Republic of the Philippines
BATANGAS STATE UNIVERSITY
The National Engineering University
BatStateU-Alangilan Campus

College of Informatics and Computing Sciences
Computer Science Department

FINAL PROJECT REPORT
CS 131 DATA STRUCTURES AND ALGORITHMS
Midterm Class A.Y. 2022-2023

***PROGRAMIZER*: To-do List Program using Linked-Lists and Queue**

Submitted by:
De Torres, Czynon John P.
Del Mundo, Ron Gabriel B.
Peñaflorida, Ace G.

Under Supervision of:
POUL ISAAC C. DE CHAVEZ

JULY 2023

TABLE OF CONTENT

Introduction

I. Overview

Programizer, a simple task organizer program designed to perform basic task-inputting, editing, viewing, and deleting by utilizing file handling methods. The program uses two major data structures in C++: linked-lists and queues. It also optimizes task management by utilizing these strong data structures, giving users a complete and efficient answer to their organizational demands.

The user can either enter as a guest to facilitate task viewing or through registration by creating an account to be stored in a database. Upon login, various task-creation options can be prompted. This includes task categorization depending on the ID, priority level, deadline(date & time), status, and name. The user can also edit a specific task information and update it in the database which runs in a text file. Furthermore, task deletion is also supported by the program through an ID system to be entered by the user.

Users can enjoy a smooth and effective work management procedure by combining linked lists and queues with Programizer. Users can easily alter, add, or remove tasks as needed thanks to the linked lists, which make it simple to manipulate individual activities. The queues, on the other hand, guarantee that tasks are handled in a prioritized and orderly manner, allowing users to concentrate on what matters most.

II. Background and Motivation

In this capstone project, a group of first-year Computer Science students from Batangas State University have developed a program driven by their motivation to provide individuals with a valuable tool for managing and prioritizing their tasks. The main objective of the program is to simplify the task-management process and assist users in making informed decisions regarding their daily activities.

By utilizing this application, users can efficiently create deadlines and organize their activities. The program boasts a user-friendly design that ensures ease of use, enabling users to effortlessly input, modify, and track their assignments. The programmers aspire for their program to cater to individuals from various fields, including students, professionals, business owners, and housewives. They envision their creation as a versatile and indispensable tool that aids users in balancing academic assignments, meeting project deadlines, planning family duties, and handling business responsibilities.

Through the development of a comprehensive and user-friendly program, the developers aim to empower individuals to take control of their schedules, eliminate inefficiencies, and maximize their time. Their ultimate goal is to contribute to the productivity and success of people in their day-to-day lives.

III. Objective

The project specifically aims to:

1. Create a console-based “to-do list” program that supports user registration and login for sustainability and task manipulation (creation, categorization, and deletion).
2. Implement various multi-modal C++ data structures in the program such as Arrays, Linked-Lists, Classes and Objects, and Queues.
3. Utilize file-handling techniques to store user information and tasks in a reusable manner.
4. Present the user’s inputted to-do list using console formatting.

IV. **Methodology** (specify needs requirements)

A. Algorithm

```
INITIALIZE current_user, current_pass, current_user_id
SET id_width to 6
SET todo_width to 8
SET date_width to 9
SET time_width to 9
SET status_width to 12
SET prio_width to 12
SET categ_width to 12

STRUCT user
      INITIALIZE id, name, pass, next

STRUCT todo
      INITIALIZE id, name, date_dl, time_dl, priority, status, category, next

SET userhead to NULL
SET todohead to NULL

FUNCTION enter()
      DISPLAY "Welcome to Programizer"
      DISPLAY "[0] Enter as a guest."
      DISPLAY "[1] Login as existing user."
      DISPLAY "[2] Register."
      DISPLAY "Choice:"
      GET option
      SWITCH option:
         CASE '0':
            CALL guestmode()
            BREAK
```

```
    CASE '1':
        CALL login()
        BREAK
    CASE '2':
        CALL regis()
        BREAK
    DEFAULT:
        DISPLAY "Entered option isn't listed. Please try again."
        CALL system("pause")
        CALL system("cls")
        CALL enter()
        BREAK

FUNCTION regis()
    CALL system(“cls”)
    SET loop to true
    INITIALIZE username_input, password_input
    DO
        PRINT "Register for a new account"
        PRINT "Enter username: "
        OBTAIN username_input
        IF CALL search_user(userhead, username_input) RETURNS true THEN
            PRINT "Username exists, please try again."
            CALL system(“pause”)
            CALL system(“cls”)
        ELSE
            PRINT "Enter password:"
            OBTAIN password_input
            CALL add_user(username_input, password_input)
            SET loop to false
            CALL initialize_td()
```

```
                CALL options()
            ENDIF
        WHILE loop is true


    FUNCTION login()
        SET loop = true
        INITIALIZE username_input, password input


        DO
            GET username_input
            CALL search_user(userhead, username_input)
            IF == false
                DISPLAY  "User isn't in database, please try again."


            ELSE
            GET password_input
                IF password_input != current _pass
                DISPLAY "Password is incorrect, please try again."
                BREAK


            SET loop = false
            CALL  initialize_td()
            CALL options()


        WHILE loop = true


    FUNCTION guestmode():
        SET current_user_id = -1
        CALL initialize_td()
        CALL options()
```

```
FUNCTION options()
    INITIALIZE choice
    CALL view_td()

    DISPLAY << "[1] Add a to-do." << endl
    DISPLAY << "[2] Complete a to-do." << endl
    DISPLAY << "[3] Edit a to-do." << endl
    DISPLAY << "[4] Delete a to-do." << endl
    DISPLAY << "[5] Exit." << endl
    DISPLAY << "Choice: ";

    GET choice

    SWITCH choice
    CASE '1'
        CALL add_td();
        CALL options();
        BREAK
    CASE '2'
        CALL complete_td();
        CALL options();
        BREAK
    CASE '3'
        CALLedit_td();
        CALL options();
        BREAK
    CASE '4'
        CALL  delete_td(todohead)
        CALL options();
        BREAK
    CASE '5'
```

```
            CALL  exit()
            CALL options();
            BREAK
        DEFAULT
            CALL options()
            BREAK

FUNCTION exit()
        SET i = 0
        SET struct todo temp = todohead
        SET filename = "database/lists/" + CONVERT current_user_id TO STRING + ".txt"

        IF current_user_id == -1:
           OPEN file(filename)
           WRITE "" TO file
           CLOSE file(filename)
        ELSE:
           WHILE temp != NULL:
              IF i == 0:
                 DECLARE ofstream user_file(filename)
                 WRITE i + "," +
                      temp->name + "," +
                      temp->date_dl + "," +
                      temp->time_dl + "," +
                      temp->priority + "," +
                      temp->status + "," +
                      temp->category TO user_file
                 CLOSE user_file
              ELSE:
                 DECLARE ofstream user_file(filename, ios::app)
                 WRITE i + "," +
```

