COMP 2012H Honors Object-Oriented Programming and Data Structures

Assignment 6 Simple Relational Database Management System

Honor Code

We value academic integrity very highly. Please read the Honor Code section on our course webpage to make sure you understand what is considered as plagiarism and what the penalties are. The following are some of the highlights:

- Do NOT try your "luck" - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copiee) is not just getting a zero in your assignment. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there will be additional disciplinary actions from the department and university, upto and including expulsion.

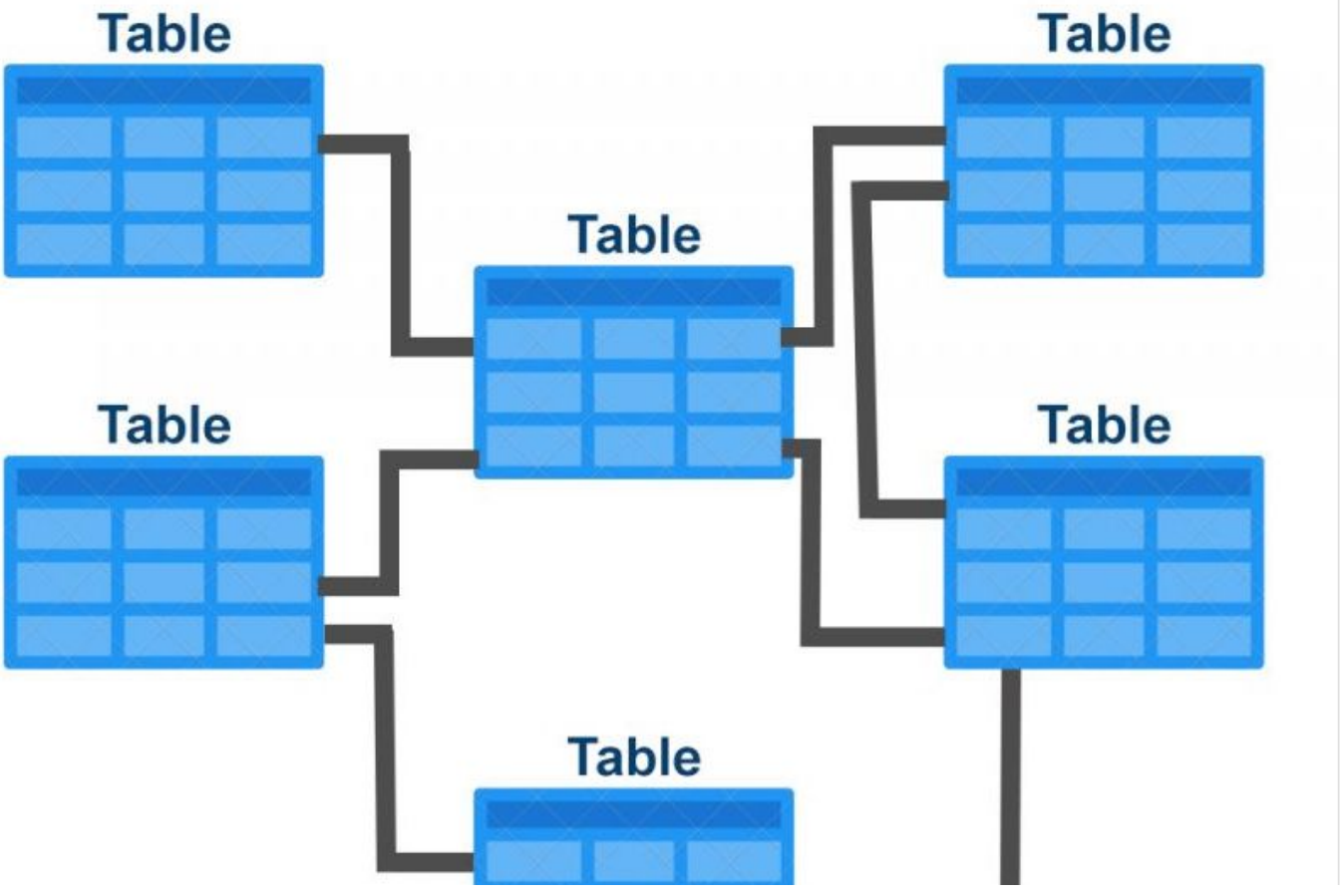
End of Honor Code

Objectives & Intended Learning Outcomes

The objective of this assignment is to familiarise you with classes and utilising their constructors & desctructors. After this assignment you should be able to:

1. Explain the structure of a program involving classes
2. Develop multiple constructors
3. Utilise constructors and desctructors for memory management
4. Perform file I/O and linked-list operations

End of Objectives & Intended Learning Outcomes



Page maintained by

Designed by Hong Wing PANG
Reviewed by SONG Sizhe
Email: sizhe.song@connect.ust.hk
Last Modified: 10/20/2022 21:41:57

Homepage

Course Homepage

Relational Database Management System

Source: https://www.stechies.com/differences-between-dbms-rdbms/

Introduction

The goal of this assignment is to implement a menu-driven Relational Database Management System (RDBMS). You can check this Wikipedia link to get an understanding of what a Relational Database entails. A relational database is a collection of data in the form of rows and columns that make up one of more tables. An RDBMS is an application used to interact with such relationship-driven databases.

RDBMS Concepts

It would be useful to familiarise yourselves with a few RDBMS concepts before beginning with the assignment:

A database can be summarised as a collection of tables. These tables can subsequently be summarised as a collection of data. The columns of the tables are referred to as attributes, fields, or simply columns. The rows, on the other hand, are called records, entries, or simply rows. Every field has a data type associated with it and a database admin can choose to enforce the types during entry. Every field also has a name. In our implementation, there will be two types of data: integer and string, but note that both types of data are stored as strings in the implementation.

Every table must also have a primary key. This is a field selected that serves as a means of uniquely identifying each record. For example, if a table has an "ID" field that is marked as a primary key, every record must be identifiable using their respective entry of the "ID" field, which means every entry in the "ID" field must be 1) **unique**, and 2) **non-empty**. In more advanced implementations, there may be multiple fields that together form the primary key.

A database supports adding, deleting, and modifying tables. It may also support operations like joining. In this assignment the inner join operation will be implemented. Inner joining involves selecting two tables and using a certain field from each table as the basis to merge the tables. For instance, if ID_1 is selected from one table, and ID_2 is selected from another table, any records from these two tables where ID_1 matches ID_2 will be merged and placed into a new table, essentially merging their common data.

A table itself supports several operations like adding and deleting fields, setting fields as the primary key, and sorting. One may also add, delete, and modify records. These operations serve to facililate data management and organisation.

Note that a database and its tables support many more applications. The functionalities of a database have been trimmed and selected for the purpose of this assignment.

End of Introduction

Description

Please read the FAQ and Change Log section regularly, and do check it one day before the deadline to make sure you don't miss any clarification, even if you have already submitted your work by then. You can also raise questions on Piazza and remember to use the "pa6" tag.

Code structure

The skeleton code structure is as follows:

```
PA6
├── Database.cpp
├── Database.h
├── Table.cpp
├── Table.h
├── String.cpp
├── String.h
└── main.cpp
```

`main.cpp` contains the Database interface;

`Database.cpp` and `Database.h` hold the Database class definition and the class function definitions respectively.

