

Ghulam Ishaque Khan Institute of Engineering Sciences and Technology

Semester Project

Data Structures and Algorithms (ES221)

Report: **Student Helper Program**

|  |  |  |
| --- | --- | --- |
| Student Names | Reg # | Degree |
| **Ayaan Azam** | **2023596** | **ES** |
| **Raahim Ali** | **2023585** | **ES** |
| **Anas Ahmed** | **2023114** | **ES** |

**1. Introduction**

This report details the analysis of the C++ source code files provided in the Budget-Manager-Batch folder. The codebase implements a multi-functional student helper program featuring a Budget Manager, a Time Manager (including task scheduling and dependency management), and mentions a Plagiarism Detector (though its implementation is not fully provided in the analyzed files). The analysis focuses on identifying key programming concepts, data structures, and algorithms employed throughout the modules.

**2. Core Programming Concepts**

The codebase demonstrates proficiency in several fundamental and advanced programming concepts:

* Object-Oriented Programming (OOP): Heavily utilized across all modules. Key classes include Task, Priority, Stack, BudgetManager, GraphNode, Graph, Node, LinkedList, PriorityQueue, and Scheduler. Encapsulation, methods, and constructors are widely used.
* Modular Programming: The application is logically divided into separate .cpp files (e.g., budget\_manager.cpp, TM\_Task.cpp, Graph.cpp, Prioirty\_Queue.cpp, Linked\_List.cpp, scheduler.cpp) managed via includes in files like main.cpp and Time\_Manager\_Main.cpp.
* Memory Management: Dynamic memory allocation (new, delete, delete[]) is employed for managing nodes in data structures like the Stack, Linked List, and Priority Queue heap.
* Input/Output: Console I/O (std::cin, std::cout) is used for user interaction and displaying information across various modules.
* Error Handling: Basic error handling is implemented, such as throwing exceptions for queue overflow/underflow in the Priority Queue and checking for invalid user input.

**3. Data Structures Utilized**

Several fundamental data structures are implemented and used:

* **Stack:** 
  + Implementation: Custom Stack class in budget\_manager.cpp.
  + Usage: Used within the BudgetManager to maintain a history (History) of accessed budget priority ranks. push is used on access/insertion, display shows history.
* **Doubly Linked List:** 
  + Implementation: Custom LinkedList class in Linked\_List.cpp. Also used conceptually for PriorityNode in budget\_manager.cpp.
  + Usage: Stores Task objects in Time\_Manager\_Main.cpp (taskList) and scheduler.cpp (completedTasks, pendingTasks). Stores Priority objects in budget\_manager.cpp. Supports operations like append, prepend, removeByName, deletefirst, deletelast.
* **Graph:** 
  + Implementation: Custom Graph class using GraphNode in Graph.cpp. Uses an array-based adjacency list concept for dependencies.
  + Usage: Manages tasks and their dependencies in Time\_Manager\_Main.cpp. Key functions include addTask, addDependency, displayDependencies, and getExecutionOrder.
* **Priority Queue:** 
  + Implementation: Custom PriorityQueue class in Prioirty\_Queue.cpp, implemented as a Min-Heap based on task priority number.
  + Usage: Central to task scheduling. Used in Time\_Manager\_Main.cpp (taskQueue) to hold tasks ready for processing based on priority. Used in scheduler.cpp (taskQueue) to manage active tasks, enabling execution of the highest-priority task first. Supports enqueue, dequeue, peek, removeByName.
* **Arrays:** Used for storing fixed sets of data like priority names/ranks, task dependencies, and implementing the heap structure.

**4. Algorithms Employed**

* Topological Sort: Implemented in Graph::getExecutionOrder to find a valid sequence for executing tasks with dependencies. Used in Time\_Manager\_Main.cpp.
* Heap Operations: heapifyUp and heapifyDown are essential for maintaining the Min-Heap property in the PriorityQueue implementation.
* Recursion: Used in BudgetManager::DisplayAllBudget for traversing and displaying budget details.
* Linear Search: Used in Graph::findTaskIndex to locate tasks within the graph structure.

**5. Domain-Specific Concepts**

* Budget Management: The budget\_manager.cpp module implements logic for allocating a total budget based on user-defined priorities (preset or custom percentages), dividing budgets daily/weekly, and handling subdivisions.
* Task Management & Scheduling: The time management components (TM\_Task.cpp, Graph.cpp, Prioirty\_Queue.cpp, Linked\_List.cpp, Time\_Manager\_Main.cpp, scheduler.cpp) collectively provide functionality to define tasks with priorities and deadlines, establish dependencies, schedule tasks using a priority queue, and track task status.

**Efficiency Analysis – Time and Space Complexity**

This section provides an analysis of the time and space complexity of the key data structures implemented in the Student Helper Program.

