Custom file systems with indexed access

CS3500 - Operating Systems Project

Team members

Bhargav	Lohith	Harshith	Aasritha
Sumanth	G.Abhishek	Pramodh	Sreevasthav
Moumitha	Yeshaswini	Jaya Vishnavi	Phalgun

UNDER THE SUPERVISION OF

Prof. Janakiram

TA: Pappu Kumar

Problem Statement

Design a lightweight system for managing structured database files with fixed-size records. The system should:

- Extend existing system calls to support the following operations :
    - Creating database files.
    - Accessing records stored in a tabular format.
    - Reading and writing records.
- Utilise an accompanying index file for each table file to :
    - Store metadata about the table structure.
    - Enable efficient direct record access based on row and column indices.

Introduction

File-Based Table Management System:

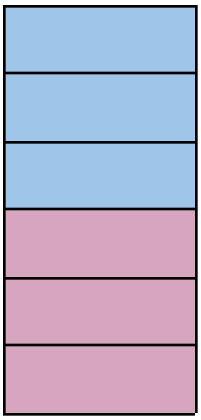
This project involves developing a file system in C that allows for the creation, deletion, reading, writing and opening of individual cells, within a table file.

Structured Metadata Management:

The system utilises an index file to maintain and manage metadata, ensuring the table's configuration is accurately tracked.

In general, the table file is typically structured in the following format:

However, our code is designed to process the table file in the following format:



Code Explanation

We have implemented a **lightweight** file-based table management system in C. It allows users to manage structured database files in a tabular format by performing the following operations:

Creation of Table Files: Allows creating new table files with a fixed number of rows, columns, and record sizes, along with an accompanying index file for metadata.

Deletion of Table Files: Deletes both the table file and its associated index file.

Opening Specific Cells in a Table: Enables direct access to a specific cell in the table by specifying its row and column indices, utilising metadata for efficient navigation.

Create_table_file:

Purpose:
Creates a table file and its corresponding index file with metadata describing the table's structure.

Code:

```c
int create_table_file(char *filename, int num_rows,int num_columns,int record_size){

    size_t file_size = num_rows*num_columns*record_size;
    int fd = open(filename,O_WRONLY|O_CREAT|O_TRUNC,0644);
    if(fd==-1){
        fprintf(stderr,"Error in creating file %s\n",filename,strerror(errno));
        return -1;
    }
    int status = ftruncate(fd,file_size);
    if(status==-1){
        fprintf(stderr,"Error in creating file %s\n",filename,strerror(errno));
        if(close(fd)==-1){
            fprintf(stderr,"Error in closing file %s\n",filename,strerror(errno));
            return -1;
        }
        return -1;
    }
    if(close(fd)==-1){
        fprintf(stderr,"Error in closing file %s\n",filename,strerror(errno));
        return -1;
    }
    char*index_suffix = "_index.txt";
    size_t index_file_len = strlen(filename)+strlen(index_suffix)+1;
    char* index_filename = malloc(index_file_len);
    if(index_filename==NULL){
        fprintf(stderr,"Unable to create index file of %s\n",filename,strerror(errno));
        return -1;
    }
    snprintf(index_filename,index_file_len,"%s%s",filename,index_suffix);

    int index_fd = open(index_filename,O_WRONLY|O_CREAT,0644);
    if(index_fd==-1){
        fprintf(stderr,"Error in creating file %s\n",index_filename,strerror(errno));
        return -1;
    }
    char buffer[50];
    snprintf(buffer,sizeof(buffer),"%d %d %d\n",num_rows,num_columns,record_size);
    ssize_t bytes_written = write(index_fd,buffer,strlen(buffer));
    if(bytes_written==-1){
        fprintf(stderr,"Error in writing to file %s\n",index_filename,strerror(errno));
        close(index_fd);
        return -1;
    }
    if(close(index_fd)==-1){
        fprintf(stderr,"Error in closing file %s\n",index_filename,strerror(errno));
        return -1;
    }
    fprintf(stdout,"Successfully created files %s & %s\n",filename,index_filename);
    return 0;
}
```

Steps:

- **Calculate File Size:** Multiply the number of rows, columns, and record size to compute the total file size.

```
size_t file_size = num_rows*num_columns*record_size;
```

- **Create and Truncate Table File:**
  - Open the table file in write-only mode, creating it if it doesn't exist otherwise truncate it (`O_WRONLY | O_CREAT | O_TRUNC`).

```
int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

  - Use `ftruncate` to set the file's size.

```
int status = ftruncate(fd,file_size);
```

- **Create Index File:**
  - Construct the index file name by appending `_index.txt` to the table filename.
  - Determine the required length of the index file name and allocate memory using `malloc`.
  - Open the index file in write-only mode:

```
int index_fd = open(index_filename, O_WRONLY | O_CREAT, 0644);
```

  - Write metadata (number of rows, columns, and record size) to the index file.

```
<number_of_rows> <number_of_columns> <record_size>
```

- **Handle Errors:**
  - Ensure proper error handling for file operations (e.g., open, write, close).
- **Clean Up:**
  - Close all open file descriptors and free dynamically allocated memory.

Time-Complexity: O(1)

Outcome:
The table file and its corresponding index file are created with the specified metadata.

Delete_table_file:

Purpose:

Deletes table file and its corresponding index file.

Code:

```c
int delete_table_file(char*filename){
    char*index_suffix = "_index.txt";
    size_t index_file_len = strlen(filename)+strlen(index_suffix)+1;
    char* index_filename = malloc(index_file_len);
    if(index_filename==NULL){
        fprintf(stderr,"Unable to create for index file of %s\n",filename,strerror(errno));
        return -1;
    }
    snprintf(index_filename,index_file_len,"%s%s",filename,index_suffix);
    if(remove(filename)!=0){
        fprintf(stderr,"Unable to delete file %s\n",filename,strerror(errno));
        return -1;
    }
    if(remove(index_filename)!=0){
        fprintf(stderr,"Unable to delete index file of %s\n",filename,strerror(errno));
        return -1;
    }
    printf("successfully deleted files %s & %s\n",filename,index_filename);
    return 0;
}
```

Steps:

- **Construct Index File Name:**
    - Append `_index.txt` to the table filename to determine the name of the index file.
    - Determine the required length of the index file name and Allocate memory using `malloc`.
- **Remove Files:** Use `remove` to delete both the table and index files.

```
remove(filename)        remove(index_filename)
```

- **Handle Errors:** Check and handle errors during file removal.

Time-Complexity: O(1)

Outcome:
On success, both the table and index files are deleted.

Open_table_file:

Description of `struct int_pair` :

The `int_pair` structure used in the `open_table_file` function is likely designed to hold two related pieces of information. In the context of the `open_table_file` function, it typically includes the following:

1. **File Descriptor**
   Represents the open file handle for the table file. This descriptor allows operations such as reading, writing, and seeking within the file.
2. **Record Size**
   Stores the size of a single record in the table, which is extracted from the metadata in the index file. This value is essential for correctly calculating offsets within the file.

```
struct int_pair{
    int fd;
    int rec_size;
}
;
```

Usage in `open_table_file`:

The `open_table_file` function uses the `int_pair` structure to return both the file descriptor and the record size to the caller.

Purpose of `open_table_file`:
Opens a table file at a specific cell and retrieves its metadata for efficient access.

Code:

```c
struct int_pair open_table_file(char* filename,int rindex,int cindex){
    struct int_pair error_pair;
    error_pair.fd = -1;
    error_pair.rec_size = 0;
    int fd = open(filename,O_RDWR);
    if(fd==-1){
        fprintf(stderr,"Unable to open table file %s\n",filename,strerror(errno));
        return error_pair;
    }
    char*index_suffix = "_index.txt";
    size_t index_file_len = strlen(filename)+strlen(index_suffix)+1;
    char* index_filename = malloc(index_file_len);
    if(index_filename==NULL){
        fprintf(stderr,"Unable to create for index file of %s\n",filename,strerror(errno));
        close(fd);
        return error_pair;
    }
    snprintf(index_filename,index_file_len,"%s%s",filename,index_suffix);
    int index_fd = open(index_filename,O_RDONLY,0644);
    if(index_fd==-1){
        fprintf(stderr,"Error in opening file %s\n",index_filename,strerror(errno));
        return error_pair;
    }
    char buffer[50];
    ssize_t bytes_read = read(index_fd,buffer,sizeof(buffer)-1);
    if(bytes_read==-1){
        fprintf(stderr,"Error in reading from file %s\n",index_filename,strerror(errno));
        close(fd);
        close(index_fd);
        return error_pair;
    }
    buffer[bytes_read] = '\0';

    if(close(index_fd)==-1){
        fprintf(stderr,"Error in closing file %s\n",index_filename,strerror(errno));
        close(fd);
        return error_pair;
    }
    int num_rows,num_cols,rec_size;
    if(sscanf(buffer,"%d %d %d",&num_rows,&num_cols,&rec_size)!=3){
        fprintf(stderr, "Error: File does not contain three numbers in the expected format\n");
        close(fd);
        return error_pair;
    }
    printf("numbers got from index file %d %d %d\n",num_rows,num_cols,rec_size);
    if(rindex>=num_rows||cindex>=num_cols){
        fprintf(stderr, "Given row and column indices exceed the file size\n");
        close(fd);
        return error_pair;
    }
    off_t byte_offset = ((rindex*num_cols)+cindex)*rec_size;
    if(lseek(fd,byte_offset,SEEK_SET)==-1){
        fprintf(stderr,"Error in opening file %s\n",index_filename,strerror(errno));
        close(fd);
        return error_pair;
    }
    struct int_pair myp;
    myp.fd = fd;
    myp.rec_size = rec_size;
    return myp;
}
```

Steps:

- **Open Table File:**
  Open the file for reading and writing using `O_RDWR`.
  - Handles errors if the file cannot be opened.

