Real-world application solutions to problems applied with ADTs

Assignment 1

Nathaly M. Ingol Leon

Foundations of Data Structures & Algorithm Design - CS 3358

Texas State University

Professor: Edwin Vargas-Garzon

Spring 2024

Introduction

Real-world problems and solutions:

Problem 1: Task Planner with Linked Lists

Let's imagine how stressed a student can be knowing that they have a lot of tasks, projects or homework to do in these months and realizing that there is a task planner so you can plan your routine to stay well organized, you will have great satisfaction. Instead of having notes disorganized or scattered all over your desk, it is a thousand times better than having a digital application to be used whenever you are. This application will make use of linked lists, where you will have the function of adding, deleting, prioritizing and showing all tasks.

We will see how to build the application, we will have a menu with the options for the functions already mentioned, after having added them, each of the tasks will be listed with “Name/Title”, “Class”, “Deadline”, “Priority” and “Completed”. All these objects are perfectly part of a linked list, ready to be manipulated according to the options that will be made in the menu. With this application you will not be able to forget any task after being added. To make it a more complex application to develop, reminders and notifications could be integrated to make it even easier for people to complete their tasks without any problem.

Flow diagram:

To not make a large diagram, a short one will be made that's going to contain the general idea of the development of the task planner.

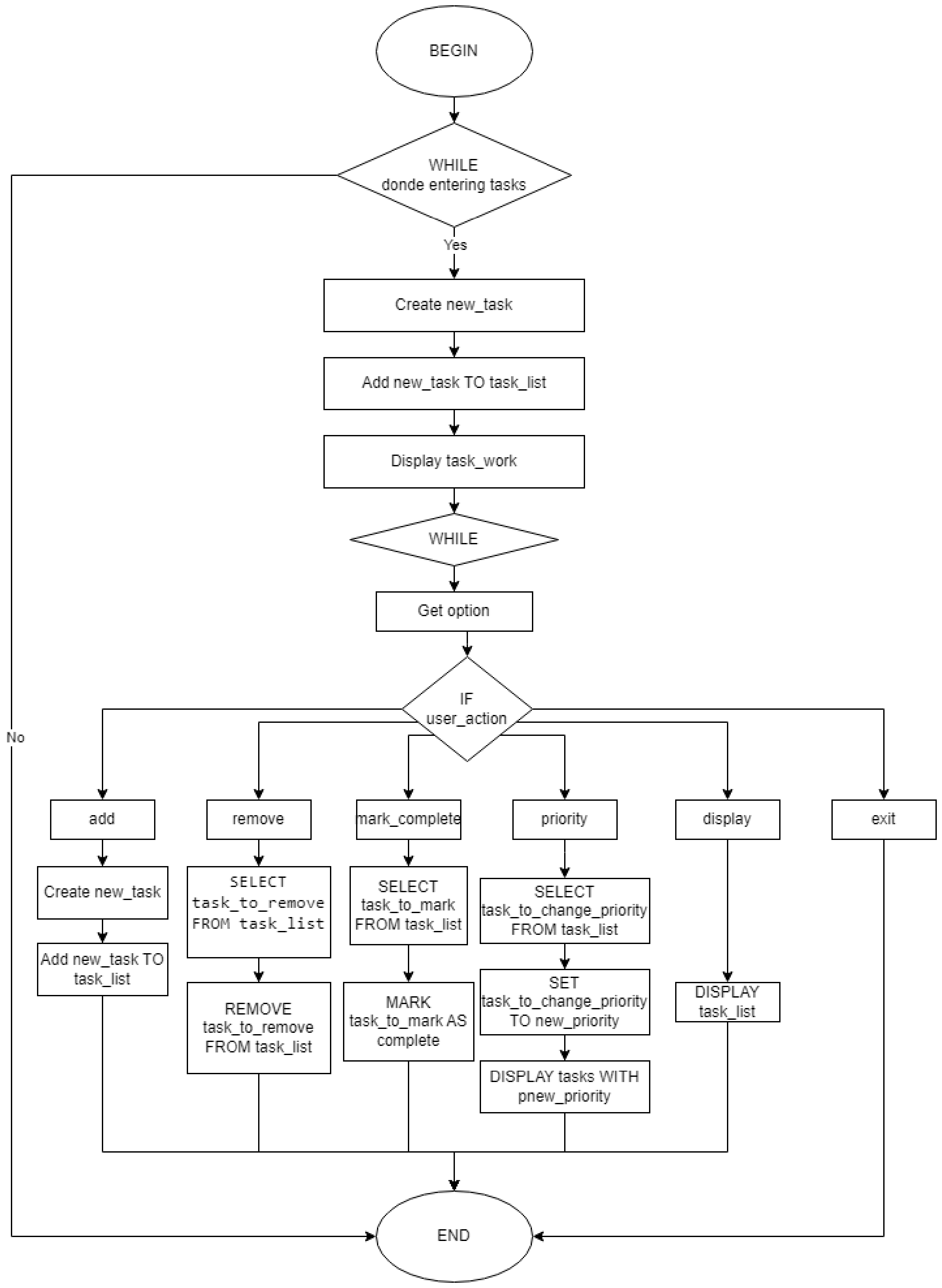

BEGIN
WHILE donde entering tasks
Yes
No
Create new_task
Add new_task TO task_list
Display task_work
WHILE
Get option
IF user_action
add
remove
mark_complete
priority
display
exit
Create new_task
Add new_task TO task_list
SELECT task_to_remove FROM task_list
REMOVE task_to_remove FROM task_list
SELECT task_to_mark FROM task_list
MARK task_to_mark AS complete
SELECT task_to_change_priority FROM task_list
SET task_to_change_priority TO new_priority