**1. Stack:**

* **Time Complexity:**
  + push: O(1) - Adding an element to the top takes constant time.
  + pop: O(1) - Removing an element from the top takes constant time.
  + display: O(n) - Iterating through and displaying all n elements in the stack takes linear time.
* **Space Complexity:**
  + O(n) - The space required is proportional to the number of elements (n) stored in the stack.

**2. Doubly Linked List:**

* **Time Complexity:**
  + append: O(1) - Adding to the end (if tail pointer is maintained) takes constant time. Otherwise, it would be O(n) to traverse to the end. Assuming a tail pointer is used for efficiency.
  + prepend: O(1) - Adding to the beginning takes constant time.
  + removeByName: O(n) - In the worst case, the element to be removed might be at the end of the list, requiring traversal through all n elements.
  + deletefirst: O(1) - Removing the first element takes constant time.
  + deletelast: O(1) - Removing the last element (if tail pointer is maintained) takes constant time. Otherwise, it would be O(n) to traverse to the second to last element. Assuming a tail pointer is used for efficiency.
* **Space Complexity:**
  + O(n) - The space required is proportional to the number of elements (n) stored in the list, as each node stores data and pointers to the next and previous nodes.

**3. Graph (using Adjacency List):**

Let V be the number of vertices (tasks) and E be the number of edges (dependencies).

* **Time Complexity:**
  + addTask: O(1) on average - Adding a new vertex usually involves adding a new entry in the adjacency list array, which takes constant time.
  + addDependency: O(1) on average - Adding a dependency (edge) involves adding a neighbor to the adjacency list of a vertex, which takes constant time on average.
  + displayDependencies: O(V + E) - Iterating through all vertices and their adjacency lists takes time proportional to the number of vertices and edges.
  + getExecutionOrder (Topological Sort): O(V + E) - The typical implementation using Kahn's algorithm or Depth First Search takes time proportional to the number of vertices and edges.
  + findTaskIndex: O(V) - In the worst case, we might need to iterate through all the vertices to find the index of a specific task.
* **Space Complexity:**
  + O(V + E) - The space required is to store the adjacency list, which contains an array of size V (for each vertex) and a total of E edges across all the lists.

**4. Priority Queue (Min-Heap):**

Let n be the number of elements in the priority queue.

* **Time Complexity:**
  + enqueue: O(log n) - Inserting an element involves placing it at the end and then heapifying up, which takes logarithmic time with respect to the number of elements.
  + dequeue: O(log n) - Removing the minimum element involves swapping it with the last element, removing the last element, and then heapifying down, which takes logarithmic time.
  + peek: O(1) - Accessing the minimum element at the root of the heap takes constant time.
  + removeByName: O(n + log n) ≈ O(n) - In the worst case, we might need to search through all n elements to find the element to remove (linear time), and then perform heapify operations which take O(log n) time.
* **Space Complexity:**
  + O(n) - The space required is proportional to the number of elements (n) stored in the heap (typically implemented using an array).

**5. Arrays:**

* **Time Complexity:**
  + Accessing an element by index: O(1) - Direct access to an element at a specific index takes constant time.
  + Searching for an element (linear search): O(n) - In the worst case, we might need to iterate through all n elements to find the desired element.
* **Space Complexity:**
  + O(n) - The space required is proportional to the number of elements (n) stored in the array.

**Summary Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Structure** | **Operation** | **Time Complexity** | **Space Complexity** |
| Stack | push | O(1) | O(n) |
|  | pop | O(1) |  |
|  | display | O(n) |  |
| Doubly Linked List | append | O(1)<sup>\*</sup> | O(n) |
|  | prepend | O(1) |  |
|  | removeByName | O(n) |  |
|  | deletefirst | O(1) |  |
|  | deletelast | O(1)<sup>\*</sup> |  |
| Graph (Adj. List) | addTask | O(1) | O(V + E) |
|  | addDependency | O(1) |  |
|  | displayDependencies | O(V + E) |  |
|  | getExecutionOrder | O(V + E) |  |
|  | findTaskIndex | O(V) |  |
| Priority Queue (Heap) | enqueue | O(log n) | O(n) |
|  | dequeue | O(log n) |  |
|  | peek | O(1) |  |
|  | removeByName | O(n) |  |
| Arrays | Access by Index | O(1) | O(n) |
|  | Linear Search | O(n) |  |

**Note:** <sup>\*</sup> assumes the use of a tail pointer for O(1) complexity.

This analysis provides a general understanding of the efficiency of the implemented data structures. The actual performance in the Student Helper Program will depend on the specific usage patterns and the number of elements being managed.

**6. Conclusion**

The Budget-Manager-Batch codebase demonstrates a practical application of fundamental OOP principles, data structures (Stack, Doubly Linked List, Graph, Priority Queue), and algorithms (Topological Sort, Heap operations) to create a multi-functional student helper tool. The code is organized modularly, addressing distinct domains like budget management and task scheduling.