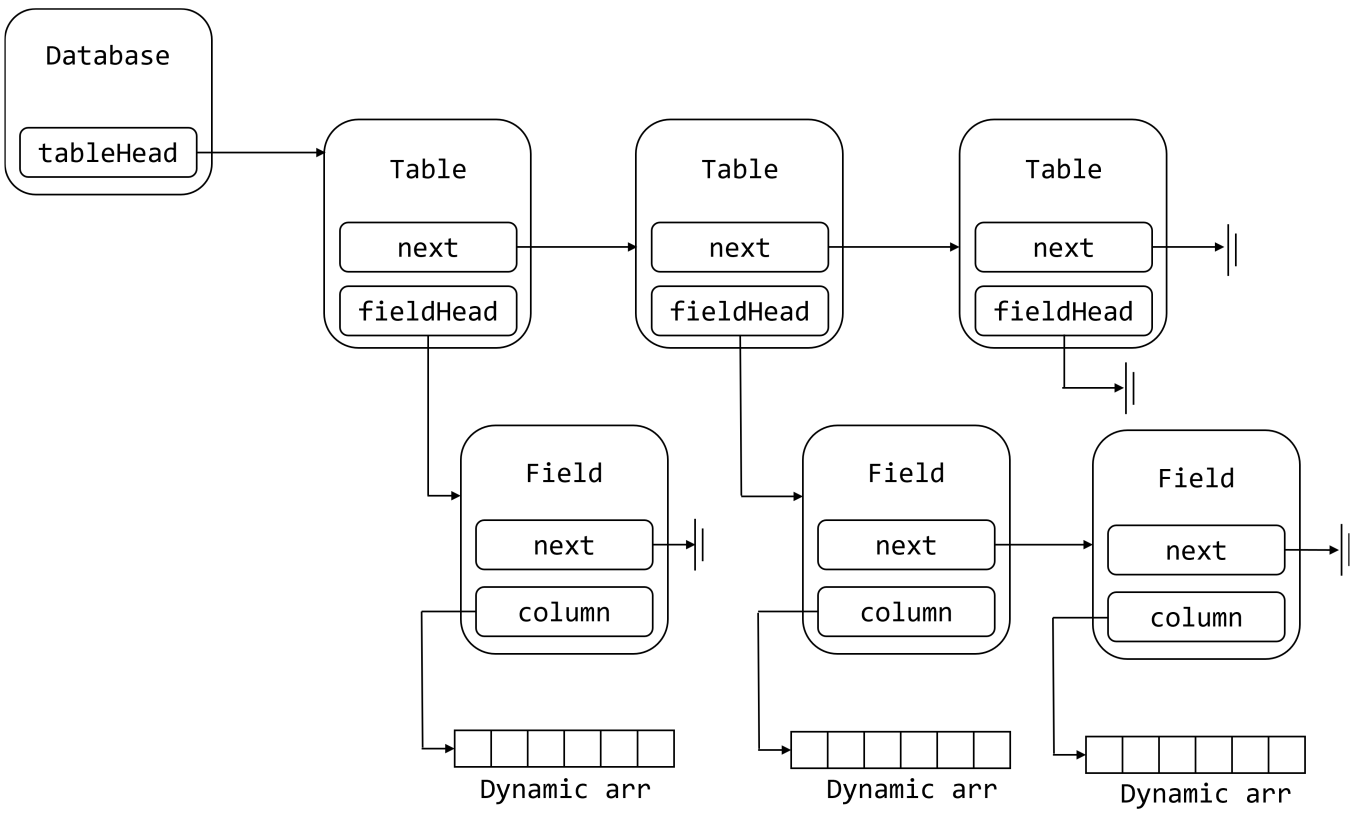
`Table.cpp` and `Table.h` hold the Table class definition and the class function definitions respectively.

`String.cpp` and `String.h` Provides the definition and functions of the String helper class.

Your job is to complete `Database.cpp` and `Table.cpp` so that the Database supports all the intended operations.

Class structure

Please go through `Database.h` and `Table.h` for the definition of the classes and structures. The following figure also gives a brief overview:



Coordinate system of individual Tables

Given a table object `TABLE`, `TABLE[i][j]` leads to a 2D array-like access of its elements. In this case, `i` is the column index, and `j` is the row index. Therefore, `TABLE[i][j]` returns the entry at (with indexing starting from 0) the `i`th column and the `j`th row. And we provide you a `operator[]` function to facilitate this access. `TABLE[5]` will return the 6th column of the table and `TABLE[5][3]` will return the entry at the 6th column and 4th row of the table (with indexing starting from 0).

Storing Data

A "save file" is a plaintext file that stores all the data of a database.

1. The first two lines are the name of the database and the number of tables it has.
2. The next three lines are the name of a table, the number of columns it has, and the number of rows it has.
3. Then comes the field names of the table, in one line, separated by a comma and a space. Every field name is followed by a single space and a number. 0 represents an

integer field and 1 represents a string field. The primary key field begins with an asterisk. The order of fields here matches the order in the table.

4. Then comes the data of the table. Each line represents a single row, and the data for each field is separated by a comma and a space. The order of the fields will match the order of the field names. The order of the rows here match the order in the table.
5. The formats specified in 2, 3, and 4 are repeated for each table. The order of tables here matches the order in the database.
6. Lastly, if a database is empty (0 table in total) it will not be followed by any table data. On the other hand, if a table is empty (0 record in total), it will not be followed by any record data.
7. There could be tables with some fields but no record. There won't be tables with no field but some records.