The heading line "FUNCTION **exit()**" prints exit() in bold.

```
                temp->name + "," +
                temp->date_dl + "," +
                temp->time_dl + "," +
                temp->priority + "," +
                temp->status + "," +
                temp->category TO user_file
            CLOSE user_file
        ENDIF
        temp = temp->next
        INCREMENT i by 1

    CALL system("cls")
    DISPLAY "Thank you for using this program!"
    RETURN

FUNCTION **add_user(string username_input, string password_input)**
    SET new_node to new user
    IF userhead is NULL THEN
        SET new_node's id to 0
    ELSE
        SET new_node's id to userhead's id + 1
    ENDIF
    SET new_node's name to username_input
    SET new_node's pass to password_input
    SET new_node's next to userhead
    SET userhead to new_node
    SET current_user_id to userhead's id

    OPEN database/users.txt in append mode
    IF file is open
        PRINT id + "," + username_input + "," + password_input + endl
```

```
          ENDIF

          CLOSE file


FUNCTION insert_user(int id, string username, string password)

          SET new_node to new user

          SET new_node's id to id

          SET new_node's name to username

          SET new_node's pass to password

          SET new_node's next to userhead

          SET userhead to new_node


FUNCTION open_user_list()

          INITIALIZE user_file, line, deets, user, pass, id;

          OPEN user_file at database/users.txt

          IF user_file is open THEN

                    WHILE there are still lines in the file

                              INITIALIZE refline

                              SET i to 0

                              WHILE details are getting divided by delimiter ','

                                        IF i = 0 THEN

                                                  SET id to divided detail

                                        ELSE IF i = 1 THEN

                                                  SET user to divided detail

                                        ELSE IF i = 2 THEN

                                                  SET pass to divided detail

                                        ENDIF

                                        INCREMENT i by 1

                              ENDWHILE

                              CALL insert_user(int id, string user, string pass)

                    ENDWHILE

          ELSE
```

```
        CREATE new file in database folder named "users/txt"
    ENDIF
    CLOSE user_file

FUNCTION search_user(struct user* head, string input)
    IF head is NULL THEN
        RETURN false
    ENDIF
    IF head's name is equal to input THEN
        SET current_user_id to head's id
        SET current_user to head's name
        SET current_pass to head's pass
        RETURN true
    ENDIF
    RETURN search_user(head's next, input)

FUNCTION view_td()
    CALL fix_td_width()
    INITIALIZE temp as new todo
    SET temp as todohead
    IF temp is a null pointer
        PRINT "List is empty."
    ELSE
        PRINT "ID" "Name" "Deadline" "" "Priority" "Status" "Category" endl
    ENDIF
    WHILE temp is not a null pointer
        INITIALIZE status
        IF temp's status is false THEN
            SET status to "Not done"
        ELSE IF temp's status is true THEN
            SET status to "Done"
        ENDIF
        PRINT temp->id
        PRINT temp->name
        PRINT temp->date_dl
        PRINT temp->time_dl
        PRINT temp->priority
        PRINT status
        PRINT temp->category
        PRINT endl
        SET temp to temp->next
    ENDWHILE
    PRINT endl
```

FUNCTION **initialize_td()**

```
    INITIALIZE user_file, line, deets, name, date, time, categ, id, prio, status
    SET filename to "database/lists/" + current_user_id + ".txt"
    OPEN user_file(filename)
    IF user_file is open
        WHILE lines are being extracted from user_file
            INITIALIZE refline(line)
            SET i to 0
            WHILE deets are being separated by delimiter ','
                IF i is 0
                    SET id to deets
                ELSE IF i is 1
                    SET name to deets
                ELSE IF i is 2
                    SET date to deets
                ELSE IF i is 3
                    SET time to deets
                ELSE IF i is 4
```

```
                        SET prio to deets
                    ELSE IF i is 5
                        IF deets is "0"
                            SET status to false
                        ELSE IF deets is "1"
                            SET status to true
                        ENDIF
                    ELSE IF i is 6
                        SET categ to deets
                    ENDIF
                    INCREMENT i by 1
                CALL insert_todo(id, name, date, time, prio, status, categ)
            ENDWHILE
        ELSE
            CREATE user_file(filename)
        ENDIF
        CLOSE user_file

    FUNCTION fix_td_width()
        SET temp to todohead
        IF temp is a null pointer THEN return
        ENDIF
        WHILE temp is not a null pointer
            IF length of temp's name + 4 is greater than todo_width
                SET todo_width to length of temp's name + 4
            ENDIF
            IF length of temp's category +4 is greater than categ_width
                SET categ_width to length of temp's category + 4
            ENDIF
            SET temp to temp->next
        ENDWHILE
```

```
FUNCTION **complete_todo(struct todo *head, int id)**
    WHILE head is not NULL
        IF head's id is equal to id
            SET head's status to true
        ENDIF
        SET head to head->next
    ENDWHILE

FUNCTION **add_td()**
    INITIALIZE name
    CALL system("cls")
    PRINT "What are you planning on doing?"
    OBTAIN name
    CALL get_deadline() RETURNING input_date, input_time
    CALL priorityLevel() RETURNING taskPriority
    CALL categoriesList() RETURNING taskCategory
    INITIALIZE new_node as new todo
    SET filename as "database/lists/" + current_user_id + ".txt"
    IF todohead is NULL THEN
        SET new_node's id TO 0
    ELSE
        SET new_node's id to todohead's id + 1
    ENDIF
    SET new_node's name to name
    SET new_node's date_dl to input_date
    SET new_node's time_dl to input_time
    SET new_node's priority to taskPriority
    SET new_node's status to false
    SET new_node's category to taskCategory
    SET new_node's next to todohead
    SET todohead to new_node
    OPEN user_file(filename) in append mode
    PRINT todohead->id + "," + todohead->name + "," + todohead->date_dl + "," +
todohead->time_dl + "," + todohead->priority + "," + todohead->status + "," +
todohead->category + endl
    CLOSE user_file
```

FUNCTION **get_deadline()**

```
    SET date_loop to true
    INITIALIZE struct vals to have variables d and taskCategory
    INITIALIZE date, time

    DO
        PRINT "What is the date of the deadline (MM/DD/YY)?"
        OBTAIN date
        IF verify_date(date) returns false
            PRINT "Error reading date\n"
        ELSE
            PRINT "What is the time of the deadline (HH:MM 24H format)?"
            OBTAIN time
            IF verify_time(time) returns false
                PRINT "Error reading time\n"
            ELSE
                SET date_loop to false
            ENDIF
        ENDIF
    WHILE date_loop is true
    RETURN vals{date,time}
```

FUNCTION **verify_date(string wholedate)**

```
    INITIALIZE month, date, year
```

```
SET thirty[] to {4, 6, 9, 11}
SET thirtyone[] to {1, 3, 5, 7, 8, 10, 12}
INITIALIZE refline(wholedate)
INITIALIZE deets

SET i to 0
WHILE refline is being divided by delimiter '/' to deets
    IF i is 0
        SET month to deets
    ELSE IF i is 1
        SET date to deets
    ELSE IF i is 2
        SET year to deets
    ENDIF
    INCREMENT i by 1
ENDWHILE

IF month is less than 1 or greater than 12
    RETURN false
ENDIF
IF month has thirty days
    IF date is less than 1 or greater than 30
        RETURN false
    ENDIF
ENDIF
IF month is February and is in a leap year
    IF date is less than 1 or greater than 29
        RETURN false
    ENDIF
ELSE IF month is just February
    IF date is less than 1 or greater than 28
```

```
            RETURN false
        ENDIF
    ENDIF
    IF month has thirty one days
        IF date is less than 1 or greater than 31
            RETURN false
        ENDIF
    ENDIF

    RETURN true

FUNCTION verify_time(string time)
    INITIALIZE hour, minutes, refline(time), deets
    SET i to 0
    WHILE refline is being divided by delimiter ':' to deets
        IF i is 0
            SET hour to deets
        ELSE IF i is 1
            SET minute to deets
        ENDIF
        INCREMENT i by 1
    ENDWHILE
    IF hour is less than 0 or hour is greater than 23
        RETURN false
    ENDIF
    IF minute is less than 0 or minute is greater than 59
        RETURN false
    ENDIF
    RETURN true

FUNCTION categoriesList()
```