DISPLAY tasks WITH pnew_priority
DISPLAY task_list
END

Fig 1. *Problem 1 Flow diagram.*

Pseudocode:

```
BEGIN

WHILE NOT done_entering_tasks DO
    CREATE new_task
    ADD new_task TO task_list
END WHILE

DISPLAY task_list

WHILE NOT exit_program DO
    GET actions

    IF actions == "add" THEN
        CREATE new_task
        ADD new_task TO task_list

    ELSE IF actions == "remove" THEN
        SELECT task_to_remove FROM task_list
        REMOVE task_to_remove FROM task_list

    ELSE IF actions == "mark_complete" THEN
        SELECT task_to_mark FROM task_list
        MARK task_to_mark AS complete

    ELSE IF actions == "priority" THEN
        SELECT change_priority FROM task_list
        SET change_priority TO new_priority
        DISPLAY tasks WITH new_priority

    ELSE IF actions == "display_list" THEN
        DISPLAY task_list

    ELSE IF actions == "exit" THEN
        BREAK
    END IF
END WHILE

END
```

Problem 2: Undo/redo function with Stacks

At some point we have lost documents or wanted to return to a previous version of it, for this an application that performs the undo or redo function will be helpful. The application would be highly appreciated by users thanks to the ADT that will be implemented called stack. This data structure is known as LIFO (Last in, First out), which if one analyzes it well, would be greatly reflected in the undo or redo. It would fulfill its function when one wants to undo something, the last action enters first and then exits to undo it.

So, let's imagine that you are in the university library making a report. Every time something new is added it is inserted into a stack. Clicking undo will return you to the previous version. The same thing happens with the redo function, it helps you recover the action if you accidentally discarded what you recently had. Now for this application to have a better function and be more complex, the undo/redo history could be limited to avoid the stack never ending and thus obtain better performance of the app. It could also be optimized to be helpful in large projects, giving users peace of mind in case they lose something in an accident.