Sample save file:

```
Sample_DB
10
Table_1
5
5
*ID 0, Name 1, Year 0, Location 1, Major 1
2, Jack, 19, Switzerland, Science
1, John, 20, UK, Engineering
4, Alexis, 20, Israel, Fashion
3, Jill, 22, Japan, Law
0, Abraham, 18, USA, Politics
Table_2
0
0
Table_3
0
0
Table_4
0
0
Table_5
4
6
Year 0, Name 1, *SID 0, Interest 1
2, John, 3, Fishing
4, Jill, 1, Swimming
1, Abraham, 0, Snooker
1, Kimiko, 6, Programming
3, Frank, 2, Basketball
2, Alexis, 4, Football
Table_6
0
0
Table_7
0
0
Table_8
0
0
Table_9
0
0
Table_10
6
4
*CID 0, Name 1, Location 1, Currency 1, Items_Bought 0, Cart_Size 0
0, Alexis, Israel, ILS, 10, 3
1, Kimiko, Japan, JPY, 0, 25
2, Frank, Serbia, RSD, 10, 10
3, Jack, Switzerland, CHF, 100, 2
```

String Helper Class

In order to shift the focus of this assignment to classes instead of string operations, a custom written `String` class has been provided. It is functionally similar to the `std::string` STL class. As STL is not covered yet at the moment, this simpler implementation has been given.

The `String` class allows you do define and manipulate char arrays in a manner that closely resembles normal variables. It is an abstraction of the char array that handles dynamic memory allocation, and it also provides additional functions. The class structure and function descriptions are shown below. You don't need to implement this. But if you are already familiar with the `std::string` class, please note that there might be subtle differences and some functions are not implemented here. Therefore, please browse through this class at least once before proceeding with the code.

```
class String
{
private:
    char *str;

public:
    String();
    String(const char *str);
    String(const String &string);
    ~String();

    int length() const;
    friend String operator+(const String &string1, const String &string2);
    friend String operator+(const String &string1, char a);
    friend std::ostream &operator<<(std::ostream &os, const String &string);
    friend std::istream &operator>>(std::istream &is, String &string);
    friend std::istream &getline(std::istream &is, String &string, char end);
    String &operator=(const String &string);
    String substr(int begin, int count) const;
    friend bool operator==(const String &string1, const String &string2);
    friend bool operator>(const String &string1, const String &string2);
    friend bool operator<(const String &string1, const String &string2);
    friend bool operator>=(const String &string1, const String &string2);
    friend bool operator<=(const String &string1, const String &string2);
    friend bool operator!=(const String &string1, const String &string2);
    friend int strcmp(const String &string1, const String &string2); // Performs C-style comp
    char &operator[](int index); // Performs C-style accessing of characters in the String.
    const char &operator[](int index) const; // Performs C-style accessing of characters in t
    friend int stoi(const String &string);
    const char *getStr() const { return str; } // simple getter for the char array since ifst
};
```

`String` objects can be created in the following ways:

```
String name; // creates an empty String (same as String name = "" or String name(""))
String name = "Some Name";
String name("Some Name");
String another = name;
String another(name);
```

All of the above perform deep copy and dynamic memory allocation. The destructor will deallocate the dynamic memory used.

For the remainder of the String section, assume `String name = "Some name";` and `String another = "Some other name";` have been defined beforehand.

```
String anotherName = name + another; // stores "Some nameSome other name" in anotherName
String anotherName = name + "Some other name"; // stores "Some nameSome other name" in anothe
String anotherName = name + 'S'; // stores "Some nameS" in anotherName
```

```
os << name << anotherName; // outputs "Some nameSome other name" to the the output stream os
is >> name; // reads input from input stream is into name until newline character \n
```

```
getline(is, name, ','); // reads input from input stream is into name
until comma ','
```

```
String anotherName = name.substr(1, 5); // stores "ome n" in anotherName
String anotherName = another.substr(3, 7); // stores "e other" in anotherName
```

```
// Simple comparison operators
name > another // returns false because "Some other name" comes after "Some name" alphabetica
name == name // returns true because the two strings are equal
name < another // returns true for the same reason name > another returns false
```

```
stoi(name) // returns 0 because "Some name" has no integer representaion
String anotherName = "5000"
stoi(anotherName) // returns 5000
```

Important Requirements

There are a few things you CANNOT do. Failure to observe these rules would potentially result in ZERO mark for your assignment. Please carefully check your code before you submit it.

