STRING = 1,
    DT_FLOAT = 2,
    DT_BOOL = 3
} DataType;

typedef struct Value {
    DataType dt;
    union v {
        int intV;
        char *stringV;
        float floatV;
        bool boolV;
    } v;
} Value;
```



```c
typedef struct RID {
    int page;
    int slot;
} RID;

typedef struct Record
{
    RID id;
    char *data;
} Record;

// information of a table schema: its attributes, datatypes,
typedef struct Schema
{
    int numAttr;
    char **attrNames;
    DataType *dataTypes;
    int *typeLength;
    int *keyAttrs;
    int keySize;
} Schema;

// TableData: Management Structure for a Record Manager to handle one relation
typedef struct RM_TableData
{
    char *name;
    Schema *schema;
    void *mgmtData;
} RM_TableData;

#define MAKE_STRING_VALUE(result, value)                \
        do {                                            \
            (result) = (Value *) malloc(sizeof(Value));             \
            (result)->dt = DT_STRING;                           \
            (result)->v.stringV = (char *) malloc(strlen(value) + 1);       \
            strcpy((result)->v.stringV, value);                 \
        } while(0)


#define MAKE_VALUE(result, datatype, value)             \
        do {                                            \
            (result) = (Value *) malloc(sizeof(Value));             \
            (result)->dt = datatype;                            \
            switch(datatype)                                \
            {                                           \
            case DT_INT:                                    \
            (result)->v.intV = value;                       \
            break;                                  \
            case DT_FLOAT:                                  \
            (result)->v.floatV = value;                     \
            break;                                  \
            case DT_BOOL:                                   \
            (result)->v.boolV = value;                      \
            break;                                  \
            }                                           \
        } while(0)


// debug and read methods
extern Value *stringToValue (char *value);
extern char *serializeTableInfo(RM_TableData *rel);
extern char *serializeTableContent(RM_TableData *rel);
extern char *serializeSchema(Schema *schema);
extern char *serializeRecord(Record *record, Schema *schema);
extern char *serializeAttr(Record *record, Schema *schema, int attrNum);
extern char *serializeValue(Value *val);
extern RC attrOffset (Schema *schema, int attrNum, int *result);
#endif

```

3.EXPR.H

The header file 'expr.h' defines data structures and functions for handling expressions used in record scans, which are implemented in 'expr.c'. Expressions can take several forms, including **constants** (stored in a 'Value' struct), **references to attribute** values (represented by their position in the schema), and **operator invocations**.Comparison operators (such as equality and less-than) are supported for all data types, as are boolean operators (AND, OR, and NOT). Operators can accept one or more expressions as input, allowing for arbitrarily nested expressions, as long as the input types are compatible. For example, an integer constant cannot be used as an input for a boolean AND operation. This expression framework is required for defining scan conditions in the record manager, which allows for efficient filtering of records based on complex query criteria.

```c
#ifndef EXPR_H
#define EXPR_H

#include "dberror.h"
#include "tables.h"

// datatype for arguments of expressions used in conditions
typedef enum ExprType {
  EXPR_OP,
  EXPR_CONST,
  EXPR_ATTRREF
} ExprType;

typedef struct Expr {
  ExprType type;
  union expr {
    Value *cons;
    int attrRef;
    struct Operator *op;
  } expr;
} Expr;

// comparison operators
typedef enum OpType {
  OP_BOOL_AND,
  OP_BOOL_OR,
  OP_BOOL_NOT,
  OP_COMP_EQUAL,
  OP_COMP_SMALLER
} OpType;

typedef struct Operator {
  OpType type;
  Expr **args;
} Operator;

// expression evaluation methods
extern RC valueEquals (Value *left, Value *right, Value *result);
extern RC valueSmaller (Value *left, Value *right, Value *result);
extern RC boolNot (Value *input, Value *result);
extern RC boolAnd (Value *left, Value *right, Value *result);
extern RC boolOr (Value *left, Value *right, Value *result);
extern RC evalExpr (Record *record, Schema *schema, Expr *expr, Value **result);
extern RC freeExpr (Expr *expr);
extern void freeVal(Value *val);