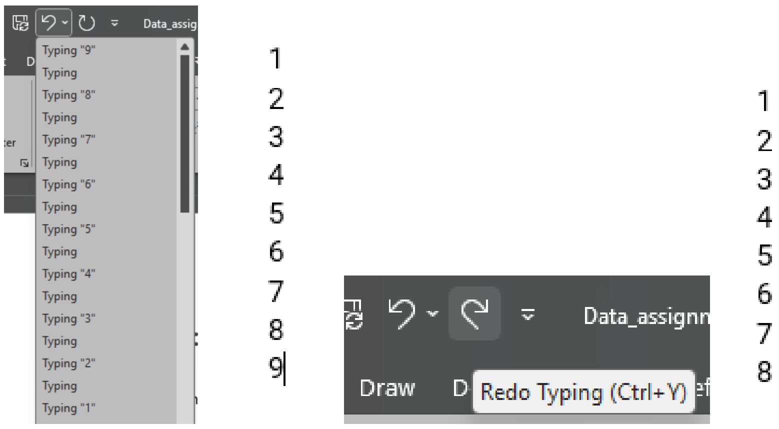


Fig 2. *Word's undo/redo function*.

Flow diagram:

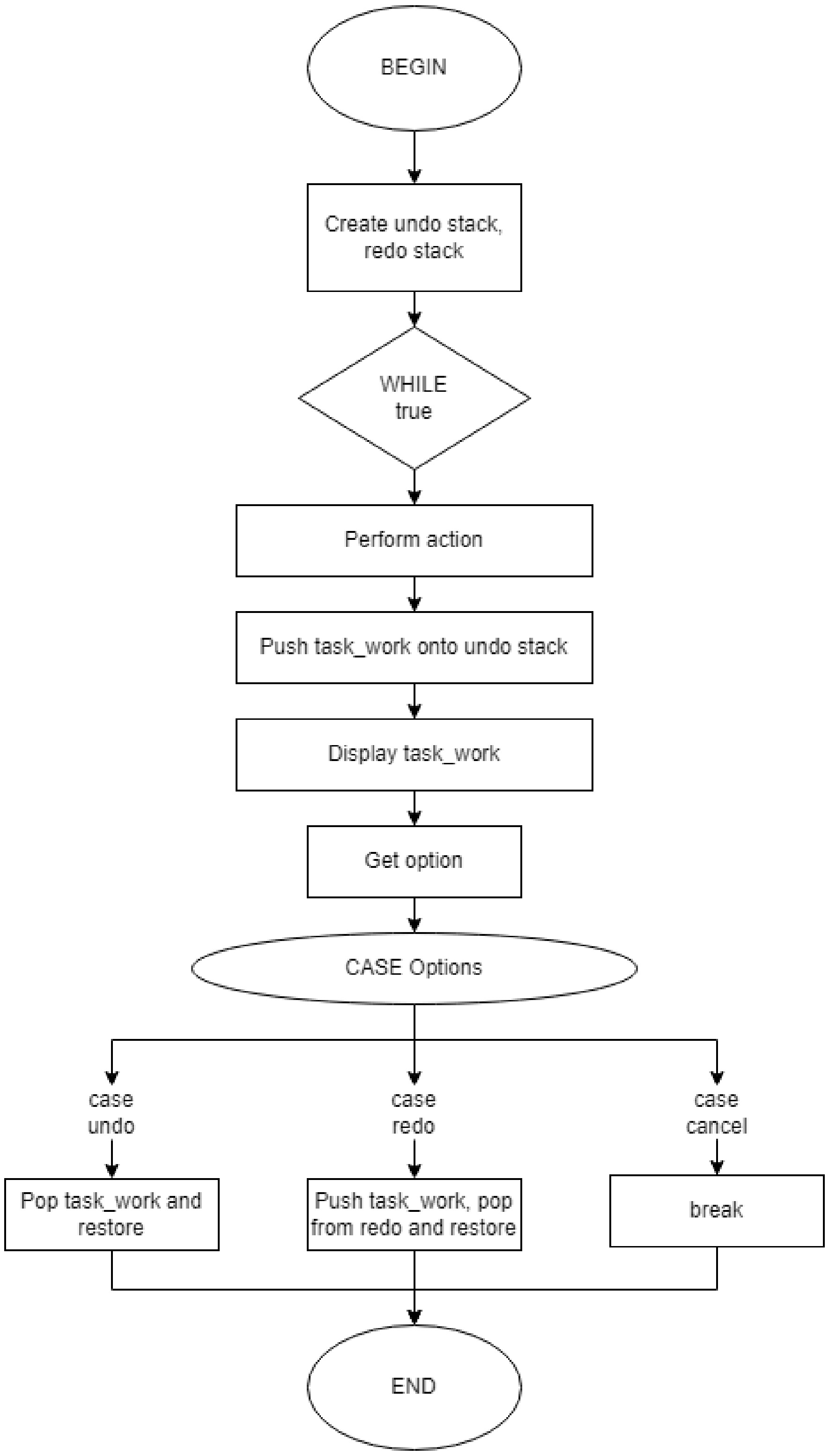

BEGIN
Create undo stack, redo stack
WHILE true
Perform action
Push task_work onto undo stack
Display task_work
Get option
CASE Options
case undo
case redo
case cancel
Pop task_work and restore
Push task_work, pop from redo and restore
break
END

Fig 3. *Problem 2 Flow diagram.*

Pseudocode:

```
BEGIN

  Create undo stack, redo stack

  WHILE true
    Perform action
    Push task_work onto undo stack
    Display current task_work

    Get option (undo, redo, cancel)

    CASE option
      WHEN undo: Pop task_work from undo, restore
      WHEN redo: Push task_work from undo onto redo, pop from redo, restore
      WHEN cancel: BREAK
    END CASE

  END WHILE

END
```

Problem 3: Chores Schedule with Queues

We all know the terror that most people have in their homes, household chores. This can stress us out a lot, we can put it on a schedule within an application that would