- You are NOT allowed to include any additional libraries.

End of Description

Part 1: Database

Implement the following functions of the `Database` class. Note that the functions do not need to be implemented in this order. Nor do you need to implement part 1 before part 2.

```
Database(const String &name, int numTables);
```

Create a new Database. In this function you can assume `0 <= numTables && numTables < 100`.

- Set `currentTable` to `nullptr`, `this->numTables` to be `numTables`, `this->name` to be `name`.
- Create `numTables` tables and name them as `Table_i` where `i` ranges from 1 to `numTables` (inclusive). The first table in the linked list (pointed by `tableHead`) should be `Table_1`, and so on.

```
Database(const String &filename);
```

Create a new Database using file provided. Refer to the file format.

- Set `currentTable` to `nullptr`.
- This function may need to call the overloaded table constructor.
- You can assume the file always exists and the format is correct.

- You can make use of the `>>` operator or the `getline(istream&, String&, char)` function to read data into a String object from a file.

Note: you will **need** to make use of `String::getStr()` to create the filestream object because the constructor for the filestream takes character arrays, and not our custom `String` class, as a parameter.

```
~Database();
```

The destructor that deallocates all of the `Table` objects added to the Database.

```
bool addTable(Table *table);
```

Push the provided table to the end of the `Table` linked-list.

- If `table` is a null pointer, print `Table is a null pointer.\n` and return `false`.
- If a table with the same name as `table` already exists in the database, print `Table with given name already exists in the database.\n` and return `false`.
- Otherwise push the new table to the end of the linked list, increase `numTables` and return `true`.

You may use the `==` operator to compare two String-type objects.

```
void addTable(const String &name);
```

Create a new table with name `name`.

- If a table with the provided name exists in the database, print `Table with given name already exists in the database.\n` and return.
- Otherwise, allocate a new table object, add it to the end of the linked list, and increase `numTables`.

```
Table *findTable(const String &name) const;
```

Return a pointer to the table with the name specified by `name`. Return `nullptr` if no table with the provided name exists in the database.

```
void listTables() const;
```

Print the tables in the database in this format:

```
The tables currently existing in the database are:
Table_1_Name
Table_2_Name
...
Table_LastOne_Name\n
```

```
void deleteTable(const String &name);
```

Delete the table with the given name from the database.

- If no table with the given name exists in the database, print `No such table exists in the database.\n`
- Otherwise, deallocate the table, remove it from the linked-list, and decrease `numTables`.

```
void setCurrentTable(const String &name);
```

Set `currentTable` to be the table specified by `name`.

- If no table with the given name exists in the database, set `currentTable` to be `nullptr` and print `No such table exists in the database.\n`

```
void saveDatabase(const String &filename) const;
```

Save the database, its tables, and their data to a file named `filename`, using the format in the Storing Data section. You may use the `<<` operator to output `String` objects and `int` values to the output filestream. Remember to prepend an asterisk to the primary key field. Remember to use commas as a separation between fields, and `\n` as a separation between records. Finally, remember that 0 represents a field of type `int` and 1 represents a field of type `String`.
Note: you will **need** to make use of `String::getStr()` to create the filestream object because the constructor for the filestream takes character arrays as a parameter.

```
void innerJoin(Table* table1, Table* table2);
```

This is an optional bonus task. Perform inner join on two tables' primary keys, as explained in the RDBMS ConceptsM section, create a new table and add it into the current database.

- If either of the two tables is a null pointer, print `No such table exists in the database.\n` and return.
- If two primary keys have different types, there is an immediate mismatch. Print `Type mismatch between target fields.\n` and return.
- Allocate a new table with the name `"[name of table1]+[name of table2]"`. For instance, if `table1` is named `ONE` and `table2` is named `TWO`, the new table should be named `ONE+TWO`. You can assume this name will not conflict with other tables in the database.
- The first field of the new table should be `keyName1+keyName2`. For instance, if the first primary key is `ID` and another primary key is `Name`, the new field should be named `ID+Name`. You can assume this name will not conflict with other fields in the new table. Set it as the `primaryKey` of the new table.
- The subsequent fields of the new table should be all the remaining fields of `table1`, followed by all the remaining fields of `table2`, the relative orders of field should remain. This should hence resemble the example below.