```
int fd = open(filename,O_RDWR);
```

- **Construct Index File Name:**
  - Append `_index.txt` to the table filename to locate the associated index file. Determine the required length of the index file name and Allocate memory using `malloc`.
- **Open Index File:** Open the index file in read-only mode
  - Handle errors if the file cannot be opened.
- **Read Metadata:**
  Read the first line of the index file into a buffer. Extract the number of rows, columns, and record size using `sscanf`.

```
sscanf(buffer,"%d %d %d",&num_rows,&num_cols,&rec_size)
```

- **Validate Indices:**
  Check if the provided row and column indices are within the bounds defined by the metadata.
- **Calculate Byte Offset:**
  Compute the byte offset for the specified cell using the formula:
  byte_offset=(row_index×num_columns+col_index)×record_size

```
off_t byte_offset = ((rindex*num_cols)+cindex)*rec_size;
```

- **Seek to Offset:**
  Use `lseek` to move the file pointer to the computed offset.

```
lseek(fd,byte_offset,SEEK_SET)
```

- **Return File Descriptor and Record Size:**
  Return a struct (`int_pair`) containing the file descriptor and record size.

Time-Complexity: O(1)**Outcome:** The file descriptor and record size are returned.

Future Enhancements

We implemented **integer-based row and column access**, which was efficient and sufficient for structured data management. However, as an **improvement**, we extended the system to **support string-based identifiers for rows and columns**, enhancing usability and flexibility. This update allows users to interact with the system more intuitively using meaningful labels rather than numerical indices. The main modification lies in the **format of the index file**, which now incorporates mappings between string identifiers and their corresponding positions. These changes ripple through the system, ensuring seamless integration of string-based access while retaining the core functionality.

Modified Index file format:

- The first line contains three integers: the number of rows, the number of columns, and the record size.
- The second line specifies the number of empty rows as an integer.
- The third line lists the indices of empty rows as space-separated integers in ascending order.
- The fourth line specifies the number of empty columns as an integer.
- The fifth line lists the indices of empty columns as space-separated integers in ascending order.
- The following lines contain:
    - A mapping of row numbers to strings, where each line consists of a row number followed by its corresponding string.
    - A mapping of column numbers to strings, where each line consists of a column number followed by its associated string.
    - The string is None if the column or the row is empty.

Create_string_table_file:

Purpose:

Creates 2 files, string_table_file and corresponding index file.
This function takes 4 arguments :-

1. Filename filename

2. Number of rows(num_rows)
3. Number of columns(num_columns)
4. Record size (record_size)

Code:

```c
int create_string_table_file(char *filename, int num_rows,int num_columns,int record_size){
    size_t file_size = num_rows*num_columns*record_size;
    int fd = open(filename,O_WRONLY|O_CREAT|O_TRUNC,0644);
    if(fd==-1){
        fprintf(stderr,"Error in creating file %s\n",filename);
        return -1;
    }
    int status = ftruncate(fd,file_size);
    if(status==-1){
        fprintf(stderr,"Error in creating file %s\n",filename);
        if(close(fd)==-1){
            fprintf(stderr,"Error in closing file %s\n",filename);
            return -1;
        }
        return -1;
    }
    if(close(fd)==-1){
        fprintf(stderr,"Error in closing file %s\n",filename);
        return -1;
    }
    char*index_suffix = "_index.txt";
    size_t index_file_len = strlen(filename)+strlen(index_suffix)+1;
    char* index_filename = malloc(index_file_len);
    if(index_filename==NULL){
        fprintf(stderr,"Unable to allocate space for index_filename of %s\n",filename);
        return -1;
    }
    snprintf(index_filename,index_file_len,"%s%s",filename,index_suffix);

    int index_fd = open(index_filename,O_WRONLY|O_CREAT,0644);
    if(index_fd==-1){
        fprintf(stderr,"Error in opening file %s\n",index_filename);
        return -1;
    }
```

```
    char buffer[1024];
    int len = snprintf(buffer, sizeof(buffer), "%d\n", num_rows);
    write(index_fd, buffer, len);
    for (int i = 0; i < num_rows; ++i) {
    |   len = snprintf(buffer, sizeof(buffer), "%d", i);
    |   write(index_fd, buffer, len);
    |   if(i<num_rows-1){
    |   |   write(index_fd," ",1);
    |   }else{
    |   |   write(index_fd, "\n", 1);
    |   }
    }

    len = snprintf(buffer, sizeof(buffer), "%d\n", num_columns);
    write(index_fd, buffer, len);
    for (int i = 0; i < num_columns; ++i) {
    |   len = snprintf(buffer, sizeof(buffer), "%d", i);
    |   write(index_fd, buffer, len);
    |   if(i<num_columns-1){
    |   |   write(index_fd," ",1);
    |   }else{
    |   |   write(index_fd, "\n", 1);
    |   }
    }

    for(int i=0;i<num_rows;i++){
    |   len = snprintf(buffer, sizeof(buffer), "%d None\n", i);
    |   write(index_fd, buffer, len);
    }
    for(int i=0;i<num_columns;i++){
    |   len = snprintf(buffer, sizeof(buffer), "%d None\n", i);
    |   write(index_fd, buffer, len);
    }

    if(close(index_fd)==-1){
    |   fprintf(stderr,"Error in closing file %s\n",index_filename);
    |   return -1;
    }
    fprintf(stdout,"Successfully created files %s & %s\n",filename,index_filename);
    return 0;


}
```

Code explanation:

- ***string_table_file*** **:** Used the normal **open()** system call with appropriate arguments to open a file, we need to specify the filename, flags and permissions. As we know string_table_file's size from the arguments provided, we need to make the size of the file as (no.of rows* no.of cols* record_size). For this, we use **ftruncate()** system call. The time complexity for this ftruncate() function is approximately O(1) for small files.
- ***index file*****:** Create space for the index file name. Then we need to open the file using the open() system call by sending appropriate flags and permissions and now we have to write to the index file the information we got from the arguments (following the format as mentioned before) given to the function create_string_table_file. All row and column encodings will be None because this is the time of creation of the file. At last **close()** index file after completion of writing.

Time-Complexity: O(R+C) where R is the no.of rows and C is the no.of columns.

Outcome: On successful completion we get a line saying
Successfully created files <filename> & <index_filename>

Delete_string_table_file:

Purpose:

Deletes both the main data file and its associated index file, effectively removing all components of a string table entry in the file system.

Code:

```c
int delete_string_table_file(char*filename){
    char*index_suffix = "_index.txt";
    size_t index_file_len = strlen(filename)+strlen(index_suffix)+1;
    char* index_filename = malloc(index_file_len);
    if(index_filename==NULL){
        fprintf(stderr,"Unable to allocate space for index_filename of %s\n",filename);
        return -1;
    }
    snprintf(index_filename,index_file_len,"%s%s",filename,index_suffix);
    if(remove(filename)!=0){
        fprintf(stderr,"Unable to delete file %s\n",filename);
        return -1;
    }
    if(remove(index_filename)!=0){
        fprintf(stderr,"Unable to delete index file of %s\n",filename);
        return -1;
    }
    printf("successfully deleted files %s & %s\n",filename,index_filename);
    return 0;
}
```

Steps:

- **Construct index file name:**
    - Generates the name for the index file by appending a suffix ("_index.txt") to the main file's name.
- **Delete files:**
    - Attempts to delete the main file first and then the index file, providing feedback on success or failure for each file.
- **Error Handling:**
    - Reports any failure in deleting the files (main or index) to assist in troubleshooting.
    - Ensures memory allocation is handled for constructing the index filename.

Time-Complexity: O(1)

Outcome:

Provides a success message indicating both files (filename and filename_index.txt) were deleted, confirming cleanup.

Create_row:

Purpose:

Creates a new row in the table by updating the index file with the new custom row name (stringname) and managing empty row tracking.

Code:

```c
int create_row(char* filename, char* stringname){
    char*index_suffix = "_index.txt";
    size_t index_file_len = strlen(filename)+strlen(index_suffix)+1;
    char* index_filename = malloc(index_file_len);
    if(index_filename==NULL){
        fprintf(stderr,"Unable to allocate space for index_filename of %s\n",filename);
        return -1;
    }
    snprintf(index_filename,index_file_len,"%s%s",filename,index_suffix);
    FILE* file = fopen(index_filename,"r");
    if(file==NULL){
        fprintf(stderr,"Error in opening file %s\n",index_filename);
        return -1;
    }
    int nrows,ncols,rec_size;
    fscanf(file,"%d %d %d",&nrows,&ncols,&rec_size);
    int nempty_rows;
    fscanf(file,"%d",&nempty_rows);
    if(nempty_rows==0){
        fprintf(stderr,"All rows are filled\n");
        fclose(file);
        return -1;
    }
    int empty_rows[nempty_rows];
    for(int i=0;i<nempty_rows;i++){
        fscanf(file,"%d",&empty_rows[i]);
    }
    int nempty_cols;
    fscanf(file,"%d",&nempty_cols);
    int empty_cols[nempty_cols];
    for(int i=0;i<nempty_cols;i++){
        fscanf(file,"%d",&empty_cols[i]);
    }
```

```
    int rows[nrows];
    char* row_string[nrows];
    for(int i=0;i<nrows;i++){
        row_string[i] = malloc(256*(sizeof(char)));
        fscanf(file,"%d %s",&rows[i],row_string[i]);
    }
    int columns[ncols];
    char* col_string[ncols];
    for(int i=0;i<ncols;i++){
        col_string[i] = malloc(256*(sizeof(char)));
        fscanf(file,"%d %s",&columns[i],col_string[i]);
    }
    fclose(file);
    int currrow = empty_rows[0];
    row_string[currrow] = stringname;
    nempty_rows = nempty_rows-1;
    int new_empty_rows[nempty_rows];
    for(int i=0;i<nempty_rows;i++){
        new_empty_rows[i] = empty_rows[i+1];
    }
    file = fopen(index_filename,"w");
    if(file==NULL){
        fprintf(stderr,"Error in opening file %s\n",index_filename);
        return -1;
    }
    fprintf(file,"%d %d %d\n",nrows,ncols,rec_size);
    fprintf(file,"%d\n",nempty_rows);
    for(int i=0;i<nempty_rows;i++){
        fprintf(file,"%d",new_empty_rows[i]);
        if(i<nempty_rows-1){
            fprintf(file," ");
        }else{
            fprintf(file,"\n");
        }
    }
    fprintf(file,"%d\n",nempty_cols);
    for(int i=0;i<nempty_cols;i++){
        fprintf(file,"%d",empty_cols[i]);
        if(i<nempty_cols-1){
            fprintf(file," ");
        }else{
            fprintf(file,"\n");
        }
    }
    for(int i=0;i<nrows;i++){
        fprintf(file,"%d %s\n",i,row_string[i]);
    }
    for(int i=0;i<ncols;i++){
        fprintf(file,"%d %s\n",i,col_string[i]);
    }
    fclose(file);
    return 0;
}
```

Steps:

- **Index File Access:**
  - Constructs index filename by appending _index.txt to the table filename.
  - Opens the index file in read mode to get current table metadata.
  - Reads basic table information(rows, columns, record size).
- **Empty Row Management:**
  - Reads the number of empty rows (nempty_rows).
  - Verifies if there are available empty rows.
  - Stores empty row indices in an array(the empty row indices are in ascending order).
  - Read empty column information for completeness.
- **Table Structure Reading:**
  - Reads existing row names and their indices.
  - Read column names and their indices.
  - Stores all information in memory arrays.
- **Row Creation:**
  - Takes the first available empty row index.
  - Assign row name stringname to that row.
  - Updates empty row tracking by removing the used index.
  - Decrements empty row counter.
- **Index File Update:**
  - Reopens index file in write mode.
  - Writes updated metadata:
    - Table dimensions and record size.
    - New empty row count and indices.
    - Empty column information.
    - Updated row-string mappings.
    - Column-string mappings.
- **Error Handling:**
  - Memory allocation checks.
  - File operation validations.
  - Empty row availability verification.

Time-Complexity: O(R+C), R: number of rows, C: number of columns

Outcome: Successfully creates a new row with the specified name in the first available empty row slot while maintaining proper index file structure.

Create_column:

Purpose:

Creates a new column in the table by updating the index file with the new custom column name (stringname) and managing empty column tracking.

Code:

```c
int create_column(char* filename, char* stringname){
    char*index_suffix = "_index.txt";
    size_t index_file_len = strlen(filename)+strlen(index_suffix)+1;
    char* index_filename = malloc(index_file_len);
    if(index_filename==NULL){
        fprintf(stderr,"Unable to allocate space for index_filename of %s\n",filename);
        return -1;
    }
    snprintf(index_filename,index_file_len,"%s%s",filename,index_suffix);
    FILE* file = fopen(index_filename,"r");
    if(file==NULL){
        fprintf(stderr,"Error in opening file %s\n",index_filename);
    }
    int nrows,ncols,rec_size;
    fscanf(file,"%d %d %d",&nrows,&ncols,&rec_size);
    int nempty_rows;
    fscanf(file,"%d",&nempty_rows);
    int empty_rows[nempty_rows];
    for(int i=0;i<nempty_rows;i++){
        fscanf(file,"%d",&empty_rows[i]);
    }
    int nempty_cols;
    fscanf(file,"%d",&nempty_cols);
    if(nempty_cols==0){
        fprintf(stderr,"All columns are filled\n");
        fclose(file);
        return -1;
    }
    int empty_cols[nempty_cols];
    for(int i=0;i<nempty_cols;i++){
        fscanf(file,"%d",&empty_cols[i]);
    }
```

```
    int rows[nrows];
    char* row_string[nrows];
    for(int i=0;i<nrows;i++){
        row_string[i] = malloc(256*(sizeof(char)));
        fscanf(file,"%d %s",&rows[i],row_string[i]);
    }
    int columns[ncols];
    char* col_string[ncols];
    for(int i=0;i<ncols;i++){
        col_string[i] = malloc(256*(sizeof(char)));
        fscanf(file,"%d %s",&columns[i],col_string[i]);
    }
    fclose(file);
    int currcol = empty_cols[0];
    col_string[currcol] = stringname;
    nempty_cols = nempty_cols-1;
    int new_empty_cols[nempty_cols];
    for(int i=0;i<nempty_cols;i++){
        new_empty_cols[i] = empty_cols[i+1];
    }
    file = fopen(index_filename,"w");
    if(file==NULL){
        fprintf(stderr,"Error in opening file %s\n",index_filename);
        return -1;
    }
    fprintf(file,"%d %d %d\n",nrows,ncols,rec_size);
    fprintf(file,"%d\n",nempty_rows);
    for(int i=0;i<nempty_rows;i++){
        fprintf(file,"%d",empty_rows[i]);
        if(i<nempty_rows-1){
            fprintf(file," ");
        }else{
            fprintf(file,"\n");
        }
    }
    fprintf(file,"%d\n",nempty_cols);
    for(int i=0;i<nempty_cols;i++){
        fprintf(file,"%d",new_empty_cols[i]);
        if(i<nempty_cols-1){
            fprintf(file," ");
        }else{
            fprintf(file,"\n");
        }
    }
    for(int i=0;i<nrows;i++){
        fprintf(file,"%d %s\n",i,row_string[i]);
    }
    for(int i=0;i<ncols;i++){
        fprintf(file,"%d %s\n",i,col_string[i]);
    }
    fclose(file);
    return 0;
}
```

Steps:

- **Index File Access:**
  - Constructs index filename by appending _index.txt to the table filename.
  - Opens the index file in read mode to get current table metadata.
  - Reads basic table information(rows, columns, record size).
- **Empty Column Management:**
  - Read empty row information for completeness.
  - Reads the number of empty columns (nempty_cols).
  - Verifies if there are available empty columns.
  - Stores empty column indices in an array(the empty column indices are in ascending order).
- **Table Structure Reading:**
  - Reads existing row names and their indices.
  - Read column names and their indices.
  - Stores all information in memory arrays.
- **Column Creation:**
  - Takes the first available empty column index.
  - Assign column name stringname to that column.
  - Updates empty column tracking by removing used index.
  - Decrements empty column counter.
- **Index File Update:**
  - Reopens index file in write mode.
  - Writes updated metadata:
    - Table dimensions and record size.
    - Empty row information.
    - New empty column count and indices.
    - Row-string mappings.
    - Updated column-string mappings.
- **Error Handling:**
  - Memory allocation checks.
  - File operation validations.
  - Empty column availability verification.

Time-Complexity: O(R+C), R: number of rows, C: number of columns

Outcome: Successfully creates a new column with the specified name in the first available empty column slot while maintaining proper index file structure.

Delete_row:

Purpose:

Deletes a specified row (identified by stringname) from the table by marking it as empty in the index file and updating the empty row tracking.

Code:

```c
int delete_row(char *filename,char* stringname){
    char*index_suffix = "_index.txt";
    size_t index_file_len = strlen(filename)+strlen(index_suffix)+1;
    char* index_filename = malloc(index_file_len);
    if(index_filename==NULL){
        fprintf(stderr,"Unable to allocate space for index_filename of %s\n",filename);
        return -1;
    }
    snprintf(index_filename,index_file_len,"%s%s",filename,index_suffix);
    FILE* file = fopen(index_filename,"r");
    if(file==NULL){
        fprintf(stderr,"Error in opening file %s\n",index_filename);
        return -1;
    }
    int nrows,ncols,rec_size;
    fscanf(file,"%d %d %d",&nrows,&ncols,&rec_size);
    int nempty_rows;
    fscanf(file,"%d",&nempty_rows);
    if(nempty_rows==nrows){
        fprintf(stderr,"All rows are empty\n");
        fclose(file);
        return -1;
    }
    int empty_rows[nempty_rows];
    for(int i=0;i<nempty_rows;i++){
        fscanf(file,"%d",&empty_rows[i]);
    }
    int nempty_cols;
    fscanf(file,"%d",&nempty_cols);
    int empty_cols[nempty_cols];
    for(int i=0;i<nempty_cols;i++){
        fscanf(file,"%d",&empty_cols[i]);
    }
```

```
int rows[nrows];
char* row_string[nrows];
int ispresent=0;
for(int i=0;i<nrows;i++){
    row_string[i] = malloc(256*(sizeof(char)));
    fscanf(file,"%d %s",&rows[i],row_string[i]);
    if(strcmp(stringname,row_string[i])==0){
        ispresent=1;
    }
}
if(ispresent==0){
    fprintf(stderr,"specified string is not a row\n");
    fclose(file);
    return -1;
}
int columns[ncols];
char* col_string[ncols];
for(int i=0;i<ncols;i++){
    col_string[i] = malloc(256*(sizeof(char)));
    fscanf(file,"%d %s",&columns[i],col_string[i]);
}
fclose(file);
int currrow;
for(int i=0;i<nrows;i++){
    if(strcmp(stringname,row_string[i])==0){
        currrow=i;
        break;
    }
}
char*none = "None";
row_string[currrow] = none;
nempty_rows = nempty_rows+1;
int new_empty_rows[nempty_rows];
int i=0;
int j=0;
while(j<nempty_rows-1&&empty_rows[j]<currrow){
    new_empty_rows[i] = empty_rows[j];
    i++;
    j++;
}
```

```
    new_empty_rows[i] = currrow;
    i++;
    while(j<nempty_rows-1){
        new_empty_rows[i] = empty_rows[j];
        i++;
        j++;
    }
    file = fopen(index_filename,"w");
    if(file ==NULL){
        fprintf(stderr,"Error in opening file %s\n",index_filename);
        return -1;
    }
    fprintf(file,"%d %d %d\n",nrows,ncols,rec_size);
    fprintf(file,"%d\n",nempty_rows);
    for(int i=0;i<nempty_rows;i++){
        fprintf(file,"%d",new_empty_rows[i]);
        if(i<nempty_rows-1){
            fprintf(file," ");
        }else{
            fprintf(file,"\n");
        }
    }
    fprintf(file,"%d\n",nempty_cols);
    for(int i=0;i<nempty_cols;i++){
        fprintf(file,"%d",empty_cols[i]);
        if(i<nempty_cols-1){
            fprintf(file," ");
        }else{
            fprintf(file,"\n");
        }
    }
    for(int i=0;i<nrows;i++){
        fprintf(file,"%d %s\n",i,row_string[i]);
    }
    for(int i=0;i<ncols;i++){
        fprintf(file,"%d %s\n",i,col_string[i]);
    }
    fclose(file);
    return 0;
}
```

Steps:

- **Index File Access:**
  - Constructs index filename by appending _index.txt to the table filename.
  - Open the index file in read mode.
  - Reads basic table information (rows, columns, record size).
  - Reads empty row and column information and stores them.
- **Table Structure Reading:**
  - Read all row names and their indices.
  - Stores row information in memory arrays.
  - Reads and stores column information.
  - Identifies target row deletion.
- **Row Deletion:**
  - Marks the target row as None.
  - Increments empty row counter.
  - Creates a new empty row array including the newly deleted row.
  - Maintains sorted order of empty row indices.
- **Index File Update:**
  - Reopens index file in write mode.
  - Writes updated metadata:
    - Table dimensions and record size.
    - New empty row count and sorted indices.
    - Empty column information.
    - Updated row-string mappings.
    - Column-string mappings.
- **Error Handling:**
  - Memory allocation checks.
  - File operation validations.
  - Row existence verification.
  - Empty table checks.

Time-Complexity: O(R+C), R: number of rows, C: number of columns

Outcome: Successfully marks the specified row as deleted and updates the index file to reflect the change while maintaining proper empty row tracking.

Delete_column:

Purpose:

Deletes a specified column (identified by stringname) from the table by marking it as empty in the index file and updating the empty column tracking.

Code:

```c
int delete_column(char *filename,char* stringname){
    char*index_suffix = "_index.txt";
    size_t index_file_len = strlen(filename)+strlen(index_suffix)+1;
    char* index_filename = malloc(index_file_len);
    if(index_filename==NULL){
        fprintf(stderr,"Unable to allocate space for index_filename of %s\n",filename);
        return -1;
    }
    snprintf(index_filename,index_file_len,"%s%s",filename,index_suffix);
    FILE* file = fopen(index_filename,"r");
    if(file==NULL){
        fprintf(stderr,"Error in opening file %s\n",index_filename);
        return -1;
    }
    int nrows,ncols,rec_size;
    fscanf(file,"%d %d %d",&nrows,&ncols,&rec_size);
    int nempty_rows;
    fscanf(file,"%d",&nempty_rows);
    int empty_rows[nempty_rows];
    for(int i=0;i<nempty_rows;i++){
        fscanf(file,"%d",&empty_rows[i]);
    }
    int nempty_cols;
    fscanf(file,"%d",&nempty_cols);
    if(nempty_cols==ncols){
        fprintf(stderr,"All columns are empty\n");
        fclose(file);
        return -1;
    }
    int empty_cols[nempty_cols];
    for(int i=0;i<nempty_cols;i++){
        fscanf(file,"%d",&empty_cols[i]);
    }
    int rows[nrows];
    char* row_string[nrows];
    for(int i=0;i<nrows;i++){
        row_string[i] = malloc(256*(sizeof(char)));
        fscanf(file,"%d %s",&rows[i],row_string[i]);
    }
```

```
int columns[ncols];
char* col_string[ncols];
int ispresent = 0;
for(int i=0;i<ncols;i++){
    col_string[i] = malloc(256*(sizeof(char)));
    fscanf(file,"%d %s",&columns[i],col_string[i]);
    if(strcmp(stringname,col_string[i])==0){
        ispresent=1;
    }
}
if(ispresent==0){
    fprintf(stderr,"specified string is not a column\n");
    fclose(file);
    return -1;
}

fclose(file);
int currcol;
for(int i=0;i<ncols;i++){
    if(strcmp(stringname,col_string[i])==0){
        currcol=i;
        break;
    }
}
char*none = "None";
col_string[currcol] = none;
nempty_cols = nempty_cols+1;
int new_empty_cols[nempty_cols];
int i=0;
int j=0;
while(j<nempty_cols-1&&empty_cols[j]<currcol){
    new_empty_cols[i] = empty_cols[j];
    i++;
    j++;
}
new_empty_cols[i] = currcol;
i++;
while(j<nempty_cols-1){
    new_empty_cols[i] = empty_cols[j];
    i++;
    j++;
}
```