make use of queues. We know this data structure as FIFO (First in, First out), which demonstrates the order in which chores must be performed for an organized coexistence. The digital queue that will be carried out will obtain the household chores, each one will obtain its respective information such as its “Name”, “Frequency”, “Due date”. As more chores are added, the new ones would go to the end of the line and wait for “Their turn”, so when they are ready they will follow the order of the oldest chores so that they are not forgotten. Just like in lists, to be able to prioritize this application it would also be helpful to add reminders, avoiding the accumulation of high priority chores. For example: dirty clothes, washing dishes, among others. In addition, with reminders you will not lose pace in completing the chores. Completing tasks on time will allow them to be more manageable and organized.

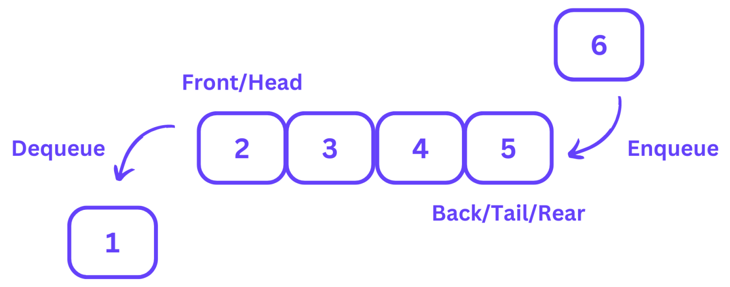


Fig 4. *Queues representation*. From Learn Loner.

Flow diagram:

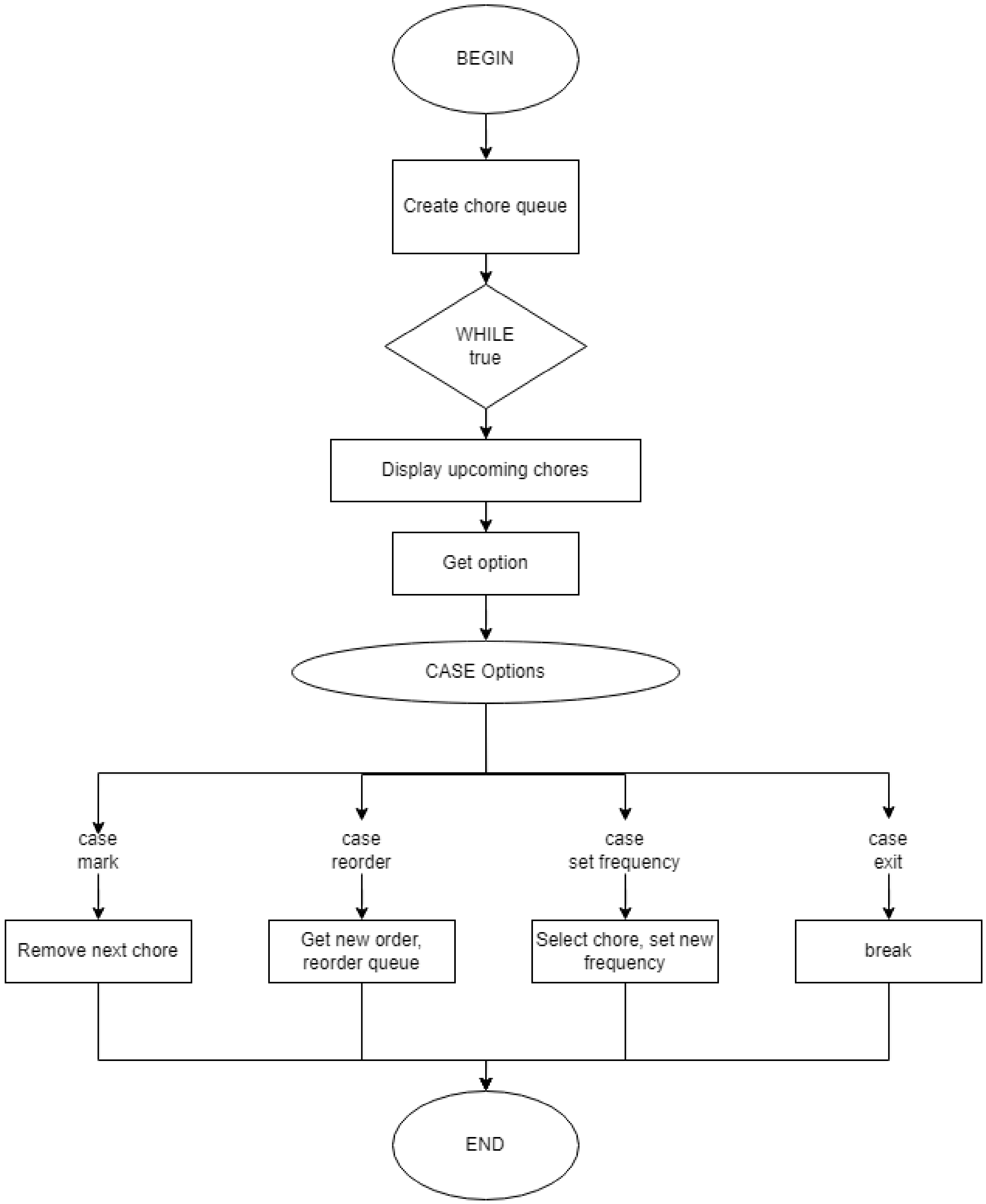


Fig 5. *Problem 2 Flow diagram*.

Pseudocode:

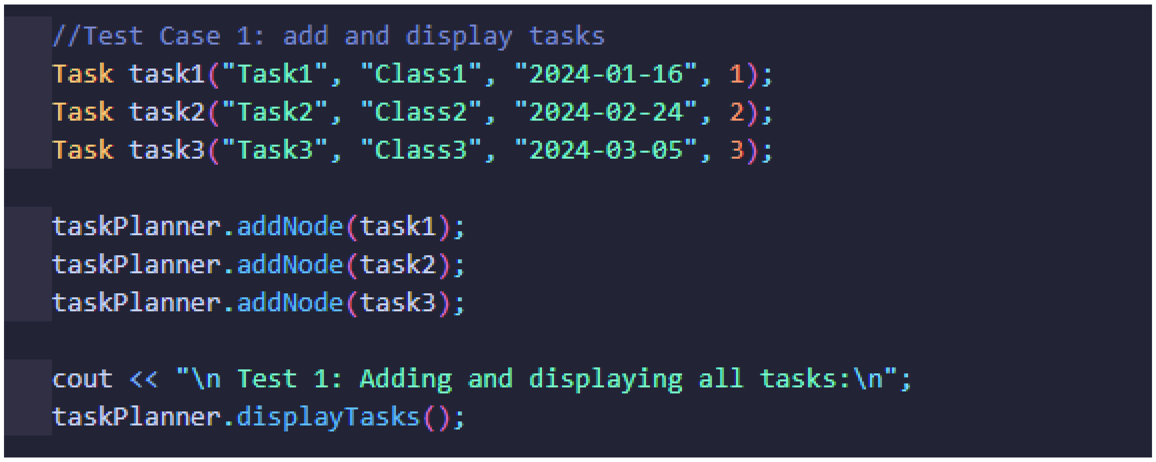
```
BEGIN
  Create chore queue

  WHILE true
    Display upcoming chores
    Get option (mark, reorder, set frequency, exit)

    CASE option
      WHEN mark: Remove next chore from queue
      WHEN reorder: Get new order, reorder queue
      WHEN set frequency: Select chore, set new frequency
      WHEN exit: BREAK
    END CASE

  END WHILE
END
```

Implementation

The Task planner application will be implemented in C++ and its files submitted separately. The test cases that will be used for this project will be listed below:

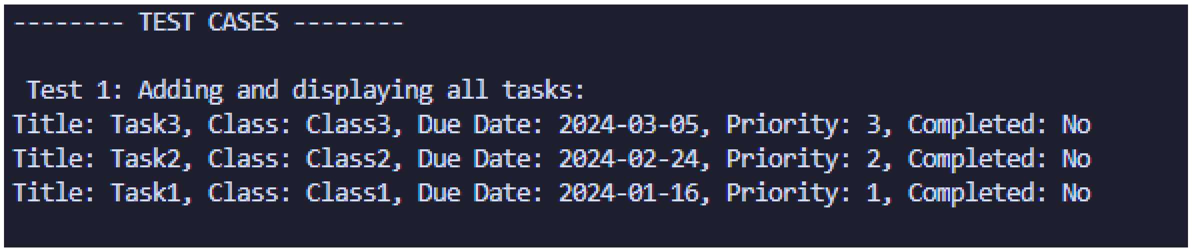
- **First test case: Adding and displaying tasks**

  We'll be adding various tasks with different attributes in the format implemented: Title, class, due date and priority. After adding, we call the function *"displayTasks()"* so we can verify if everything was added correctly.



```
//Test Case 1: add and display tasks
Task task1("Task1", "Class1", "2024-01-16", 1);
Task task2("Task2", "Class2", "2024-02-24", 2);
Task task3("Task3", "Class3", "2024-03-05", 3);

taskPlanner.addNode(task1);
taskPlanner.addNode(task2);
taskPlanner.addNode(task3);

cout << "\n Test 1: Adding and displaying all tasks:\n";
taskPlanner.displayTasks();
```

Fig 6. *Test case 1: Adding and displaying tasks.*

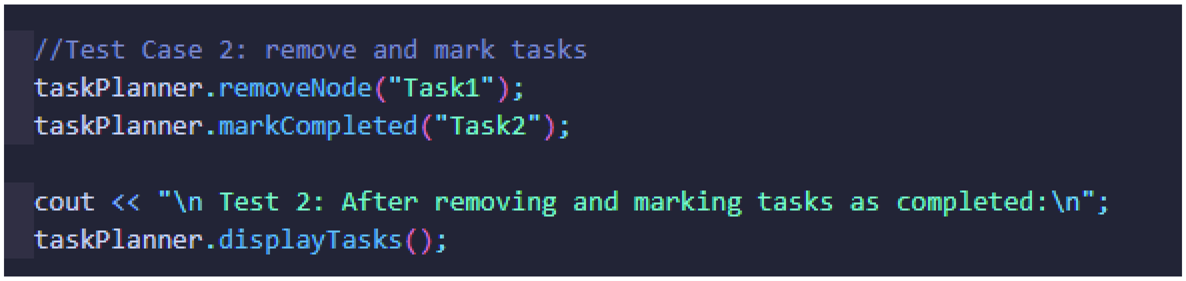


```
-------- TEST CASES --------

 Test 1: Adding and displaying all tasks:
Title: Task3, Class: Class3, Due Date: 2024-03-05, Priority: 3, Completed: No
Title: Task2, Class: Class2, Due Date: 2024-02-24, Priority: 2, Completed: No
Title: Task1, Class: Class1, Due Date: 2024-01-16, Priority: 1, Completed: No
```