#define CPVAL(_result,_input)           \
  do {                  \
    (_result)->dt = _input->dt;           \
  switch(_input->dt)              \
    {                 \
    case DT_INT:              \
      (_result)->v.intV = _input->v.intV;         \
      break;                \
    case DT_STRING:             \
      (_result)->v.stringV = (char *) malloc(strlen(_input->v.stringV) + 1);  \
      strcpy((_result)->v.stringV, _input->v.stringV);      \
      break;                \
    case DT_FLOAT:              \
      (_result)->v.floatV = _input->v.floatV;       \
      break;                \
    case DT_BOOL:               \
      (_result)->v.boolV = _input->v.boolV;       \
      break;                \
    }                 \
} while(0)
```

```
#define MAKE_BINOP_EXPR(_result,_left,_right,_optype)     \
    do {                \
      Operator *_op = (Operator *) malloc(sizeof(Operator));    \
      _result = (Expr *) malloc(sizeof(Expr));        \
      _result->type = EXPR_OP;            \
      _result->expr.op = _op;           \
      _op->type = _optype;            \
      _op->args = (Expr **) malloc(2 * sizeof(Expr*));      \
      _op->args[0] = _left;           \
      _op->args[1] = _right;            \
    } while (0)

#define MAKE_UNOP_EXPR(_result,_input,_optype)        \
  do {                  \
    Operator *_op = (Operator *) malloc(sizeof(Operator));    \
    _result = (Expr *) malloc(sizeof(Expr));        \
    _result->type = EXPR_OP;            \
    _result->expr.op = _op;           \
    _op->type = _optype;            \
    _op->args = (Expr **) malloc(sizeof(Expr*));      \
    _op->args[0] = _input;            \
  } while (0)

#define MAKE_ATTRREF(_result,_attr)         \
  do {                  \
    _result = (Expr *) malloc(sizeof(Expr));        \
    _result->type = EXPR_ATTRREF;         \
    _result->expr.attrRef = _attr;          \
  } while(0)

#define MAKE_CONS(_result,_value)         \
  do {                  \
    _result = (Expr *) malloc(sizeof(Expr));        \
    _result->type = EXPR_CONST;           \
    _result->expr.cons = _value;          \
  } while(0)



#endif // EXPR

```

4.RECORD_MGR.H

The recordmgr.h header file defines the record manager's interface, which is necessary for managing tables and records. The functions of the record manager are classified into five major types:

Table and Record Manager Management: These functions initiate and shut down the record manager, as well as create, open, and close tables.

This category includes functions for inserting, deleting, updating, and retrieving records from a table. Each record is identified by a Record ID (RID), which is made up of a page number and a slot number.

Scan-Related Functions: Use the expression framework (expr.h) to search for records in a table based on specific conditions. A scan operation enables the efficient retrieval of records that match a specific search condition.

Schema Management Functions: Manage schemas by creating, modifying, and retrieving details like attribute counts, names, data types, and keys.

Functions for Attribute Values and Record Creation: Utilities for extracting, modifying, and creating records based on a schema.

Each of these function types is critical to the efficient storage, retrieval, and management of records in a structured table format.

4.1 **Table and Record Manager Functionalities** Similar to previous assignments, the record manager has functions for initializing and shutting down the system. Additionally, it includes functions for **creating, opening, and closing tables**. Creating a table entails creating the underlying **page file** and storing necessary metadata, such as the **schema**, **free-space management information**, and other details in **Table Information pages**. **Opening a table** is required before performing any operations such as scanning or inserting data. Once a table is open, clients can interact with it using the 'RMTableData' structure. **Closing a table** ensures that any pending changes are written to the **page file**, which prevents data loss. Also, the function 'getNumTuples()' returns the total number of records (tuples) stored in the table.

4.2 **Recording Functions** These functions allow for essential record operations such as retrieving, deleting, inserting, and updating records in a table.
Retrieving a record: The function retrieves the corresponding record from storage using the Record ID (RID), which consists of a page number and a slot number.
Delete a record: Removes the record with the specified RID and adjusts the free-space management accordingly.
Creating a new record: The record manager assigns a unique RID to the new record and updates the record structure passed to insertRecord().
Update an existing record: Modifies a record's values while leaving its RID unchanged.

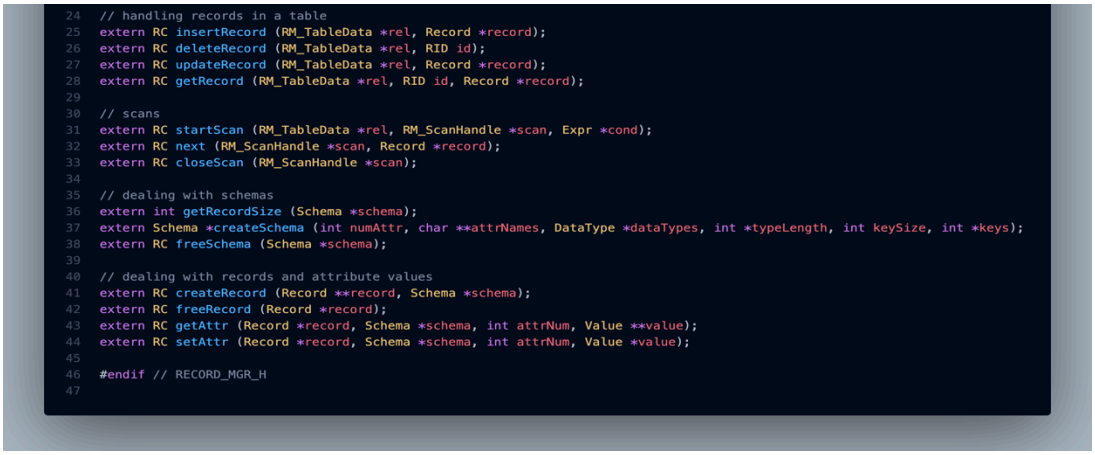
4.3 **Scan Functions.**Scanning allows clients to retrieve records from a table that meet a predefined condition (known as an Expr).A scan starts by calling startScan(), which creates an RMScanHandle structure.The client then repeatedly calls next() to retrieve the next record that meets the scan condition.If NULL is specified as the scan condition, the scan will return all records from the table.When all matching records have been retrieved, next() returns RC_RM_NO_MORE_TUPLES, signaling the end of the scan. If a record is successfully returned, it returns RC_OK, unless an error occurs.