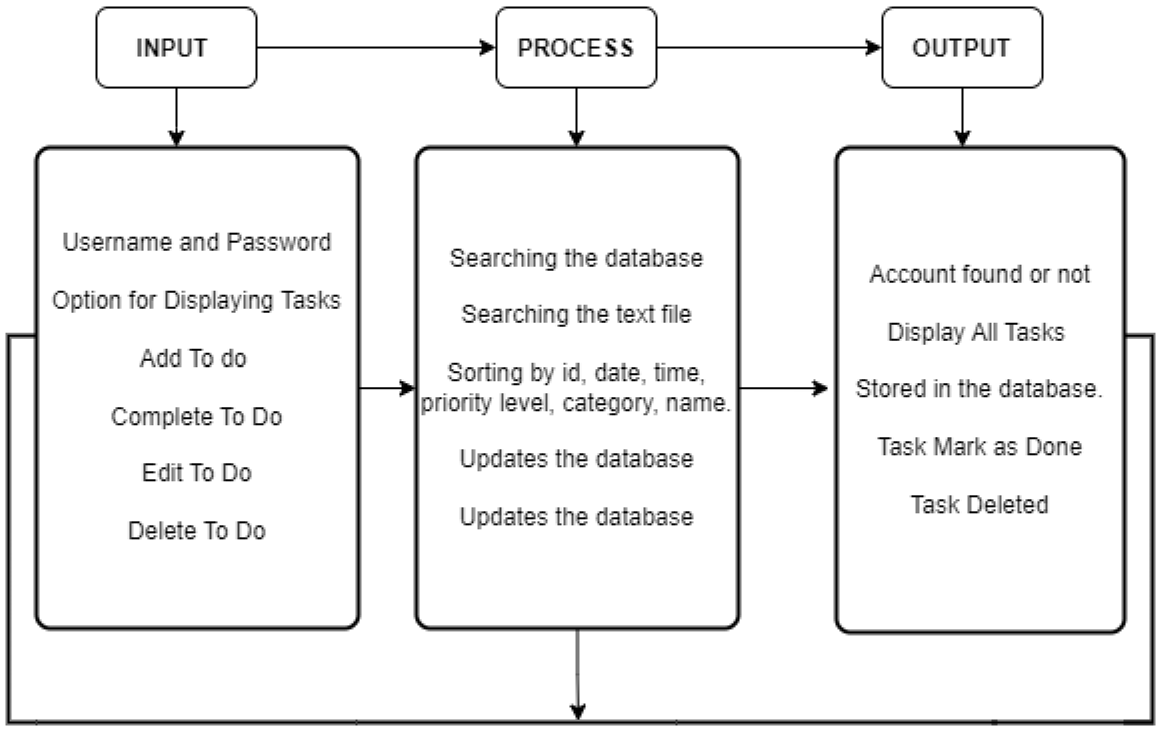
```
INITIALIZE chooseCateg, userCateg, newCateg, queue<string> categoriesList

SET loop_initializer to true

PUSH "Work" to categories

PUSH "Personal" to categories

PUSH "Fitness and Health" to categories

PUSH "Academics" to categories

PUSH "Music" to categories

DO

    PRINT "\n   1: Choose Task Category\n   2: Add New\n   Enter Here: "

    OBTAIN chooseCateg

    SWITCH chooseCateg

        CASE 1:

            CALL system("cls")

            PRINT "\nTask Categories:\n"

            CALL show_categories(categories)

            PRINT "\n\n   Enter Task Category: "

            OBTAIN userTaskCateg

                SET tempQueue to categories

                SET categoryFound to false

                WHILE tempQueue is not empty

                    SET existingCateg to front of tempQueue

                    POP tempQueue

                    IF existingCateg is equal to userTaskCateg

                        SET categoryFound to true

                        BREAK

                    ENDIF

                ENDWHILE

                IF categoryFound is equal to true

                    RETURN userTaskCateg

                ELSE

                    PRINT "Category Not Found!\n"
```

```
                        ENDIF
                    BREAK
                CASE 2:
                    PRINT "\nNew Category: "
                    OBTAIN newCateg
                    PUSH newCateg to categories
                    BREAK
            ENDSWITCH
        WHILE loop_initalizer is true

FUNCTION show_categories(queue<string> q)
        WHILE q is not empty
            PRINT front of q + endl
            POP queue
        ENDWHILE
        PRINT endl

FUNCTION priorityLevel()
        INITIALIZE priorityTask
        SET task_loop to true
        DO
            PRINT "\n\tChoose Priority Level: \n"
      PRINT "\t  1: Critical/Important\n"
      PRINT "\t  2: Not Priority Today\n"
      PRINT "\t  3: Low Importance\n"
      PRINT "\n\t   Enter Here: "
            OBTAIN priorityTask

            IF priorityTask is not 1, 2, or 3
                CALL system("cls")
                PRINT "Invalid Input"
            ELSE
                SET task_loop to false
                RETURN (priorityTask == 1 ? 1 : (priorityTask == 2 ? 2 : 3))
            ENDIF
        WHILE task_loop is true

FUNCTION insert_todo(int n, string name, string date, string time, int prio, bool status, string categ)
        INITIALIZE new_node as new todo
        SET new_node's id to n
        SET new_node's name to name
        SET new_node's date_dl to date
        SET new_node's time_dl to time
        SET new_node's priority to prio
        SET new_node's status to status
        SET new_node's category to categ
        SET new_node's next to todohead
        SET todohead to new_node

FUNCTION complete_td()
        CALL view_td()
        SET loop = true
        INITIALIZE id

        DISPLAY "What has been done? "
        GET id
        CALL complete_todo(todohead,id)

FUNCTION edit_td()
        CALL system("cls")
        CALL view_td()
```

```
SET loop to true
INITIALIZE id, property, new_prio, new_name, new_date, new_time, new_categ
INITIALIZE temp as new todo
SET temp to todohead

DO
    PRINT "Which item do you want to edit?"
    OBTAIN id
    PRINT "What do you want to edit?" + endl
    PRINT "[1] Name" + endl
    PRINT "[2] Date of deadline" + endl
    PRINT "[3] Time of deadline" + endl
    PRINT "[4] Priority" + endl
    PRINT "[5] Category" + endl
    OBTAIN property
    SWITCH property OF
        CASE 1:
            PRINT "What should be the new name?"
            OBTAIN new_name
            WHILE temp is not a null pointer
                IF temp's id is equal to id
                    SET temp's name to new_name
                ENDIF
                SET temp to temp->next
            ENDWHILE
            SET loop to false
            BREAK
        CASE 2:
            PRINT "What should be the new date of deadline?"
            OBTAIN new_date
            IF verify_date(new_date)
                WHILE temp is not a null pointer
                    IF temp's id is equal to id
                        SET temp's date_dl to new_date
                    ENDIF
                    SET temp to temp->next
                ENDWHILE
            ENDIF
            SET loop to false
            BREAK
        CASE 3:
            PRINT "What should be the new time of deadline?"
            OBTAIN new_time
            IF verify_time(new_time)
                WHILE temp is not a null pointer
                    IF temp's id is equal to id
                        SET temp's time_dl to new_time
                    ENDIF
                    SET temp to temp->next
                ENDWHILE
            ENDIF
            SET loop to false
            BREAK
        CASE 4:
            CALL priorityLevel() RETURNING new_prio
            WHILE temp is not a null pointer
                IF temp's id is equal to id
                    SET temp's priority to new_prio
                ENDIF
                SET temp to temp's next
            ENDWHILE
```

```
                SET loop to false
                BREAK
            CASE 5:
                CALL categoriesList() RETURNING new_categ
                WHILE temp is not a null pointer
                    IF temp's id is equal to id
                        SET temp's category to new_categ
                    ENDIF
                    SET temp to temp's next
                ENDWHILE
                SET loop to false
                BREAK
            OTHERS:
                PRINT "Not in the choices"
        ENDSWITCH
    WHILE loop is true

FUNCTION delete_td(struct todo *head)
    CALL system("cls")
    SET *curr to head
    SET *previous to NULL
    INITIALIZE del_id
    CALL view_td()
    PRINT "What do you want to delete? "
    OBTAIN del_id
    IF !head
        RETURN
    ENDIF
    IF head's id is equal to del_id
        SET curr to head's next
        DELETE head
        SET head to curr
    ELSE
        SET curr to head
        WHILE curr is not NULL and curr's id is not del_id
            SET previous to curr
            SET curr to curr's next
        ENDWHILE
        SET previous's next to curr's next
        DELETE curr
    ENDIF
    CALL view_td()
    CALL view_td()
    CALL system("cls")

BEGIN
    CALL open_user_list()
    CALL enter()
    RETURN 0
END
```

B. **IPO Model**



Tools Descriptions

I. **Interface**

A. **Functions**

1. **void enter()** - serves as the main entry point for a program and menu system. It presents a menu with three options: entering as a guest, logging in as an existing user, or registering as a new user. Upon completion of the selected action, the function concludes by invoking the `options` function to be discussed further.
2. **void regis() -** responsible for handling the registration process in the program. It prompts the user to register for a new account and gathers their input for the username and password. Next, it checks if the username already exists by calling the `search_user` function. If it does, an error message is displayed. If the username is unique, the function prompts for a password. Afterward, it calls the `add_user` function, which adds the new user's information to the program's user database.
3. **void login()** - prompts the user to enter his or her username and password. The search_user function is invoked to check whether the provided credentials are existing in the database. If not, the program pauses, clears the console screen again and continues

to the next iteration of the loop. If the entered username is existing, the user is prompted to enter the password. The program will then check if it matches a predefined password stored in the ‘current_pass’ variable. If the passwords match, it’ll exit the loop and calls the initialize_td() and options() functions.