*ID 0	Name 1	Age 0
1	John	15
2	Boris	28
4	Hugh	4
6	Vanessa	13

Location 1	Age 0	*Customer_ID 0	Other_ID 0	Rating 0
France	15	1	20	4
Spain	28	2	18	4
Peru	4	3	30	5
Bahrain	4	4	35	2
Brunei	20	5	17	2

*ID+Customer_ID 0	Name 1	Age 0	Location 1	Age(T2) 0	Other_ID 0	Rating 0
1	John	15	France	15	20	4
2	Boris	28	Spain	28	18	4
4	Hugh	4	Bahrain	4	35	2

Note that if there is a field with the same name in `table1` and `table2`, there is a potential problem with duplicate field names. Hence, the corresponding field from `table1` will remain the same name, and the corresponding field from `table2` should be named `[fieldName](T2)`. See the examples in the figure.

- The entries of the new table should be the conjunction of all pairs of records from `table1` and `table2` where the data in `fieldName1` matches `fieldName2`. This can be understood

using the figure above.
- The order of rows in the new table should match their order in `table1`.
- **Push this new table into the database at the end of the linked list.**

Be careful about preserving typing, names, and the naming exceptions during name matching. You may use the `+` operator to concatenate two String objects. For example, `"dead"+"beef"` would become `"deadbeef"`.

End of Part 1

Part 2: Table

Implement the following functions of the `Table` class. Note that the functions do not need to be implemented in this order. Nor do you need to implement part 1 before part 2.

```
Table(const String &name);
```

The constructor. Create a new Table.
- Initilaise pointers to `nullptr`.
- Initialise `numRows` and `numCols` to 0, and `tableSize` to 100.
- Set table `name` as provided.

```
Table(ifstream &ifs, const String &name);
```

Overloaded Table constructor that takes in an already initialised input filstream and name. This function is related to `Database::Database(const String &filename)`.
- Set table `name` as provided.
- Initilaise pointers to `nullptr`.
- Read numCols, numRows, field names, and records, as described in the format specified the in the Storing Data section.
- Set `tableSize` to 100, in this function you can assume that number of rows will be less than 100.

Remember that the primary key field has been prepended with an asterisk and every field name is followed by a space and a number indicating its type. The separator between field names and the individual columns is a comma and a space. You may find it more useful to use `getline(istream&, String&, char)` here because the third parameter describes a delimiter and second paramter is populated with data until that delimiter.

```
~Table();
```

Deallocate all of the fields in the `Table` object. Remember to deallocate the dynamic array.

```
void addRecord(int index, String *record);
```

Adds a record to the provided row `index` of the table, `record` is an array containing the values of every field (same order as the fields themselves).

Null pointer:

- If `record` is a null pointer, print `Record is empty.\n` and return.

Dimensions mismatch:

- If the target index is greater than the number of rows in the table or smaller than 0, print `Table row index is out of bounds.\n` and return.

Primary key error:

- If primary key stored in `record` is empty (`""`) or is the same as another record, print `Empty or duplicate primary key.\n` and return.

Type mismatch:

- If a field has type `INT` but the corresponding value in `record` is not an integer, print `Cannot insert non-integer in integer field.\n` and return.
- Do validation according to the order above. For example, if there are both dimension mismatch and primary key error, you print the message of dimension mismatch and return.

Procedure:

- If the new record will not be the last row, do some moving to make space for the new record. The relative order of those original records should remain the same.
- Insert the data from the record and increase numRows.
- If `numRows` is equal to `tableSize` then:
  - Allocate a new dynamic array with 100 more positions and update `tableSize`.
  - Copy all the data from the old column to the new column.
  - Deallocate the old column and make the current field point to the new column.

You may use the `==` operator to compare two String-type objects. Additionally, you may use the provided `isInteger(const String&)` function to check whether a particular `String` object is a number or not.

```
void addField(int index, const String &name, TYPE type);
```

Allocate a new `Field` struct and place it at position `index` of the linked list.

Dimensions mismatch:

- If the target `index` is greater than `numCols` or smaller than 0, print `Table column index is out of bounds.\n` and return.

Name error:

- If a field with the given `name` already exists in the table, print `Field with given name already exists in table.\n` and return.
- Again, do validation according to the order above.