```c
#ifndef RECORD_MGR_H
#define RECORD_MGR_H

#include "dberror.h"
#include "expr.h"
#include "tables.h"

// Bookkeeping for scans
typedef struct RM_ScanHandle
{
    RM_TableData *rel;
    void *mgmtData;
} RM_ScanHandle;

// table and manager
extern RC initRecordManager (void *mgmtData);
extern RC shutdownRecordManager ();
extern RC createTable (char *name, Schema *schema);
extern RC openTable (RM_TableData *rel, char *name);
extern RC closeTable (RM_TableData *rel);
extern RC deleteTable (char *name);
extern int getNumTuples (RM_TableData *rel);
```



```c
// handling records in a table
extern RC insertRecord (RM_TableData *rel, Record *record);
extern RC deleteRecord (RM_TableData *rel, RID id);
extern RC updateRecord (RM_TableData *rel, Record *record);
extern RC getRecord (RM_TableData *rel, RID id, Record *record);

// scans
extern RC startScan (RM_TableData *rel, RM_ScanHandle *scan, Expr *cond);
extern RC next (RM_ScanHandle *scan, Record *record);
extern RC closeScan (RM_ScanHandle *scan);

// dealing with schemas
extern int getRecordSize (Schema *schema);
extern Schema *createSchema (int numAttr, char **attrNames, DataType *dataTypes, int *typeLength, int keySize, int *keys);
extern RC freeSchema (Schema *schema);

// dealing with records and attribute values
extern RC createRecord (Record **record, Schema *schema);
extern RC freeRecord (Record *record);
extern RC getAttr (Record *record, Schema *schema, int attrNum, Value **value);
extern RC setAttr (Record *record, Schema *schema, int attrNum, Value *value);