4. **void guest_mode()** - sets the ‘current_user_id’ variable to -1 indicating that no user is currently logged in. It then invokes the functions initialize_td() and options() to initialize the to-do list and allow them to access the available options.
5. **void options()** - invokes the view_td function and then prompts the user with a menu of options, including adding a to-do, completing a to-do, editing a to-do, deleting a to-do, and exiting. Option 1 calls the add_td function(), 2; complete_td(), 3; edit_td(), 4; delete_td(), and 5; exits the menu.
6. **void initialize_td()** - initializes the task information variables (line, details, name, date, time, category, id, and priority level). It then constructs a filename by concatenating a directory path and the current user’s ID (converted to a string). The loop initializes the ‘i’ variable for indexing and creates a *stringstream* from the line to extract and assign individual details separated by commas. When i = 0, the loop converts the detail into an int type and assigns it to the ‘id’ variable. Similarly, it assigns the details to variables *name*, *date*, *time*, *prio*, *status*, and *categ* based on the value of *i*. After extracting the details, it calls insert_td() function and closes the ‘user_file.’
7. **void add_td()** - asks the user to enter a task, and then sets the variables *date* and *time* in the get_deadline() function to retrieve the deadline, and manages task priority and category with priorityLevel() function. It will then allocate memory for a new todo struct and assigns it to the new_node pointer. It constructs a filename for the user's to-do list, assigning inputted names, deadline dates, deadline times, priority, status, and category to the new_node struct. Upon updating the *todohead* pointer, it opens the user's to-do list file in append mode, writes the new to-do item details, and closes the file.

8. **void view_td()** - calls the fix_td_width() function for columns formatting and then uses setw() and left manipulators to format the columns and then displays a header row with column labels for the ID, Name, Deadline, Time, Priority, Status, and Category of the to-do items. The while loop verifies the status text based on the current temp node's

status member and displays the content of each to-do item formatted with the setw() and left manipulators until the temp pointer becomes nullptr.

9. **void complete_todo(struct todo* head, int id)** - takes a head pointer and an integer ID to access a linked list of to-do items. It then employs a while loop to iterate through each node and determines whether the ID matches the provided id. If it does, the loop changes the status variable to true, indicating that the to-do item has been completed.
10. **void complete_td()** - invokes the view_td function and then asks the user the *id* of the task. It then calls the complete_todo() function with arguments set to *todohead* and *id*.
11. **void edit_td() -** The user is required to input the ID of the item to be edited as well as the property to be changed (name, deadline date, deadline time, priority, or category). Other attributes, such as date, time, priority, and category, are treated similarly. The loop will continue until the user finishes editing and sets the loop to false. Following each adjustment, the code iterates through the to-do list, looking for the item with the matching ID and updating the corresponding property. At the end of each iteration, the loop condition is evaluated to determine whether to continue editing or quit the loop.
12. **void delete_td() -** user is asked to input the ID of the item to be deleted. It loops through the linked list to find the node with the same ID and deletes it.
13. **auto get_deadline()** - enters a loop and prompts the user to enter the task's deadline date and time. After the loop, it creates a vals struct, initializes it with the date and time values, and returns it.
14. **void fix_td_width() -** goes through each name and category in the linked list which are the lengths that change, and changes the max width depending on the longest length of the given string. If the length of name/category is longer than the set default length, the values of the given variables for the length of name/category are then set as that length + 4.
15. **bool verify_time(string time) -** separates a string using getline and the delimiter being a semi-colon. The part before the semi-colon would be the hour, and the one after would be the minute. After that, if hour is less than 0 or greater than 23, it returns false. If minute is less than 0 or greater than 59, it returns false. If it was able to pass those two, it would return true.

16. **bool verify_date(string wholedate) -** separates a string using getline and the delimiter being a forward slash. First part would be the month, second is the date, third is the year. If the month is less than 1 or greater than 12, it would return false. Next is, if month is in the list of months with 30 days, it would check if date is less than 1 or greater than 30, and return false. Next, if month is 2 (February) and it's a leap year, it would check if the date is less than 1 or greater than 29, and then return false. But if month is just 2 and it was any other year, it would check if the date is less than 1 or greater than 28, and then return false. Lastly, if the month is in the list of months with 31 days, it would check if the date is less than 1 or greater than 31, and return false. If it passed all of that and didn't return false, it would return true.
17. **int priorityLevel()** - asks the user to select the level of priority of the task to be inputted. 1 for high importance; 2 for moderately importance, and 3 for low importance.
18. **string categoriesList() -** uses queues for the selection of the task category. The user can either choose an existing task category or add a new one. If option 1 is selected, it invokes the show_categories() function to display the default task categories. The user's input will then be searched through the existing queue. If found, the function will return the value of the *userTaskCateg* variable, otherwise, the loop will continue. On the other hand, if option 2 is prompted, the inputted task will be pushed to the end of the categories queue.
19. **void show_categories(queue<string> q) -** takes one queue parameter and loops through it until the queue is empty, retrieves the element at the front of the queue using the front function, and then removes it using the pop function.
20. **bool search_user(struct user* head, string name) -** To show that the user could not be located, it first determines whether the current head, which represents the end of the list, is NULL. If it is, it returns false. It sets the current_user_id, current_user, and current_pass variables with the corresponding values of the current node and returns true to signify that the user was found if the name of the current head matches the supplied name. The method recursively calls itself with the next node in the linked list (head->next) if the names do not match. Until a match is found or the end of the list is reached, this recursive procedure keeps going. Finally, if the user is located, the function returns true; otherwise, it returns false.

21. **void open_user_list() -** access the file "users.txt" in the directory "database". If the file is present, its contents are read line by line, and each line's contents are divided into distinct values (ID, username, and password), which are then passed to the insert_user() method. A blank "users.txt" file is created if the file doesn't already exist.
22. **void insert_user(int n, string user_input, string pass_input) -** the newly constructed node are given the values of n, user_input and pass_input. The current userhead is entered into the new node's next field, and userhead is then modified to refer to the new node. This function essentially produces a new node with the supplied values and inserts it at the start of a linked list, where userhead links to the list's first node.
23. **void add_user(string user_input, string pass_input) -** The function determines if the userhead, or the linked list's head, is NULL. If it is, the new node's id field is set to 0. If not, the id field is set to the userhead node's current id value + 1. The specified user_input and pass_input values are allocated to the new node's name and pass fields. The userhead is modified to point to the new node, and the next field of the new node is set to the current userhead. The new userhead's id value is entered into the current_user_id variable. The program additionally opens the "database/users.txt" file in append mode,

B. Libraries

1. *fstream* - it is implemented in the program as the main method of storing the tasks entered by the user.
2. *sstream* - provides classes that support iostreams operations on strings allowing string input and output. Used in the program to
3. *string* - used in the program to manipulate the strings as well as to convert int types to string types.
4. *queue* - mainly utilized in the program to categorize tasks as well as adding new categories.
5. *algorithm* - provides a range of containers, such as vectors, lists, and maps, as well as algorithms for searching, sorting, and manipulating data.