Procedure

- Allocate a new field. Additionally, allocate a `String` array and make `Field::column` point to it. Increase numCols.
- Place the field at the appropriate position in the linked-list. For instance, if the the target index is 1, set the new field's index to be the new second (index 1) column.
- If the field is of type `INT`, initialise all cells to `"0"`, otherwise initialize them to `""`.
- Additionally, if this the first field to be added to the table, set it to be the primary key.

```
void deleteRecord(int row);
```

Delete a record from the table.

- If `row` is greater than or equal to the number of rows in the table or smaller than 0, print `Table row index is out of bounds.\n` and return.
- Otherwise, delete the row indicated by `row`. This is done by remove the corresponding value in every field. If the deleted row is not the last row, do some moving to fill the blank. The relative order of the other records should remain the same.
- Decrease `numRows`.

```
void modifyRecord(int row, int column, const String &newVal);
```

Modify the cell specified by `row` and `column` to hold `newVal`.

Dimensions mismatch:

- If `row` is greater than or equal to `numRows` or less than 0, print `Table row index is out of bounds.\n` and return.
- If `column` is greater than or equal to `numCols` or less than 0, print `Table column index is out of bounds.\n` and return.

Primary key error:

- If this modification is going to trigger a primary key conflict or if the primary key is going to be empty (`""`), print `Empty or duplicate primary key.\n` and return.

Type mismatch:

- If strVal is a `string` but the column that it corresponds to is of type `INT`, print `Cannot insert non-integer in integer field.\n` and return.
- Again, do validation according to the order above.

Again, you may use the provided `isInteger(const String&)` function to check whether a

```
void setPrimaryKey(const String &name);
```

Set the field specified by `name` as the new primary key.

Errors:

- If there is no field with the name `name` in the table, print `No such field with provided name exists.\n` and return.
- If there is such a field, check the column.
  - If the column has any duplicates, print `Cannot set field with duplicate elements as primary key.\n` and return.
  - If the column has any empty data (`""`), print `Cannot set field with empty data as primary key.\n` and return.

```
void setColumnIndex(int index, const String &target);
```

Move the field specified by `target` so that it is at position `index`.

- If no field with the given name exists in the table, print `No such field with provided name exists.\n` and return.
- If `index` is greater than or equal to `numCols` or less than 0, print `Table column index is out of bounds.\n` and return.
- Again, do validation according to the order above.

```
void deleteField(const String &name);
```

Delete the field specified by `name`.

- If no field with the given name exists in the table, print `No such field with provided name exists.\n` and return.
- If the target field to be deleted is the primary key, print `Cannot delete primary key field.\n` and return.
- Otherwise, deallocate the field and the dynamic array, decrease `numCols`.

```
Field* findField(const String &name) const;
```

Return a pointer to the field specified by `name`.

- If no field with the given name exists in the table, print `No such field with provided name exists.\n` and return `nullptr`.

```
void sortTable(Field* field);
```

Sorts the rows by `field` in an ascending order. If the field has type `INT` then ascending order should be numeric. Otherwise, the ascending order should be alphabetic.

- If `field` is a null pointer, print `Invalid field provided.\n` and return.
- Otherwise you may assume this `field` exists in the table and you can start sorting.

For `String` objects you may use the comparison operators `>, <, >=, <=` to compare two Strings. These operators check for alphabetical orders. For integers, comparisons must be done after converting them to integers. This can be achieved by using the `stoi()` function that returns the numeric value of the number held in the String object. **You may assume that the field to be sorted will not contain duplicate values.**
End of Part 2

Resources & Sample I/O

- Skeleton code: PA6_skeleton.zip
- Demo programs (last update at **10/10 12:56**):
  Windows / Linux / macOS x86 / macOS arm
  (If you find the demo program doesn't behave as you expect, please first download and try the lastest version, if the problem is still there you can let me know. Thanks!)
- Sample Save File: Sample 1

If you encounter problems running the macOS demo programs, you can try look for a Windows PC on campus or use the Virtual Barn from anywhere to run the Windows demo program.

End of Resources & Sample I/O

Submission & Grading

Deadline: 22 October 2022 Saturday HKT 23:59.
Submit a zip file to ZINC. Compress the source files `Database.cpp` and `Table.cpp` directly, **not a folder containing them**.