Fig 7. *Test case 1 in terminal.*

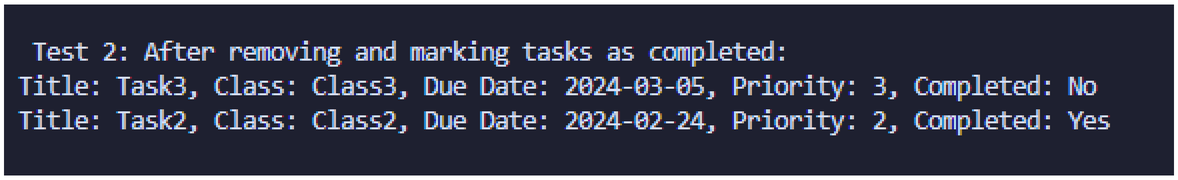
- **Second test case: Removing and marking tasks**

  Calling "removeNode()" to remove a specific tasks and "markCompleted" to mark as completed.



```
//Test Case 2: remove and mark tasks
taskPlanner.removeNode("Task1");
taskPlanner.markCompleted("Task2");

cout << "\n Test 2: After removing and marking tasks as completed:\n";
taskPlanner.displayTasks();
```

Fig 8. *Test case 2: Removing and marking tasks as completed.*

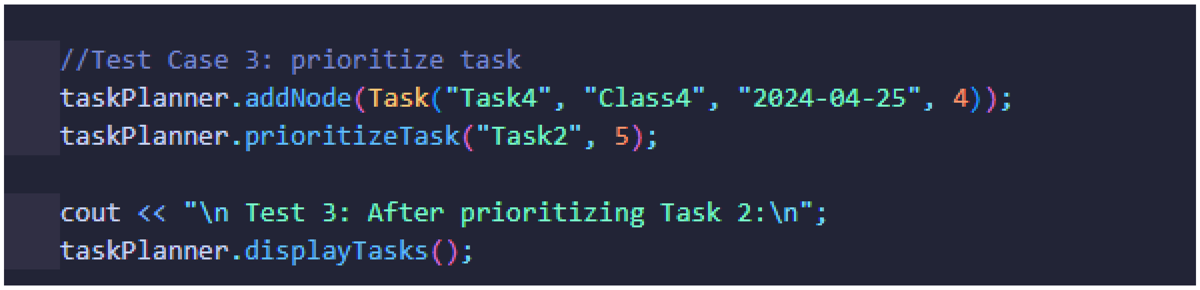


```
 Test 2: After removing and marking tasks as completed:
Title: Task3, Class: Class3, Due Date: 2024-03-05, Priority: 3, Completed: No
Title: Task2, Class: Class2, Due Date: 2024-02-24, Priority: 2, Completed: Yes
```