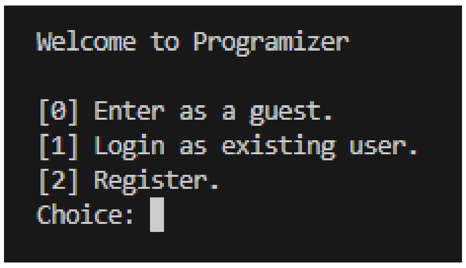
6. *stdio.h* - specifically used in reading the user input and then storing it to respective variables.
7. *iomanip* - provides functions in formatting the tasks once displayed in the terminal using setw() and manipulators.

II. Features

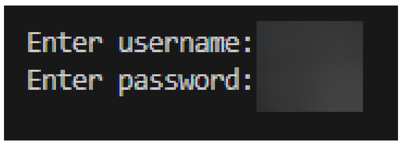
1. **User Registration and Log In.** The user can register to create an account, and the program will then require a username and password to be stored in a database text file. Upon registration, the user can now log in by entering his or her credentials as a registered user. The program will then search for the username and password in the database and check whether the information exists in the text file.
2. **Enter as a Guest.** The guest user can view the progress of the to-do list with indicated restrictions like not being able to add or edit any existing tasks.
3. **Task Creation.** The user can categorize the task depending on the date, time, priority level, status, category, and task name tol be stored in the task database.
4. **Task Editing.** The program supports task customization once added as an official task. The user can either change the task's deadline (date and time), priority level, status, category, or name.
5. **Task Completion.** Once a task has been completed, the user can mark it as done and its status will be changed to finish.
6. **Task Deletion.** A task can be deleted from the database once prompted by the user.
7. **Task Viewing.** All tasks can be retrieved in the text file that serves as the database and can be displayed using the *setw()* and left manipulators formatting.

III. Specifications

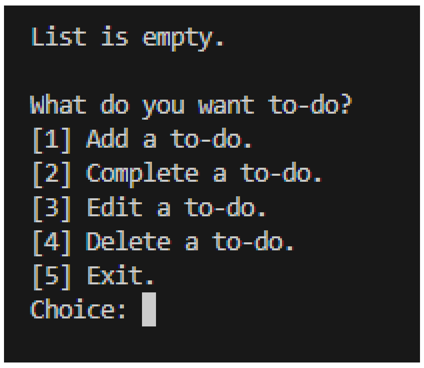
A. Task Creation



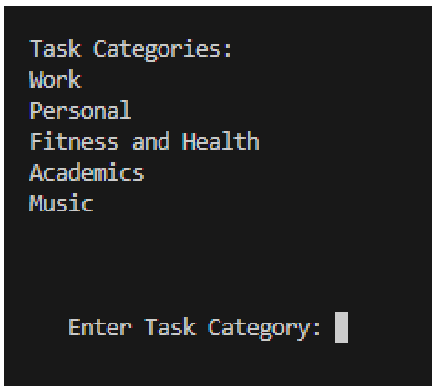
```
Welcome to Programizer

[0] Enter as a guest.
[1] Login as existing user.
[2] Register.
Choice:
```



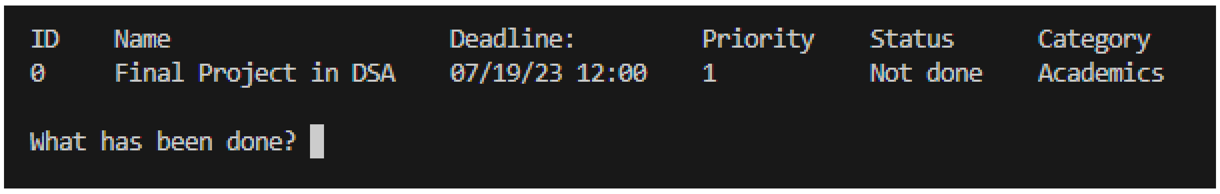
```
Enter username: [redacted]
Enter password:
```



```
List is empty.

What do you want to-do?
[1] Add a to-do.
[2] Complete a to-do.
[3] Edit a to-do.
[4] Delete a to-do.
[5] Exit.
Choice:
```

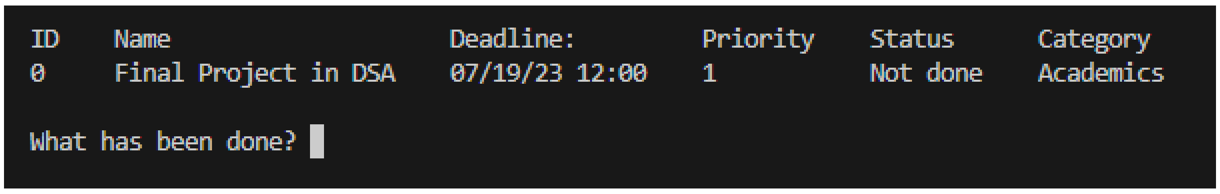


```
Task Categories:
Work
Personal
Fitness and Health
Academics
Music



   Enter Task Category:
```



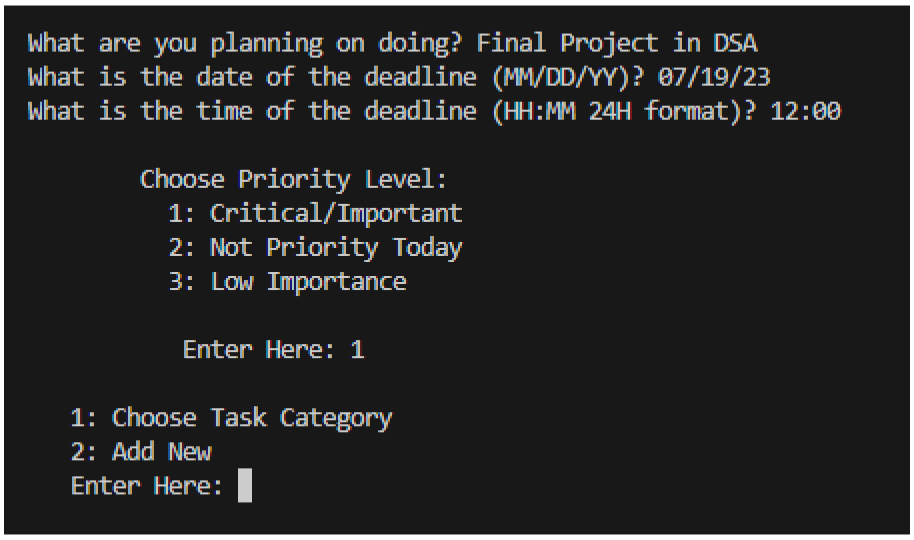
```
ID   Name                 Deadline:        Priority    Status      Category
0    Final Project in DSA 07/19/23 12:00   1           Not done    Academics

What has been done?
```

B. Task Completion

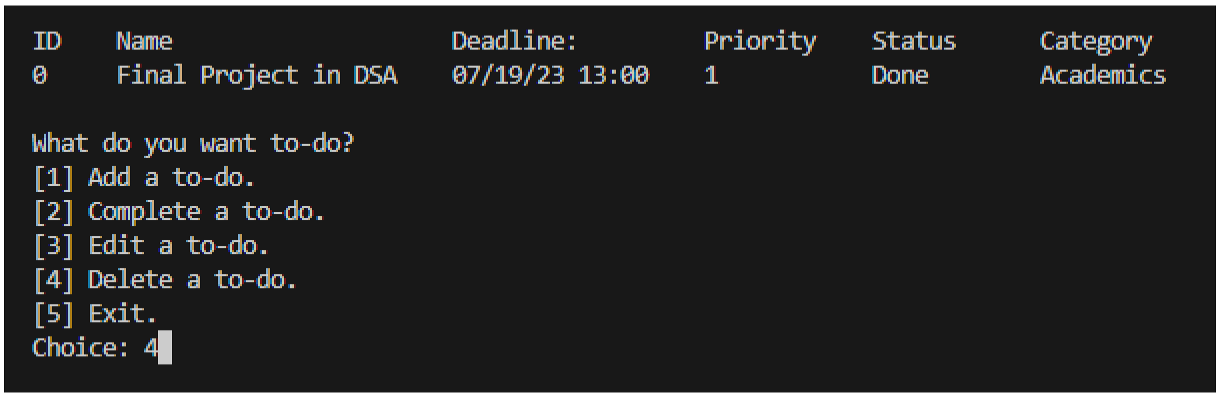


```
ID   Name                 Deadline:        Priority    Status      Category
0    Final Project in DSA 07/19/23 12:00   1           Not done    Academics

What has been done?
```