#endif // RECORD_MGR_H
```

4.5 **Schema Functions.**Schema functions act as tools for working with table schemas. They include:
Calculating record size is a function that calculates the total size (in bytes) of a record for a given schema. Because the schema has a fixed number of attributes with predefined data types, the function calculates the necessary memory allocation based on attribute size.
Create a new schema: This function creates a schema by specifying the number of attributes, names, data types, and key attributes. It ensures that all metadata is properly stored and organized.

4.6 **Attribute Functions.**Attribute functions allow you to retrieve and modify attribute values within a record, as well as create new records that follow a specific schema.
Getting and setting attribute values: These functions allow you to access specific attributes of a record and modify them with new values.
Creating a new record: This function creates a new record by allocating enough memory in the data field to store the binary representation of all attributes specified by the schema.

5. Optional Extensions (bonus points)

You can earn up to 20% bonus points by implementing one or more optional extensions. It is recommended that you focus on implementing one or two well-defined extensions rather than trying to implement several incomplete ones. The following are the extension ideas:

Use Transaction IDs and Tombstones to track deleted records. This is a common concept in real-world systems, but your implementation may not require record relocation (because the records are fixed size). This extension improves realism in your system by handling deletions more robustly.

Enable support for SQL-style NULL values in your record manager. This entails adjusting the expression code, the Value struct, and the binary record representation. A practical solution would be to use NULL bitmaps, which can track which record attributes are null.

When inserting or updating records, ensure that primary key constraints are met. Make sure that no two records have the same value for the primary key attributes. This protects the integrity of your table's primary key by preventing duplicate key values.

To perform ordered scans, add a parameter to the scan function that sorts results by specific attributes. This can be a list of user-provided attributes that define the sort order. If you're feeling daring, you can use external sorting for large data sets, which allows your system to sort arbitrarily large tables outside of memory.

Create a simple user interface to interact with the record manager. The interface should allow users to create new tables, insert, update, and delete records, as well as perform scans. This could be a command-line shell or a menu-driven interface that allows users to perform database operations interactively.

Add support for conditional updates with scans. Create a new function, updateScan(), that accepts a condition (expression) and a callback function to update the matching records. This enables users to specify a condition (such as a filter) for which records to update and then pass a function that returns the updated record. Alternatively, you could expand the expression model to include new types of expressions (such as arithmetic operations like adding two integers), and updateScan could accept a list of expressions to apply to each matching record's attributes.

6. Source code structure

Your source code should be organized in a clear and manageable structure so that everything is well-structured and easy to build. Here is the recommended organization for your Record Manager implementation:

Create a folder called assign3.
This folder will contain all of the source code files for your record manager implementation. Make sure to keep everything related to the assignment here so it's easy to find and manage.

Reusing existing implementations:

Storage Manager: Since this is part of your previous work, move your existing storage manager implementation to this new folder.
Buffer Manager: Similarly, copy your buffer manager implementation, as you will be using it to manage table pages.

Folder Contents: Within the assign3 folder, the following files and directories should exist:

Header and C Files: Include all.h and.c files used to implement the record manager. You can begin by defining the structure of your record manager and then populating it with functions from recordmgr.h, expr.h, and other necessary header files.

Makefile: The Makefile should specify how to compile and link your project. It should be configured to compile all of your source files into an executable for the record manager.

Additional.c and.h files should contain any new implementations of record manager functionality, such as schema, record, scan, and expression handling.

README file:

README.txt or README.md: Create a markdown (.md) or plain text (.txt) file outlining your project. This should include:
A description of your records manager.
The features you've implemented.
Any optional extensions you have developed (if applicable).
Instructions for building and running the project, including any dependencies or setup steps.
assign3/

├── Makefile

├── README.md

├── storage_manager/

│ ├── storage_manager.c

│ ├── storage_manager.h

├── buffer_manager/

│ ├── buffer_manager.c

│ ├── buffer_manager.h

├── record_manager/

│ ├── recordmgr.c

│ ├── recordmgr.h

│ ├── expr.c

│ ├── expr.h

│ ├── tables.c

│ ├── tables.h

│ └── ...

7. Test Cases.

To ensure that your Record Manager functionality works properly, you must implement test cases. Here's how to arrange and implement them:

Helper Functions: (helper.h)

Define common test helpers, for example:
ASSERT_TRUE(): Determines whether a condition is true.
ASSERT_EQUAL() compares the expected and actual values.
Test Case for Expressions (test_expr.c
Evaluate expression-related functionality (expr.h).
Example: Evaluate expressions such as equality or logical operators.
Update your Makefile to generate the expr binary.
Test Case for Record Manager (test_assign3_1.c).
Test the Record Manager functionality (recordmgr.h).
For example, try inserting, deleting, updating, and scanning records.
Update your Makefile to generate the assign3 binary.