Memory Leak

Please bear in mind that we will also check memory leak in test cases. If any of your test cases have memory leak, you will receive a 10% penalty. This penalty will be applied only once in this PA.

Grading Scheme

Task Function	**Grade**
Task 1 - `Database(const String &name, int numTables);`	3%
Task 2 - `Database(const String &filename);`	4%
Task 3 - `bool addTable(Table *table);`	5%
Task 4 - `void addTable(const String &name);`	5%
Task 5 - `Table *findTable(const String &name) const;`	3%
Task 6 - `void listTables() const;`	3%
Task 7 - `void deleteTable(const String &name);`	7%
Task 8 - `void setCurrentTable(const String &name);`	3%
Task 9 - `void saveDatabase(const String &filename) const;`	6%
Task 10 - `Table(const String &name);`	3%
Task 11 - `Table(ifstream &ifs, const String &name);`	7%
Task 12 - `void addRecord(int index, String *record);`	7%
Task 13 - `void addField(int index, const String &name, TYPE type);`	7%
Task 14 - `void deleteRecord(int row);`	4%
Task 15 - `void modifyRecord(int row, int column, const String &newVal);`	7%
Task 16 - `void setPrimaryKey(const String &name);`	5%

Task 17 - `void setColumnIndex(int index, const String &target);`	5%
Task 18 - `void deleteField(const String &name);`	5%
Task 19 - `Field* findField(const String &name) const;`	4%
Task 20 - `void sortTable(Field* field);`	7%
Bonus - `void innerJoin(Table* table1, Table* table2);`	5%

End of Submission & Grading

Change Log

- **10/12 21:41**
  1. Addtional description for the bonus task. The order of rows in the new table should match their order in `table1`. Piazza@200
- **10/11 23:23**
  1. Add an assumption in `Table::sortTable()`: the field to be sorted will not contain duplicate values so that the result will be the same no matter your sorting is stable or not. Piazza@@194
- **10/10 12:56**
  1. Fix `String::operator>()` and `String::operator<()`. You can download the new skeleton zip and replace your `String.cpp` with the new one. But generally, this update should not affect how you implement your parts.
- **10/8 01:38**
  1. Typo. Piazza@163
  2. `Table::setPrimaryKey` will check 1) if the column has empty data and 2) if the column has duplicates, while the priority is not specified. So we guarantee that these two types of errors will not exist at the same time in the test cases. Piazza@163
  3. You can assume that there will be no comma and space in field names. The figure for `Database:innerJoin()` has been updated. If you cannot see it, clear the cache of your browser and refresh the page.Piazza@163 and Piazza@164
- **10/7 23:07**
  1. Add an extra assumption to `Table::Table(ifstream...)` that the number of rows will be less than 100. Piazza@157
  2. Fix typos. Piazza@159
  3. Add a global assumption. In this PA, we will not have integer strings like "01". That means if two strings are not equal by string comparison, then it is guaranteed that their corresponding integers are not equal. Piazza@161
- **10/7 19:30**
  1. Test case 44 didn't show correct output. It has been fixed and all submissions has started to be regraded. The demo programs are also updated. Piazza@156
  2. Fix an error in the description of `Table::addField()` and `Table::deleteRecord()`. The dimension mismatch validation should all check if the index is "smaller than 0". Piazza@155_f1
- **10/7 02:10**
  1. Fix an error in the description of `Table::deleteRecord()` and `Table::setColumnIndex`. The dimension mismatch validation should check if the index is "greater than or equal to" but not ~~"greater than"~~. Piazza@155
- **10/6 13:42**
  1. Remove the two destructors from grading scheme because they are already covered by memory leak check. Their weights are transferred to other tasks.
- **10/5 15:13**
  1. Fix typos on this page. Piazza@144
  2. `Database::findTable` is now a constant member function in skeleton codes `Database.h`. Please download the skeleton codes again or simply add it yourself. Piazza@144
  3. `Table::modifyRecord` will also check if the new primary key is empty. Please read the task description again. Piazza@145

End of Change Log

Frequently Asked Questions

Q: My code doesn't work, here it is, can you help me fix it?
A: As the assignment is a major course assessment, to be fair, you are supposed to work on it by yourself and we should never finish the tasks for you. We are happy to help with explanations and advice, but we are **not allowed** to directly debug for you.

End of FAQ