

**Data Structures and Algorithms ES-221**

**Semester Project Report**

**Task Scheduler**

**2023706; 2023725**

**Project overview:**

The **Task Scheduler** is an application designed to help users manage, organize, and prioritize their daily tasks efficiently. It allows users to add, remove, update, and view tasks while providing a recommended execution order based on task priority and deadlines. The system ensures that users focus on the most urgent and important tasks first, promoting better time management and productivity.

This project combines key data structures such as circular linked list and priority queues with real time date and time using UNIX timestamps. Circular linked list is used to store all the tasks whether completed or not. Each node points to the next and the last node links back to the head. Using circular linked list is efficient as there is no need to check for nullptr. Each tasks includes essential details such as unique ID, description, deadline, priority levels (1-5), estimated completion time and completion status. The circular linked list maintains entire list of tasks while priority queue dynamically builds a task schedule based on priority and deadlines.

This project also provides a user-friendly menu for interaction, it allows users to perform actions such as marking tasks as completed, changing priorities of the tasks and displaying all the tasks. All date and time are set using the standard format (YYYY:MM:DD: HH:MM) and error handling ensures smooth user experience.

**Implemented data structures:**

Circular linked list:

This data structure is used to store and manage all the full list of tasks.

Custom class (TaskList) maintains circular singly linked list. Each node is the instance of Node structure which contains a task object along with a pointer to the next node.

Priority Queue:

This data structure determines the execution order of pending task.

It uses priority\_queue to automatically maintain task order based on the overhead operator. It compares the logic priorities. Whenever a task is added or modified the queue is rebuilt from the linked list to reflect updated priorities.

Custom task struct:

It encapsulates all task related data. it also implements a custom comparison operator to control the scheduling order.

**Implemented algorithms:**

Custom comparison logic:

It ensures that the priority\_queue behaves like min-heap on custom rules such that the lowest priority value and the earliest deadline comes first. This logic determines how tasks are ordered in the priority queue. If both the priorities are equal it compares the deadlines, giving preference to earlier deadlines.

Queue rebuilding strategy:

It rebuilds the queue whenever a new task is added, or any task is modified. It ensures that the priority queue always reflect the latest state of the tasks. Whenever a task is added, updated, marked as completed, the program clears the existing queue and rebuilds it by traversing the circular linked list.

Task search algorithm:

Each task is given a unique ID. The gettask(id) performs full traversal to find the task by its ID. It also searches linearly through the list over a circular linked list. The algorithm starts from the head node and traverses each node one by one. It compares the current ID with the targeted ID until it finds a match

**Performance Analysis:**

Using benchmark code, program was run multiple times and tasks were added/ removed in loop determined by us. Through <chrono> library, we obtained time required for each of the times program was run. Results were plotted for clear visual representation.

void benchmark(int numTasks) {

TaskScheduler scheduler;

auto startAdd = chrono::high\_resolution\_clock::now();

for (int i = 0; i < numTasks; ++i) {

scheduler.addTask("Task " + to\_string(i + 1), rand() % 100) }

auto endAdd = chrono::high\_resolution\_clock::now();

chrono::duration<double, milli> elapsedAdd = endAdd - startAdd;

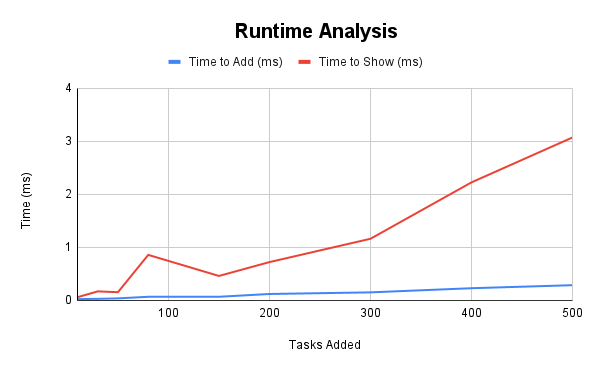
auto startShow = chrono::high\_resolution\_clock::now();

scheduler.showTasks(); // (optional, can comment if too much output for 1000+ tasks)

auto endShow = chrono::high\_resolution\_clock::now();

chrono::duration<double, milli> elapsedShow = endShow - startShow;

cout << "Time to Add Tasks: " << elapsedAdd.count() << " ms\n";

 cout << "Time to Show Tasks: " << elapsedShow.count() << " ms\n";}

Through dynamic memory allocation and reliance on data structures like queues, dependence on loops (for/if-else) was minimized. Hence it was observed that execution time did not drastically increase with number of dummy tasks added.

In order to visualize execution time for each of the functions used, we used the ‘clock ()’ to save CPU time before and after the function runs. Subtracting the before/ after values and using them to calculate passed time gave output values which are represented in the pie chart below:

**Challenges faced and solutions:**

The challenges faced while making this project were:

managing the circular linked list: Ensuring the correct node is linked while adding/removing tasks. Solution for this problem was to carefully implement addtask() and removetask() with checks of head and tail pointers and single node conditions.

Keeping priority queue updated: The queue becomes outdated after a task is added, edited or removed which led to incorrect scheduling. The solution for this problem was to implement queue rebuilding strategy which reconstructs the queue whenever a task is added, removed or modified.

Input handling: invalid inputs such as out of range priority negative time can cause problems and logic errors. The solution to overcome this issue was that we added input checks and provided clear error message to guide the user.

Future improvements:

Future improvements that could further enhance the functionality of this project providing improved task search. This can be done by replacing linear search with binary search tree to improve the look up speed. Furthermore, adding notification system for approaching deadlines using system notifications.