```
    file = fopen(index_filename,"w");
    if(file==NULL){
        fprintf(stderr,"Error in opening file %s\n",index_filename);
        return -1;
    }
    fprintf(file,"%d %d %d\n",nrows,ncols,rec_size);
    fprintf(file,"%d\n",nempty_rows);
    for(int i=0;i<nempty_rows;i++){
        fprintf(file,"%d",empty_rows[i]);
        if(i<nempty_rows-1){
            fprintf(file," ");
        }else{
            fprintf(file,"\n");
        }
    }
    fprintf(file,"%d\n",nempty_cols);
    for(int i=0;i<nempty_cols;i++){
        fprintf(file,"%d",new_empty_cols[i]);
        if(i<nempty_cols-1){
            fprintf(file," ");
        }else{
            fprintf(file,"\n");
        }
    }
    for(int i=0;i<nrows;i++){
        fprintf(file,"%d %s\n",i,row_string[i]);
    }
    for(int i=0;i<ncols;i++){
        fprintf(file,"%d %s\n",i,col_string[i]);
    }
    fclose(file);
    return 0;
}
```

Steps:

- **Index File Access:**
    - Constructs index filename by appending _index.txt to the table filename.
    - Open the index file in read mode.
    - Reads basic table information (rows, columns, record size).
    - Reads empty row and column information and stores them.
- **Table Structure Reading:**
    - Read all row names and their indices.

    - Read column names and their indices.
    - Stores column information in memory arrays.
    - Identifies target column deletion.
- **Column Deletion:**
    - Marks the target column as None.
    - Increments empty column counter.
    - Creates a new empty column array including the newly deleted column.
    - Maintains sorted order of empty column indices.
- **Index File Update:**
    - Reopens index file in write mode.
    - Writes updated metadata:
        - Table dimensions and record size.
        - Empty row information.
        - New empty column count and sorted indices
        - Row-string mappings.
        - Updated column-string mappings.
- **Error Handling:**
    - Memory allocation checks.
    - File operation validations.
    - Column existence verification.
    - Empty table checks.

Time-Complexity: O(R+C), R: number of rows, C: number of columns

Outcome: Successfully marks the specified column as deleted and updates the index file to reflect the change while maintaining proper empty column tracking.

Open_string_table_file:

Description of struct FILE_ID:

```
struct File_ID {
  int fd;
  int row_index;
  int col_index;
  int rec_size;
  int nrows;
  int ncols;
};
```

Purpose of struct File_ID:

This struct is used as a modified File descriptor with additional details that are required for using string table file.

1. int fd:
    a. The file descriptor for the open file. It is used to perform file operations (e.g., read, write, seek) on the file.
    b. If the file fails to open or there is an error, this value is set to -1.
2. int row_index:
    a. The index in the table corresponds to the specified row name (rname).
    b. If the specified row name is not found, this value is set to -1.
3. int col_index:
    a. The index in the table corresponds to the specified column name (cname).
    b. If the specified column name is not found, this value is set to -1.
4. int rec_size:
    a. The size of a single record (in bytes) in the file. This value is used to calculate the byte offset for accessing specific data in the file.
5. int nrows:
    a. The total number of rows in the string table file.
6. int ncols:
    a. The total number of columns in the string table file.

Purpose of the open_string_table_file function:
Opens the table file at a specified index which takes three arguments, a filename, 2 strings corresponding to the row and column respectively and returns a file descriptor of type struct File_ID (containing other metadata of the record at that location). It abstracts the complexity of file parsing, validation, and navigation, making it easier for other parts of the program to work with table files.

Code:

```c
struct File_ID open_string_table_file(char* filename,char* rname,char* cname){
    struct File_ID error;
    error.fd = -1;
    error.col_index = -1;
    error.row_index = -1;
    error.rec_size = -1;
    error.ncols = -1;
    error.nrows = -1;
    int fd = open(filename,O_RDWR);
    if(fd==-1){
        fprintf(stderr,"Unable to open table file %s\n",filename);
        return error;
    }
    char*index_suffix = "_index.txt";
    size_t index_file_len = strlen(filename)+strlen(index_suffix)+1;
    char* index_filename = malloc(index_file_len);
    if(index_filename==NULL){
        fprintf(stderr,"Unable to allocate space for index_filename of %s\n",filename);
        return error;
    }
    snprintf(index_filename,index_file_len,"%s%s",filename,index_suffix);
    FILE* file = fopen(index_filename,"r");
    if(file==NULL){
        fprintf(stderr,"Error in opening file %s\n",index_filename);
        return error;
    }
    int nrows,ncols,rec_size;
    fscanf(file,"%d %d %d",&nrows,&ncols,&rec_size);
    int nempty_rows;
    fscanf(file,"%d",&nempty_rows);
    int empty_rows[nempty_rows];
    for(int i=0;i<nempty_rows;i++){
        fscanf(file,"%d",&empty_rows[i]);
    }
```

```
int nempty_cols;
fscanf(file,"%d",&nempty_cols);
if(nempty_cols==ncols){
    fprintf(stderr,"All columns are empty\n");
    fclose(file);
    return error;
}
int empty_cols[nempty_cols];
for(int i=0;i<nempty_cols;i++){
    fscanf(file,"%d",&empty_cols[i]);
}
int rows[nrows];
int rownum = -1;
char* row_string[nrows];
for(int i=0;i<nrows;i++){
    row_string[i] = malloc(256*(sizeof(char)));
    fscanf(file,"%d %s",&rows[i],row_string[i]);
    if(strcmp(rname,row_string[i])==0){
        rownum = i;
    }
}
if(rownum==-1){
    fprintf(stderr,"specified string is not a row\n");
    fclose(file);
    return error;
}
int columns[ncols];
char* col_string[ncols];
int colnum = -1;
for(int i=0;i<ncols;i++){
    col_string[i] = malloc(256*(sizeof(char)));
    fscanf(file,"%d %s",&columns[i],col_string[i]);
    if(strcmp(cname,col_string[i])==0){
        colnum = i;
    }
}
```

```
    if(colnum==-1){
        fprintf(stderr,"specified string is not a column\n");
        fclose(file);
        return error;
    }

    fclose(file);

    off_t byte_offset = ((rownum*ncols)+colnum)*rec_size;
    if(lseek(fd,byte_offset,SEEK_SET)==-1){
        fprintf(stderr,"Error in changing file descripter's offset for file %s\n",index_filename);
        close(fd);
        return error;
    }
    struct File_ID myqd;
    myqd.fd = fd;
    myqd.row_index = rownum;
    myqd.col_index = colnum;
    myqd.rec_size = rec_size;
    myqd.nrows = nrows;
    myqd.ncols = ncols;
    return myqd;
}
```

Steps:

- Creating an error File descriptor for returning if any error is encountered in the process with all attributes “-1”.
- Opening the string table file specified by input filename, returns error File_ID, if any open of the files and allocations is unsuccessful.
- Empty rows information and empty columns information is read from the index file and is stored.
- The encodings of the rows and columns are read from the corresponding index file and stored.
- Calculate the row number(rownum) corresponding to the input string rname if the input is valid, else returning error File_ID.
- Calculating the column number(colnum) corresponding to the input string cname if the input is valid, else returning error File_ID.
- Calculating the bite_offset from the obtained values of row number(rownum), column number(colnum), number of columns(ncols) and record size(rec_size).
    - **Bite_offset = ((rownum*ncols)+colnum)*rec_size.**
- Get the file descriptor at the corresponding row and column using lseek and calculated bite_offset. If it is unsuccessful, we return error File_ID.
- Building the corresponding FILE_ID with other attributes and returning it.

Time Complexity: O(R+C), R: number of rows, C: number of columns.

Read_table:

Purpose:

This function reads a single record from the file based on metadata provided by input, which is of type struct File_ID. It abstracts the details of memory allocation and file operations, ensuring efficient and error-resilient access to the table's data.

Code:

```
void* read_table(struct File_ID fid) { // Time complexity = O(rec_size)
    void *record = malloc(fid.rec_size);
    if (!record) {
        fprintf(stderr, "Error: Memory allocation failed\n");
        return NULL;
    }
    if (read(fid.fd, record, fid.rec_size) != fid.rec_size) {
        fprintf(stderr, "Error: Failed to read record from file\n");
        free(record);
        return NULL;
    }
    return record;
}
```

Steps:

- **Allocate Memory for the Record:**
  - Dynamically allocates memory to hold the record of size fid.rec_size, ensuring the program can handle records of varying sizes.
  - Ensures error handling in case memory allocation fails.
- **Read Data from the File:**
  - Reads exactly fid.rec_size bytes from the file, starting at the current file offset, into the allocated memory.
  - Uses the file descriptor (fid.fd) to perform the read operation, which is already positioned at the correct offset (as calculated by other parts of the program, such as open_string_table_file).
- **Error Handling:**
  - If the read operation fails or does not read the expected number of bytes, the function reports the error and frees the allocated memory to avoid leaks.
- **Return the Retrieved Record:**

- If the operation is successful, the function returns a pointer to the allocated memory containing the record data.
- The caller can then process or interpret this data as needed.

Note: Here, the returning record is allocated using malloc, so we need to free the memory after using it elsewhere.

Time-Complexity: O(rec_size), rec_size: size of the record.

Write_table:

Purpose:

Writes data to a specific position in a file. Its signature is as below

```
int write_table(struct File_ID fid, void *buffer, size_t size);
```

As we know fid contains fd, row_index, col_index, rec_size, nrows, ncols and buffer is the pointer to the data that needs to be written to the file and size is the no.of bytes to write from buffer.

Return Value:

Returns the number of bytes written on success, else returns -1.

Code:

```c
int write_table(struct File_ID fid, void *buffer, size_t size) { // Time complexity = O(size)
    int pos = lseek(fid.fd, 0, SEEK_CUR);
    if (pos == -1) {
        fprintf(stderr, "Error: Failed to get current file position\n");
        return -1;
    }

    int used = pos - ((fid.row_index * fid.ncols) + fid.col_index) * fid.rec_size;
    int available = fid.rec_size - used;

    if (size > available) {
        fprintf(stderr, "Warning: Record size exceeds available space; only %d bytes will be written\n", available);
        ssize_t written = write(fid.fd, buffer, available);
        if (written == -1) {
            fprintf(stderr, "Error: Failed to write to file\n");
        }
        return written;
    }

    ssize_t written = write(fid.fd, buffer, size);
    if (written == -1) {
        fprintf(stderr, "Error: Failed to write to file\n");
    }
    return written;
}
```

Code explanation:

When we want to write something to the file, we need to know where we are currently before doing the write operation. For this we need **lseek()**.