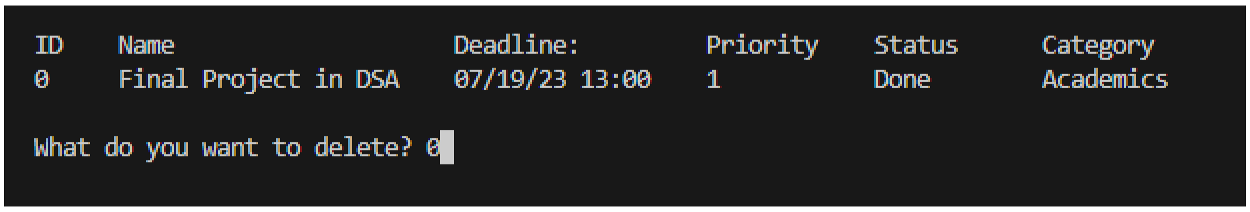
C. Task Deletion



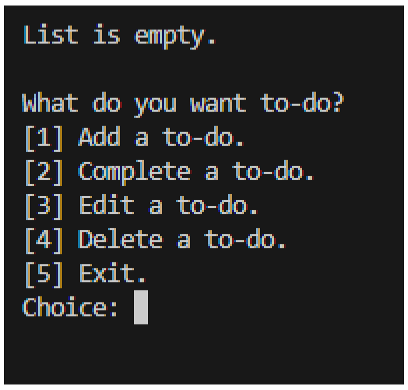
```
What are you planning on doing? Final Project in DSA
What is the date of the deadline (MM/DD/YY)? 07/19/23
What is the time of the deadline (HH:MM 24H format)? 12:00

        Choose Priority Level:
          1: Critical/Important
          2: Not Priority Today
          3: Low Importance

           Enter Here: 1

   1: Choose Task Category
   2: Add New
   Enter Here:
```



```
ID    Name                    Deadline:         Priority    Status      Category
0     Final Project in DSA    07/19/23 13:00    1           Done        Academics

What do you want to-do?
[1] Add a to-do.
[2] Complete a to-do.
[3] Edit a to-do.
[4] Delete a to-do.
[5] Exit.
Choice: 4
```



```
ID    Name                    Deadline:         Priority    Status      Category
0     Final Project in DSA    07/19/23 13:00    1           Done        Academics

What do you want to delete? 0
```



```
List is empty.

What do you want to-do?
[1] Add a to-do.
[2] Complete a to-do.
[3] Edit a to-do.
[4] Delete a to-do.
[5] Exit.
Choice:
```

IV. Analysis

Programizer utilized file-handling methods, linked lists, and queues to create organized database management by categorizing the to-do list tasks inputted by the user. Linked lists were used in adding new tasks through dynamic allocation to represent a single task with various properties like ID, name, deadline, priority, status, and category. These tasks' information will be appended and written in the database text file using a comma-separated format. In addition, task editing was implemented using the same data structure, in which the user can enter the task ID, and the program will then use a pointer to locate the specific task node to modify and update its value with the new data provided by the user. As for task deletion, traversal management was implemented to navigate through the linked list in order to locate and delete a specific task with the given ID and free the allocated memory for the deleted item. All of these functionalities were facilitated by the user when viewing the tasks. Also, linked-list were used in the user registration and login. Through the use of structs, the user's ID, name, and password were defined and added to the user database (through file appending), available for searching each user when registering or logging in to the program. Lastly, the queue data structure was used in adding new task categories to the existing or default categories provided by the developers. Overall, these data structures served as the major and most practical option in implementing the methods of the Programizer as they offer efficient memory management and fast insertion and deletion operations.

Future Work

Recommendations

- This project could use better and safer databases like MySQL, PostgreSQL or NoSQL. Just using text files to keep data is unsafe and not intuitive considering the amount of data that is being stored.