Fig 9. *Test case 2 in terminal.*

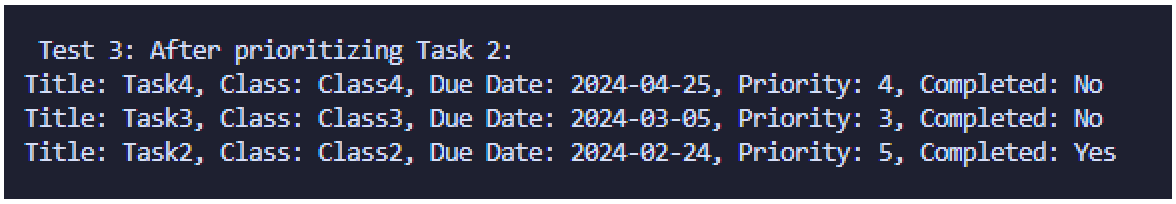
- **Third test case: Prioritize tasks**

  It will help us in changing the prioritization of a task, wether we want it to make it more important or less. This is done by calling “prioritizeTask()” to update the priority. After updating, display the tasks to see if it was updated correctly.



```
//Test Case 3: prioritize task
taskPlanner.addNode(Task("Task4", "Class4", "2024-04-25", 4));
taskPlanner.prioritizeTask("Task2", 5);

cout << "\n Test 3: After prioritizing Task 2:\n";
taskPlanner.displayTasks();
```

Fig 10. *Test case 3: Prioritizing tasks.*

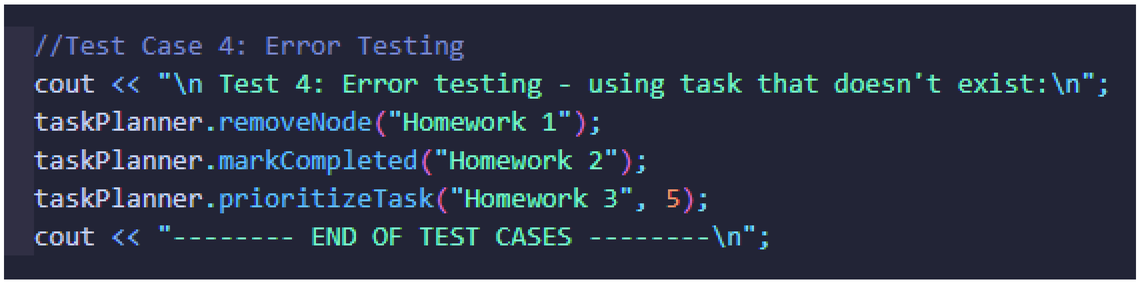


```
 Test 3: After prioritizing Task 2:
Title: Task4, Class: Class4, Due Date: 2024-04-25, Priority: 4, Completed: No
Title: Task3, Class: Class3, Due Date: 2024-03-05, Priority: 3, Completed: No
Title: Task2, Class: Class2, Due Date: 2024-02-24, Priority: 5, Completed: Yes
```

Fig 11. *Test case 3 in terminal.*

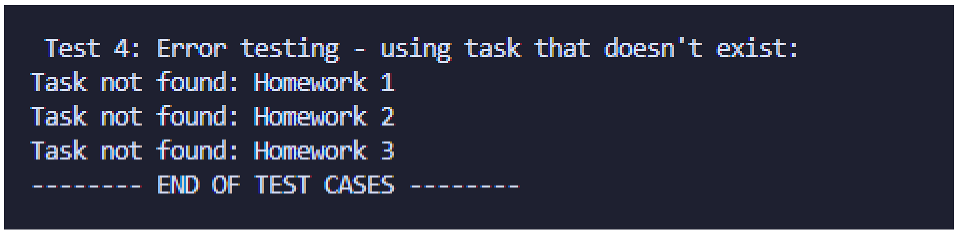
- **Fourth test case: Testing errors**

  In this last test case, the error handling will be realized by trying to remove/delete, marking as complete or change the priority of a task that doesn't exist in our list. This would help in verifying the error messages.



```cpp
//Test Case 4: Error Testing
cout << "\n Test 4: Error testing - using task that doesn't exist:\n";
taskPlanner.removeNode("Homework 1");
taskPlanner.markCompleted("Homework 2");
taskPlanner.prioritizeTask("Homework 3", 5);
cout << "-------- END OF TEST CASES --------\n";
```

Fig 12. *Test case 4: Testing errors.*



```
 Test 4: Error testing - using task that doesn't exist:
Task not found: Homework 1
Task not found: Homework 2
Task not found: Homework 3
-------- END OF TEST CASES --------
```

Fig 13. *Test case 4 in terminal.*