```c
lseek(fid.fd, 0, SEEK_CUR)
```

This lseek() is used to reposition the read/write file offset associated with a file descriptor. This offset determines where the next read/write operation will occur in the file. SEEK_CUR gives the curr position and 0 is the offset. So the above line returns the current position of the file descriptor and also moves the fd to that position. Time Complexity for lseek() is O(1).

The variable used calculates the number of bytes already written in the current record by determining how far into the record the current position is. So, the available space for writing in that particular record will be (fid.rec_size - used). If the argument size is greater than the available space then we can write only some bytes. So, In the case of size > available we write only the first few(= available) bytes from buffer to file and give a warning saying that

Warning: Record size exceeds available space; only <available> bytes will be written

For writing to file we use general write() which takes fid.fd, buffer, available as arguments and returns the number of bytes written successfully to the file.

So, for appending when the user writes to a file in a particular record and again after some time the user asks to write to that same record then we are appending because the file descriptor goes to a position in the record till where the data is written before.

Time-Complexity: O(size) where size is the number of bytes to be written from the buffer.

Table File Operations

This project provides a library and a set of executables for working with file-based tables. It allows creating, deleting, and managing table files, rows, and columns, along with user-accessible functions for reading and writing records. The project ensures that all functions and binaries can be used system-wide, just like standard utilities.

Features:

Library Functions:

1. **Table File Operations**
    - **Create**

      ```
      int create_string_table_file(char *filename, int
      num_rows, int num_columns, int record_size);
      ```

    - **Delete**

      ```
      int delete_string_table_file(char *filename);
      ```

2. **Row and Column Management**
    - **Create Row**

      ```
      int create_row(char *filename, char *row_name);
      ```

- **Delete Row**

  ```
  int delete_row(char *filename, char *row_name);
  ```

- **Create Column**

  ```
  int create_column(char *filename, char *column_name);
  ```

- **Delete Column**

  ```
  int delete_column(char *filename, char *column_name);
  ```

3. **Open, Read, Write Records**

   - **Open**

     ```
     struct File_ID open_string_table_file(char *filename,
     char *row_name, char *column_name);
     ```

   - **Read**

     ```
     void* read_table(struct File_ID fid);
     ```

   - **Write**

     ```
     int write_table(struct File_ID fid, void *buffer,
     size_t size);
     ```

Executable Binaries:

Six executables are provided:

- **`create_table_file`**: Create a Table file.
- **`delete_table_file`**: Delete a Table file.
- **`create_row`**: Add a new row to the Table file.
- **`delete_row`**: Remove a row from the Table file.
- **`create_column`**: Add a new column to the Table file.
- **`delete_column`**: Remove a column from the Table file.

Setup Process:

1. Building the Static Library

To allow global usage of `#include <table_file.h>` in any program, we build a static library and place it in the system's library path.

Steps:

1. Compile `table_file.c` to an object file:

```
gcc -c table_file.c -o table_file.o
```

2. Create the static library:

```
ar rcs libtable_file.a table_file.o
```

3. Move the static library to `/usr/local/lib`:

```
sudo mv libtable_file.a /usr/local/lib/
```

4. Move the header file to `/usr/local/include`:

```
sudo cp table_file.h /usr/local/include/
```

5. Update the linker cache:

```
sudo ldconfig
```

6. Change Permission to permit read for all users:

```
sudo chmod 644 /usr/local/include/table_file.h
```

2. Compilation of user program

Once the library and header are in place, you can use `#include <table_file.h>` in your programs.

For example: `main.c`

Compile:

```
gcc -o test_program main.c -ltable_file
```

3. Compiling the Executable Binaries

Steps:

- **Compile each utility with the library**
    - `gcc -o create_table_file create_table_file.c -ltable_file`

```c
#include<table_file.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
    char *filename = argv[1];
    int num_rows = atoi(argv[2]);
    int num_columns = atoi(argv[3]);
    int record_size = atoi(argv[4]);
    create_string_table_file(filename, num_rows, num_columns, record_size);
    return 0;
}
```

This code is a command-line program that creates a new table file with specified dimensions. It takes four command-line arguments - the filename, number of rows, number of columns, and record size. The program converts the numeric arguments to integers using atoi() and calls create_string_table_file() from `<table_file.h>` to create a new table file with the given specifications.

    - `gcc -o delete_table_file delete_table_file.c -ltable_file`

```c
#include<table_file.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
    char *filename = argv[1];
    delete_string_table_file(filename);
    return 0;
}
```

This code is a straightforward command-line program that deletes an entire table

file and its associated index file. It takes one command-line argument - the filename of the table to delete. The program uses the delete_string_table_file() function from `<table_file.h>` to completely remove both the main table file and its corresponding index file from the system.

- `gcc -o create_row create_row.c -ltable_file`

```c
#include <table_file.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    char *filename = argv[1];
    char *row_name = argv[2];
    create_row(filename, row_name);
    return 0;
}
```

This code is a simple command-line program that creates a new row in a table file. It takes two arguments from the command line - the table filename and the name of the row to create. The program uses the create_row() function from `<table_file.h>` to add the new row to the specified table file.

- `gcc -o delete_row delete_row.c -ltable_file`

```c
#include <table_file.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    char *filename = argv[1];
    char *row_name = argv[2];
    delete_row(filename, row_name);
    return 0;
}
```

This code is a command-line program that deletes a specific row from an existing table file. It takes two command-line arguments - the table filename and the name of the row to delete. The program uses the delete_row() function from `<table_file.h>` to remove the specific row from the specified table file.

- `gcc -o create_column create_column.c -ltable_file`

```c
#include <table_file.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    char *filename = argv[1];
    char *column_name = argv[2];
    create_column(filename, column_name);
    return 0;
}
```

This code is a simple command-line program that creates a new column in a table file. It takes two arguments from the command line - the table filename and the name of the column to create. The program uses the create_column() function from `<table_file.h>` to add the new column to the specified table file.

- `gcc -o delete_column delete_column.c -ltable_file`

```c
#include <table_file.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    char *filename = argv[1];
    char *column_name = argv[2];
    delete_column(filename, column_name);
    return 0;
}
```

This code is a command-line program that deletes a specific column from an existing table file. It takes two command-line arguments - the table filename and the name of the column to delete. The program uses the delete_column() function from `<table_file.h>` to remove the specific column from the specified table file.

- **Move the executables to `/usr/local/bin` for global access**:
  - `sudo mv create_table_file /usr/local/bin/`

- `sudo mv delete_table_file /usr/local/bin/`
- `sudo mv create_row /usr/local/bin/`
- `sudo mv delete_row /usr/local/bin/`
- `sudo mv create_column /usr/local/bin/`
- `sudo mv delete_column /usr/local/bin/`

Usage Instructions:

1. **Table File Management**
   - Create a Table File:

     ```
     create_table_file <filename> <num_rows> <num_columns>
     <record_size>
     ```

   - Delete a Table File:

     ```
     delete_table_file <filename>
     ```

2. **Row and Column Management**
   - Add a Row:

     ```
     create_row <filename> <row_name>
     ```

   - Remove a Row:

     ```
     create_row <filename> <row_name>
     ```

   - Add a Column:

     ```
     create_column <filename> <column_name>
     ```

   - Remove a Column:

     ```
     delete_column <filename> <column_name>
     ```

3. **Reading and Writing Records**

- Use the `open_string_table_file`, `read_table` and `write_table` functions in your programs to manipulate table records programmatically.

Testing

1. **Creation and Deletion**