Appendix A: Source Code

```
#include <iostream>
#include <fstream> // file stream
#include <string>  // string
#include <sstream>
#include <stdio.h>   // scanf
#include <algorithm> // find
#include <queue>     // queue
```

```
#include <iomanip>   // setw

using namespace std;

string current_user, current_pass;
int current_user_id;

// widths
int id_width = 6,
    todo_width = 8,
    date_width = 9,
    time_width = 9,
    status_width = 12,
    prio_width = 12,
    categ_width = 12;

// nodes
struct user
{
    int id;
    string name;
    string pass;
    user *next;
};

struct todo
{
    int id;
    string name;
    string date_dl;
    string time_dl; // current available time method is a bit confusing
    int priority;

    // 1 - most important

    // 2 - mid

    // 3 - least important

    bool status; // done or not done

    string category;

    todo *next;

};


// node initialization

user *userhead = NULL;

todo *todohead = NULL;


// starter functions

void enter();

void regis();

void login();

void guestmode();

void options();

void exit();


// user node functions

void add_user(string user_input, string pass_input);

void insert_user(int n, string user_input, string pass_input);

void open_user_list();

bool search_user(struct user *head, string name);


// todo node functions

void view_td();

void initialize_td();

void fix_td_width();
void complete_todo(struct todo *head, int id);


// functions for todo
void add_td();
auto get_deadline();
bool verify_date(string wholedate);
bool verify_time(string time);
string categoriesList();
int priorityLevel();
void show_categories(queue<string> q);
void insert_todo(int n, string name, string date, string time, int prio, bool status, string categ);
void complete_td();
void edit_td();
void delete_td(struct todo *head);


int main()
{
    open_user_list();
    enter();
    return 0;
}


void add_user(string user_input, string pass_input)
{
    struct user *new_node = new user;


    if (userhead == NULL)
        new_node->id = 0;
    else
        new_node->id = userhead->id + 1;
    new_node->name = user_input;
    new_node->pass = pass_input;
    new_node->next = userhead;
    userhead = new_node;

    current_user_id = userhead->id;

    ofstream user_file("database/users.txt", ios::app);
    if (user_file.is_open())
        user_file << userhead->id << "," << user_input << "," << pass_input << endl;
    user_file.close();
}

void insert_user(int n, string user_input, string pass_input)
{
    struct user *new_node = new user;
    new_node->id = n;
    new_node->name = user_input;
    new_node->pass = pass_input;
    new_node->next = userhead;
    userhead = new_node;
}

void open_user_list()
{
    ifstream user_file;
    string line, deets, user, pass;
    int id;
    user_file.open("database/users.txt");
    if (user_file.is_open())
    {
        while (getline(user_file, line))
        {
            stringstream refline(line);
            int i = 0;
            while (getline(refline, deets, ','))
            {
                if (i == 0)
                    id = stoi(deets);
                else if (i == 1)
                    user = deets;
                else if (i == 2)
                    pass = deets;
                i++;
            }
            insert_user(id, user, pass);
        }
    }
    else
        ofstream user_file("database/users.txt");

    user_file.close();
}

bool search_user(struct user *head, string name)
{
    if (head == NULL)
        return false;
    if (head->name == name)
    {
        current_user_id = head->id;
        current_user = head->name;
        current_pass = head->pass;
        return true;
    }

    return search_user(head->next, name);
}

void insert_todo(int n, string name, string date, string time, int prio, bool status, string categ)
{
    struct todo *new_node = new todo;
    new_node->id = n;
    new_node->name = name;
    new_node->date_dl = date;
    new_node->time_dl = time;
    new_node->priority = prio;
    new_node->status = status;
    new_node->category = categ;
    new_node->next = todohead;
    todohead = new_node;
}

void fix_td_width()
{
    struct todo *temp = todohead;

    if (temp == nullptr)
        return;
    while (temp != nullptr)
    {
        if (temp->name.length() + 4 > todo_width)
            todo_width = temp->name.length() + 4;
        if (temp->category.length() + 4 > categ_width)
```

```
            categ_width = temp->category.length() + 4;
        temp = temp->next;
    }
}

void initialize_td()
{
    ifstream user_file;
    string line, deets, name, date, time, categ;
    int id, prio;
    bool status;
    string filename = "database/lists/" + to_string(current_user_id) + ".txt";
    user_file.open(filename);
    if (user_file.is_open())
    {
        while (getline(user_file, line))
        {
            stringstream refline(line);
            int i = 0;
            while (getline(refline, deets, ','))
            {
                if (i == 0)
                    id = stoi(deets);
                else if (i == 1)
                    name = deets;
                else if (i == 2)
                    date = deets;
                else if (i == 3)
                    time = deets;
                else if (i == 4)
                    prio = stoi(deets);
                else if (i == 5)
                {
                    if (deets == "0")
                        status = false;
                    if (deets == "1")
                        status = true;
                }
                else if (i == 6)
                    categ = deets;
                i++;
            }
            insert_todo(id, name, date, time, prio, status, categ);
        }
    }
    else
        ofstream user_file(filename);

    user_file.close();
}

void view_td()
{
    fix_td_width();

    todo *temp = new todo;
    temp = todohead;

    if (temp == nullptr)
    {
        cout << "List is empty." << endl;
    }
```

```
    else
        cout << left << setw(id_width) << "ID"
             << left << setw(todo_width) << "Name"
             << left << setw(date_width) << "Deadline:"
             << left << setw(time_width) << ""
             << left << setw(prio_width) << "Priority"
             << left << setw(status_width) << "Status"
             << left << setw(categ_width) << "Category" << endl;

    while (temp != nullptr)
    {
        string status;
        if (temp->status == false)
            status = "Not done";
        if (temp->status == true)
            status = "Done";
        cout << left << setw(id_width) << temp->id
             << left << setw(todo_width) << temp->name
             << left << setw(date_width) << temp->date_dl
             << left << setw(time_width) << temp->time_dl
             << left << setw(prio_width) << temp->priority
             << left << setw(status_width) << status
             << left << setw(categ_width) << temp->category << endl;
        temp = temp->next;
    }
    cout << endl;
}

void complete_todo(struct todo *head, int id)
{
    while (head != NULL)
    {
        if (head->id == id)
        {
            head->status = true;
        }
        head = head->next;
    }
}

void show_categories(queue<string> q)
{
    while (!q.empty())
    {
        cout << q.front() << endl;
        q.pop();
    }
    cout << endl;
}

string categoriesList()
{
    int chooseCateg;
    string userTaskCateg;
    bool loop_initialzer = true;
    string newCateg;
    queue<string> categories;
    categories.push("Work"); // defaults
    categories.push("Personal");
    categories.push("Fitness and Health");
    categories.push("Academics");
    categories.push("Music");
```

```
do
{
    cout << "\n   1: Choose Task Category\n   2: Add New\n   Enter Here: ";
    cin >> chooseCateg;

    switch (chooseCateg)
    {
    case 1:
        system("cls");
        cout << "\nTask Categories:\n";
        show_categories(categories);
        cout << "\n\n   Enter Task Category: ";
        getline(cin >> ws, userTaskCateg);
        {
            queue<string> tempQueue = categories; // Create a temporary queue for searching
            bool categoryFound = false;
            while (!tempQueue.empty())
            {
                string existingCateg = tempQueue.front();
                tempQueue.pop();
                if (existingCateg == userTaskCateg)
                {
                    categoryFound = true;
                    break;
                }
            }
            if (categoryFound)
            {
                return userTaskCateg;
            }
```

```
                else
                {
                    cout << "Category Not Found!\n";
                }
            }
            break;

        case 2:
            cout << "\nNew Category: ";
            cin >> newCateg;
            categories.push(newCateg);
            break;
        }
    } while (loop_initialzer);
}

int priorityLevel()
{
    int priorityTask;
    bool task_loop = true;
    do
    {
        cout << "\n\tChoose Priority Level: \n"
             << "\t  1: Critical/Important\n"
             << "\t  2: Not Priority Today\n"
             << "\t  3: Low Importance\n"
             << "\n\t   Enter Here: ";

        cin >> priorityTask;

        if (priorityTask != 1 && priorityTask != 2 && priorityTask != 3)
        {
            system("cls");
            cout << "Invalid Input";
        }
        else
        {
            task_loop = false;
            return (priorityTask == 1 ? 1 : (priorityTask == 2 ? 2 : 3));
        }

    } while (task_loop);
}

bool verify_date(string wholedate)
{
    int month, date, year;
    int thirty[] = {4, 6, 9, 11};
    int thirtyone[] = {1, 3, 5, 7, 8, 10, 12};
    stringstream refline(wholedate);
    string deets;

    int i = 0;
    while (getline(refline, deets, '/'))
    {
        if (i == 0)
            month = stoi(deets);
        else if (i == 1)
            date = stoi(deets);
        else if (i == 2)
            year = stoi(deets);
        i++;
    }

    if ((month < 1) || (month > 12))
        return false;
    if (find(thirty, thirty + 4, month) != thirty + 4) // if month has thirty days
        if ((date < 1) || (date > 30))
            return false;
    if (month == 2 && (year % 4 == 0)) // if month is feb and is in a leap year
        if ((date < 1) || (date > 29))
            return false;
        else if (month == 2) // if month is feb on any other year
            if ((date < 1) || (date > 28))
                return false;
    if (find(thirtyone, thirtyone + 4, month) != thirtyone + 4)
        if ((date < 1) || (date > 31))
            return false;
    return true;
}

bool verify_time(string time)
{
    int hour, minute;
    stringstream refline(time);
    string deets;

    int i = 0;
    while (getline(refline, deets, ':'))
    {
        if (i == 0)
            hour = stoi(deets);
        else if (i == 1)
```

```
            minute = stoi(deets);
        i++;
    }

    if ((hour < 0) || (hour > 23))
        return false;
    if ((minute < 0) || (minute > 59))
        return false;
    return true;
}

auto get_deadline()
{
    bool date_loop = true;
    struct vals
    {
        string d, t;
    };
    string date, time;

    do
    {
        cout << "What is the date of the deadline (MM/DD/YY)? ";
        cin >> date;
        if (!verify_date(date))
            cout << "Error reading date\n";
        else
        {
            cout << "What is the time of the deadline (HH:MM 24H format)? ";
            cin >> time;
            if (!verify_time(time))
```

```
                cout << "Error reading time\n";
            else
            {
                date_loop = false;
            }
        }

    } while (date_loop);
    return vals{date, time};
}

void add_td()
{
    string name;
    system("cls");
    cout << "What are you planning on doing? ";
    getline(cin >> ws, name);

    auto [input_date, input_time] = get_deadline();
    int taskPriority = priorityLevel();
    string taskCategory = categoriesList();

    struct todo *new_node = new todo;
    string filename = "database/lists/" + to_string(current_user_id) + ".txt";

    if (todohead == NULL)
        new_node->id = 0;
    else
        new_node->id = todohead->id + 1;
    new_node->name = name;
    new_node->date_dl = input_date;
    new_node->time_dl = input_time;
    new_node->priority = taskPriority;
    new_node->status = false;
    new_node->category = taskCategory;
    new_node->next = todohead;
    todohead = new_node;

    ofstream user_file(filename, ios::app);
    if (user_file.is_open())
        user_file << todohead->id << ","
                  << todohead->name << ","
                  << todohead->date_dl << ","
                  << todohead->time_dl << ","
                  << todohead->priority << ","
                  << todohead->status << ","
                  << todohead->category << endl;
    user_file.close();
}

void complete_td()
{
    system("cls");
    view_td();

    bool loop = true;
    int id;

    cout << "What has been done? ";
    cin >> id;
    complete_todo(todohead, id);
}
```

```
void edit_td()
{
    system("cls");
    view_td();

    bool loop = true;
    int id, property, new_prio;
    string new_name, new_date, new_time, new_categ;
    struct todo *temp = new todo;
    temp = todohead;
    do
    {
        cout << "Which item do you want to edit? ";
        cin >> id;
        cout << "What do you want to edit? " << endl
             << "[1] Name" << endl
             << "[2] Date of deadline" << endl
             << "[3] Time of deadline" << endl
             << "[4] Priority" << endl
             << "[5] Category" << endl;
        cin >> property;
        switch (property)
        {
        case 1:
            cout << "What should be the new name? ";
            getline(cin >> ws, new_name);
            while (temp != nullptr)
            {
                if (temp->id == id)
                    temp->name = new_name;
                temp = temp->next;
            }
            loop = false;
            break;
        case 2:
            cout << "What should be the new date of deadline? ";
            cin >> new_date;
            if (verify_date(new_date))
                while (temp != nullptr)
                {
                    if (temp->id == id)
                        temp->date_dl = new_date;

                    temp = temp->next;
                }
            loop = false;
            break;
        case 3:
            cout << "What should be the new time of deadline? ";
            cin >> new_time;
            if (verify_time(new_time))
                while (temp != nullptr)
                {
                    if (temp->id == id)
                        temp->time_dl = new_time;

                    temp = temp->next;
                }
            loop = false;
            break;
            case 4:
                new_prio = priorityLevel();
                while (temp != nullptr)
                {
                    if (temp->id == id)
                        temp->priority = new_prio;

                    temp = temp->next;
                }
                loop = false;
                break;
            case 5:
                new_categ = categoriesList();
                while (temp != nullptr)
                {
                    if (temp->id == id)
                        temp->category = new_categ;
                    temp = temp->next;
                }
                loop = false;
                break;
            default:
                cout << "Not in the choices. ";
            }
        } while (loop);
}

void delete_td(struct todo *head)
{
    system("cls");
    todo *curr = head, *previous = NULL;
        int del_id;

        view_td();

        cout << "What do you want to delete? ";
        cin >> del_id;

        if (!head)
            return;
        if (head->id == del_id)
        {
            curr = head->next;
            delete head;
            head = curr;
        }
        else
        {
            curr = head;
            while (curr != NULL && curr->id != del_id)
            {
                previous = curr;
                curr = curr->next;
            }
            previous->next = curr->next;
            delete curr;
        }
        view_td();
        view_td();
        system("cls");
}
```

```
void exit()
{
    int i = 0;
    struct todo *temp = todohead;
    string filename = "database/lists/" + to_string(current_user_id) + ".txt";
    if (current_user_id == -1)
    {
        ofstream user_file(filename);
        user_file << "";
        user_file.close();
    }
    else
        while (temp != NULL)
        {
            if (i == 0)
            {
                ofstream user_file(filename);
                user_file << i << ","
                          << temp->name << ","
                          << temp->date_dl << ","
                          << temp->time_dl << ","
                          << temp->priority << ","
                          << temp->status << ","
                          << temp->category << endl;
                user_file.close();
            }
            else
            {
                ofstream user_file(filename, ios::app);
                user_file << i << ","
                          << temp->name << ","
```

```
                        << temp->date_dl << ","
                        << temp->time_dl << ","
                        << temp->priority << ","
                        << temp->status << ","
                        << temp->category << endl;
                user_file.close();
            }
            temp = temp->next;
            i++;
        }
    system("cls");
    cout << "Thank you for using this program!";
    return;
}

void options()
{
    char choice;
    system("cls");
    view_td();

    cout << "What do you want to-do?" << endl
         << "[1] Add a to-do." << endl
         << "[2] Complete a to-do." << endl
         << "[3] Edit a to-do." << endl
         << "[4] Delete a to-do." << endl
         << "[5] Exit." << endl
         << "Choice: ";

    cin >> choice;
    switch (choice)
    {
    case '1':
        add_td();
        options();
        break;
    case '2':
        complete_td();
        options();
        break;
    case '3':
        edit_td();
        options();
        break;
    case '4':
        delete_td(todohead);
        options();
        break;
    case '5':
        exit();
        break;
    default:
        options();
        break;
    }
}

void regis()
{
    system("cls");
    bool loop = true;
    string username_input, password_input;

    do
    {
        cout << "Register for a new account." << endl
             << endl;

        cout << "Enter username: ";
        cin >> username_input;
        if (search_user(userhead, username_input) == true)
        {
            cout << "Username exists, please try again. ";
            system("pause");
            system("cls");
        }
        else
        {
            cout << "Enter password: ";
            cin >> password_input;
            add_user(username_input, password_input);

            loop = false;
            initialize_td();
            options();
        }

    } while (loop);
}

void login()
{
```

```
system("cls");

bool loop = true;

string username_input, password_input;


do

{

    cout << "Enter username: ";

    cin >> username_input;

    if (search_user(userhead, username_input) == false)

    {

        cout << "User isn't in database, please try again.";

        system("pause");

        system("cls");

    }

    else

    {

        cout << "Enter password: ";

        cin >> password_input;

        if (password_input != current_pass)

        {

            cout << "Password is incorrect, please try again.";

            system("pause");

            system("cls");

            break;

        }

        loop = false;


        initialize_td();

        options();

    }

} while (loop);
```

```
}

void guestmode()
{
    current_user_id = -1;

    initialize_td();
    options();
}

void enter()
{
    system("cls");
    char option;

    cout << "Welcome to Programizer" << endl
         << endl;
    cout << "[0] Enter as a guest." << endl
         << "[1] Login as existing user." << endl
         << "[2] Register." << endl
         << "Choice: ";
    cin >> option;

    switch (option)
    {
    case '0':
        guestmode();
        break;
    case '1':
        system("cls");
        login();
        break;
    case '2':
        system("cls");
        regis();
        break;
    default:
        cout << "Entered option isn't listed. Please try again. " << endl;
        system("pause");
        system("cls");
        enter();
        break;
    }
}
```