```
[devi@archlinux OSProjectTesting]$ create_table_file Grp6.tbl 12 5 40
Successfully created files Grp6.tbl & Grp6.tbl_index.txt
[devi@archlinux OSProjectTesting]$ delete_table_file Grp6.tbl
successfully deleted files Grp6.tbl & Grp6.tbl_index.txt
[devi@archlinux OSProjectTesting]$ create_table_file Grp6.tbl 12 5 40
Successfully created files Grp6.tbl & Grp6.tbl_index.txt
```

Creating a table file named Grp6.tbl with n_rows = 12, n_cols = 5, rec_size = 40. So now the Grp6.tbl file and its index file Grp6.tbl_index.txt have been created successfully.

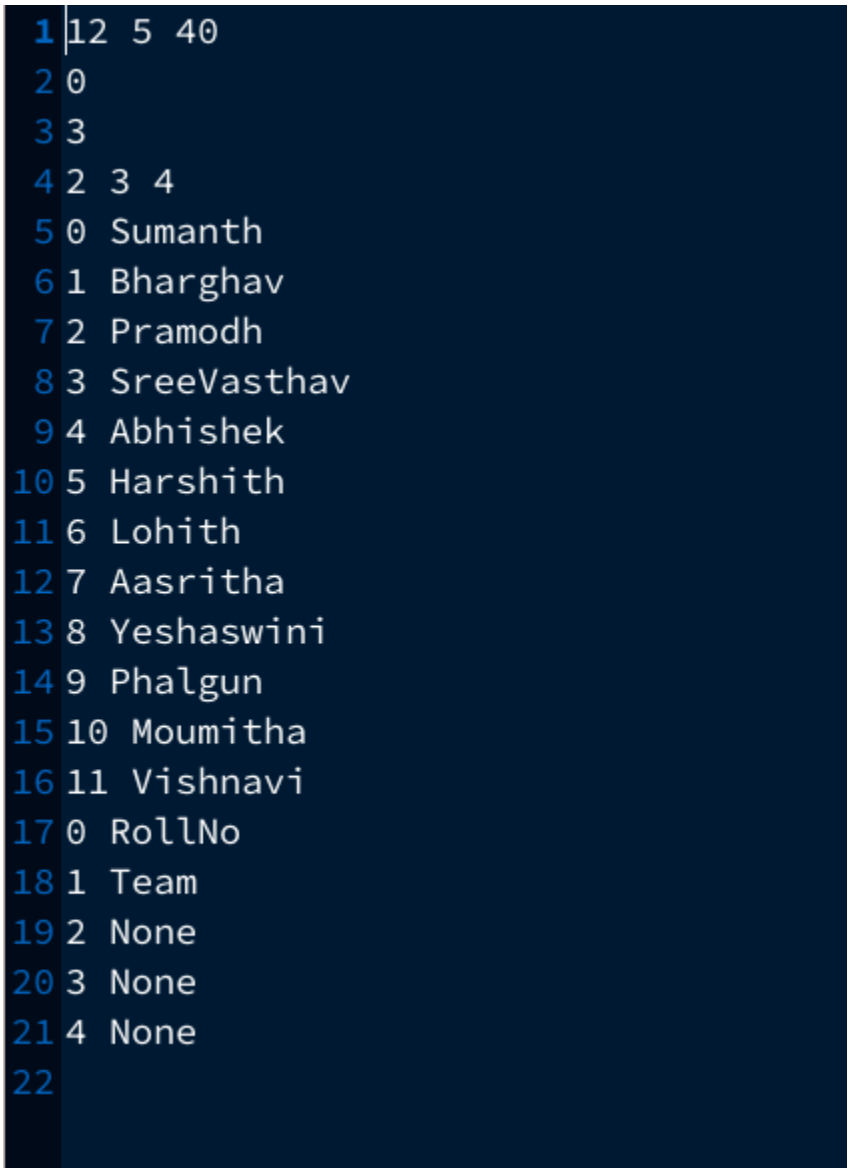
```
12 5 40
12
0 1 2 3 4 5 6 7 8 9 10 11
5
0 1 2 3 4
0 None
1 None
2 None
3 None
4 None
5 None
6 None
7 None
8 None
9 None
10 None
11 None
0 None
1 None
2 None
3 None
4 None

```

Just after creation, this 👆is Grp6.tbl_index.txt file.

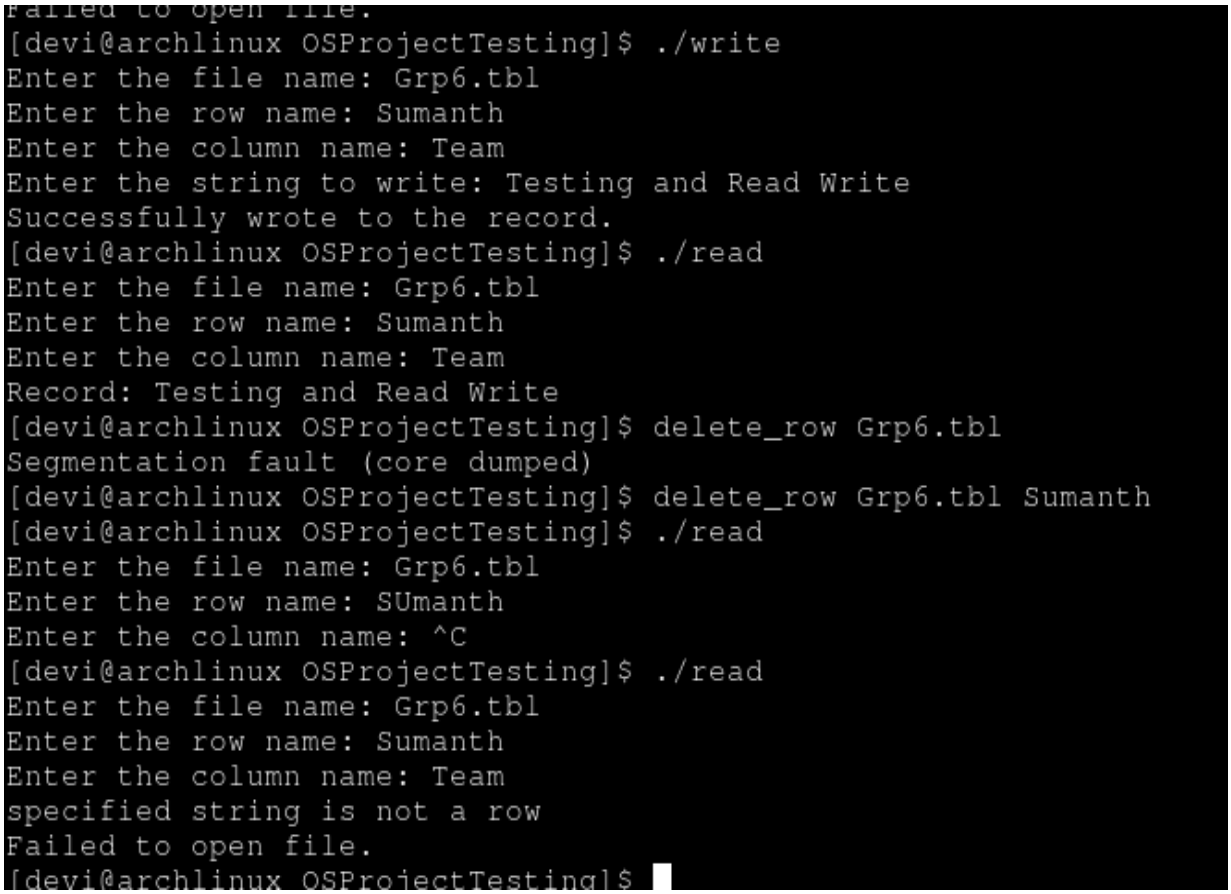
2. **Testing create_row, create_column, delete_row & delete_column**

```
[devi@archlinux OSProjectTesting]$ create_row Grp6.tbl Sumanth
[devi@archlinux OSProjectTesting]$ create_row Grp6.tbl Bharghav
[devi@archlinux OSProjectTesting]$ create_row Grp6.tbl Pramodh
[devi@archlinux OSProjectTesting]$ create_row Grp6.tbl SreeVasthav
[devi@archlinux OSProjectTesting]$ create_row Grp6.tbl Abhishek
[devi@archlinux OSProjectTesting]$ create_row Grp6.tbl Harshith
[devi@archlinux OSProjectTesting]$ create_row Grp6.tbl LOhith
[devi@archlinux OSProjectTesting]$ de
deallocvt               delete_column           derb
debugedit               delete_row              desktop-file-edit
debugfs                 delete_table_file       desktop-file-install
declare                 delpart                 desktop-file-validate
decode_tm6000           depmod                  devlink
[devi@archlinux OSProjectTesting]$ delete_row Grp6.tbl LOhith
[devi@archlinux OSProjectTesting]$ delete_row Grp6.tbl LOhith
specified string is not a row
[devi@archlinux OSProjectTesting]$ create_row Grp6.tbl Lohith
[devi@archlinux OSProjectTesting]$ create_row Grp6.tbl Aasritha
[devi@archlinux OSProjectTesting]$ create_row Grp6.tbl Yeshaswini
[devi@archlinux OSProjectTesting]$ create_row Grp6.tbl Phalgun
[devi@archlinux OSProjectTesting]$ create_row Grp6.tbl Moumitha
[devi@archlinux OSProjectTesting]$ create_row Grp6.tbl Vishnavi
[devi@archlinux OSProjectTesting]$ create_column Grp6.tbl RollNo
[devi@archlinux OSProjectTesting]$ create_column Grp6.tbl Team
[devi@archlinux OSProjectTesting]$
```

Creating rows (i.e, naming the rows) Sumanth, Bharghav, Pramodh, SreeVasthav, Abhishek, Harshith, Lohith, Aasritha, Yeshaswini, Phalgun, Moumitha, Vishnavi in table file Grp6.tbl and creating columns (i.e, naming the columns) RollNo, Team in table file Grp6.tbl. So, Grp6.tbl_index.txt have been updated successfully.



Just after creating rows and columns, this 👆is Grp6.tbl_index.txt file.



Deleting the row Sumanth in the table file Grp6.tbl and checking for row Sumanth which gives the result as specified string is not a row.

Just after deleting row Sumanth in the table file, this 👆is Grp6.tbl_index.txt file.

3. Testing Read & Write

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <table_file.h>

int main() {
    char filename[100], row_name[100], column_name[100];
    printf("Enter the file name: ");
    scanf("%99s", filename);
    printf("Enter the row name: ");
    scanf("%99s", row_name);
    printf("Enter the column name: ");
    scanf("%99s", column_name);

    struct File_ID fid = open_string_table_file(filename, row_name, column_name);
    if (fid.fd == -1) {
        fprintf(stderr, "Failed to open file.\n");
        return 1;
    }

    void *record = read_table(fid);
    if (record) {
        printf("Record: %s\n", (char *)record);
        free(record);
    } else {
        fprintf(stderr, "Failed to read the record.\n");
    }

    close(fid.fd);
    return 0;
}
```