College of Informatics and Computing Sciences
RUBRICK

	VERY GOOD	GOOD	FAIR	POOR	
	4	3	2	1	POINTS
POINT DISTRIBUTION	**15**	**10**	**8**	**5**	
Project Content	Provide a clear purpose ideas and evidences that support the project concept	Somewhat clear purpose, ideas and evidence that support the project concept	Attempts to define purpose which adequately does not provide ideas and evidence that support the project concept	Does not clearly define the purpose, ideas and does not show the evidences that support the project concept	15
POINT DISTRIBUTION	**15**	**10**	**8**	**5**	
Project Knowledge/Ideas	Demonstrate full knowledge with explanation and elaboration	At ease to answer question without further explanation	Can answer some of the question with no further explanation	No answer to all questions	15
POINT DISTRIBUTION	**10**	**8**	**5**	**3**	
Project Presentation	(100%) Presentation is well organized and reflect a logical order Presentation contains no grammar errors and easy to understand	(25%) Some of the presentation does not reflect logical order Presentation has no serious grammar errors, complete and understandable	(50%) of the presentation does not reflect logical order Presentation may contain some grammar errors and hard to understand	Extremely(75%) of the presentation does not reflect logical order Presentation contains several grammar errors and hard to understand	10
POINT DISTRIBUTION	**40**	**30**	**10**	**5**	
Project Code and Requirements	All the requirement of the project are successfully done	Some of the requirement does not successfully done	Most of the requirement are not successfully done	All of the requirements are not successfully done	40
POINT DISTRIBUTION	**20**	**15**	**10**	**5**	
Project Technicality and Creativity	All software technical aspect on functionality, usability and user friendly are being met	Some of the software technical aspect on functionality, usability and user friendliness of the project are not being met.	Most of the software technical aspect on functionality, usability, accuracy and user friendliness of the project are not being met	All of software technical aspect on functionality, usability, accuracy and user friendliness of the project are not being me	20
TOTAL POINTS					100

GANTT CHART

TITLE	PROJECT DEVELOPMENT STAGES	VERIFICATION ACTIVITIES	TASK	MONTH							
				NOVEMBER				DECEMBER			
	Requirement Analysis	Team Discussion and specified project requirements	Project Concept	1	2	3	4	1	2	3	4
						🟧	🟧				
	Design, Coding and Testing	Code construction	Test Driven Development 50-80% code				🟧				
	Presentation	Presentation of project	100%					🟧			