test_read.c

The code takes a filename, row name, and column name as input, opens the file using open_string_table_file(), and tries to read a record matching the specified row and column using read_table. If successful, it prints the **Record** and **closes the file**. Errors during file opening or record reading are handled with appropriate messages, and resources are cleaned up before exiting.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <table_file.h>

int main() {
    char filename[100], row_name[100], column_name[100], input[100];
    printf("Enter the file name: ");
    scanf("%99s", filename);
    printf("Enter the row name: ");
    scanf("%99s", row_name);
    printf("Enter the column name: ");
    scanf("%99s", column_name);

    printf("Enter the string to write: ");
    scanf(" %99[^\n]", input);

    struct File_ID fid = open_string_table_file(filename, row_name, column_name);
    if (fid.fd == -1) {
        fprintf(stderr, "Failed to open file.\n");
        return 1;
    }

    size_t size = strlen(input) + 1; // Include null terminator
    if (write_table(fid, input, size) == -1) {
        fprintf(stderr, "Failed to write to the record.\n");
    } else {
        printf("Successfully wrote to the record.\n");
    }

    close(fid.fd);
    return 0;
}
```

test_write.c

The code takes a filename, row name, column name, and a string to write to a file. It opens the file using `open_string_table_file`, writes the string to the specified location using `write_table`, and checks for any errors during the process. Finally, it closes the file after completing the write operation.

```
[devi@archlinux OSProjectTesting]$ gcc -o write test_write.c -ltable_file
[devi@archlinux OSProjectTesting]$ gcc -o read test_read.c -ltable_file
[devi@archlinux OSProjectTesting]$ ./write
Enter the file name: Grp6.tbl
Enter the row name: Sumanth
Enter the column name: RollNo
Enter the string to write: CS22B073
Successfully wrote to the record.
[devi@archlinux OSProjectTesting]$ ./read
Enter the file name: Grp6.tbl
Enter the row name: Sumanth
Enter the column name: RollNo
Record: CS22B073
```

“read and write”.png image

1. A record (`CS22B073`) is successfully written to the file (`Grp6.tbl`) under a valid row (`Sumanth`) and column (`RollNo`).
2. The `read` program retrieves the record correctly, showing that both read and write operations work with valid inputs and metadata.

```
[devi@archlinux OSProjectTesting]$ ./write
Enter the file name: NonExistingFile
Enter the row name: SOmeROw
Enter the column name: SOmeCOl
Enter the string to write: random
Unable to open table file NonExistingFile
Failed to open file.
[devi@archlinux OSProjectTesting]$ ./write
Enter the file name: Grp6.tbl
Enter the row name: NonExistingRow
Enter the column name: SomeCol
Enter the string to write: random
specified string is not a row
Failed to open file.
[devi@archlinux OSProjectTesting]$ ./write
Enter the file name: Grp6.tbl
Enter the row name: Sumanth
Enter the column name: NonExistingCol
Enter the string to write: randfom
specified string is not a column
Failed to open file.
```

“read and write error cases”.png image

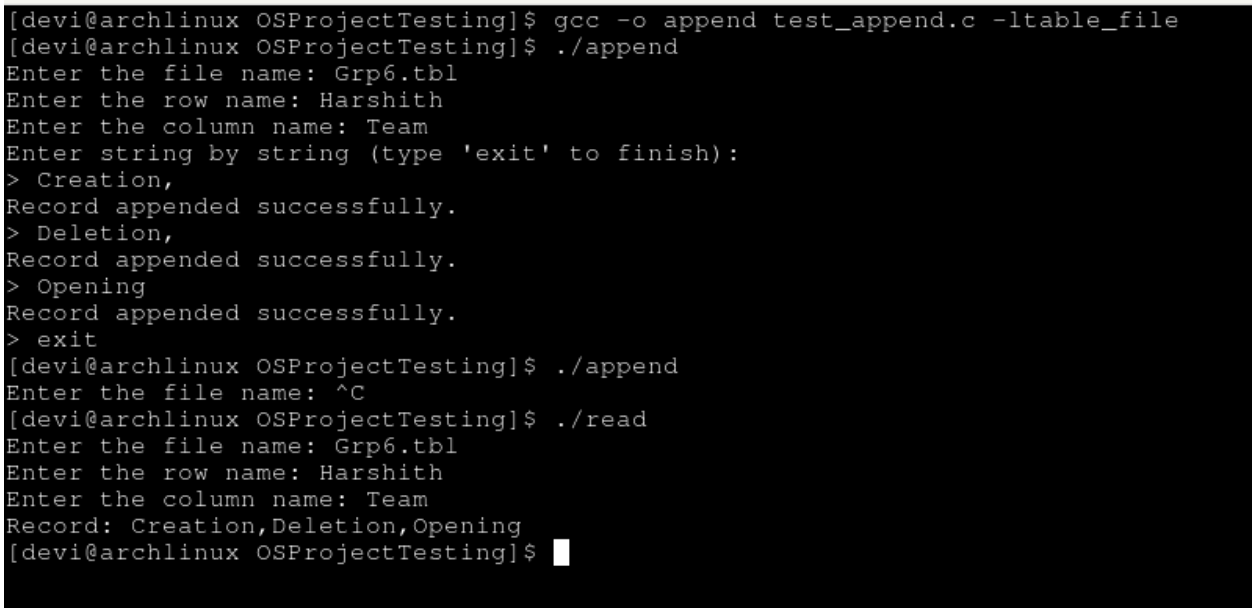
1. The program fails because the specified table file (`NonExistingFile` or `NonExistingRow`) does not exist.

2. Another failure occurs when invalid rows or columns (`NonExistingRow`, `NonExistingCol`) are provided, indicating the row or column does not match metadata in the table.

4. Append

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <table_file.h>
#include <unistd.h>

int main() {
    char filename[100], row_name[100], column_name[100];
    printf("Enter the file name: ");
    scanf("%99s", filename);
    printf("Enter the row name: ");
    scanf("%99s", row_name);
    printf("Enter the column name: ");
    scanf("%99s", column_name);
    struct File_ID fid = open_string_table_file(filename, row_name, column_name);
    if (fid.fd == -1) {
        fprintf(stderr, "Failed to open file.\n");
        return 1;
    }

    printf("Enter string by string (type 'exit' to finish):\n");
    char input[fid.rec_size];
    while (1) {
        printf("> ");
        scanf(" %s", input);

        if (strcmp(input, "exit") == 0) {
            input[0]='\0';
            write_table(fid, input, 1);
            break;
        }

        if (write_table(fid, input, strlen(input)) == -1) {         //not adding null terminator
            fprintf(stderr, "Error: Failed to write record.\n");
        } else {
            printf("Record appended successfully.\n");
        }
    }

    close(fid.fd);
    return 0;
}
```

The code takes a filename, a row name, and a column name as inputs to identify a specific table within a file. It opens the file using the `open_string_table_file` function and allows the user to append multiple strings to the specified row and column using the `write_table` function. The process continues until the user types `exit`. It ensures proper error handling during file operations and closes the file after all records are appended successfully.



Records such as `Creation`, `Deletion`, and `Opening` are successfully written to the file (`Grp6.tbl`) under the valid row (`Harshith`) and column (`Team`). The read program retrieves these records correctly, demonstrating that both the write and read operations function as expected with valid inputs and metadata.