

**COMSATS University, Islamabad**

**Islamabad Campus**

**Department of Computer Science**

**Read before Attempt**

|  |  |
| --- | --- |
| **Assignment No. 1: Application of list and linked list** | |
| **Course code and Title:** CSC211, Data Structure (and Algorithm) | |
| **Instructor:** Tanveer Ahmed |  |
| **Assigned Date: September 23,2024** | **Due Date: October 7,2024** |
| **Total Marks: --** | |
| **CLO-4:** **Implement data structures and algorithms** | |
| **Instructions:**   1. This is an individual assignment. You will submit your work individually through your logins (course portal) 2. Try to get the concepts, consolidate your concepts and ideas from these questions 3. You should concern **recommended books** for clarify your concepts as handouts are not sufficient. 4. **Try to make solution by yourself and protect your work from other students. If I found the solution files of some students are same then I will reward zero marks to all those students.** 5. Deadline for this assignment is **October 7, 2024.** This deadline will not be extended. | |

|  |
| --- |
| **Question # 1** |
| Imagine you are developing a task management system for a to-do list application. Each task is stored as a node in a singly linked list, where each node contains the following fields:   * **Task Name** (a string representing the task's title), * **Task Description** (a brief explanation of the task), * **Priority** (indicating the task's urgency), * **Due Date** (the deadline for completing the task), * **Next** (a pointer to the next task in the list).   Implement the following linked list operation:   1. **Add** a new task based on the task's urgency. 2. **Remove** a task based on its Task Name. 3. **Print** all tasks in reverse order (without changing the structure of the linked list). |
| **Question # 2** |
| You are designing a patient registration system for a hospital. Each patient is stored as a node in a singly linked list, where each node contains the following fields:   * **Registration Number** (a unique integer for each patient), * **Patient Name** (the name of the patient), * **Age** (the patient's age), * **Address** (the patient's home address), * **Phone Number** (the patient's contact information), * **Next** (a pointer to the next registered patient in the list).   Patients are registered one after the other. Implement the following:   1. **Register**(add) a new patient at the end of the list. 2. **Remove** a patient when they are discharged (based on their **Registration Number**). 3. **Find** and print the details of a patient using their **Registration Number**. |
| **Question # 3** |
| Consider a navigation system that stores the locations visited during a road trip. Each location is stored as a node in a doubly linked list, where each node contains the following fields:   * **Location Name** (the name of the visited location), * **Coordinates** (GPS coordinates or location data), * **Visited Date** (the date the location was visited), * **Description** (additional notes or information about the location), * **Previous** (a pointer to the previous location in the list), * **Next** (a pointer to the next location in the list).   This allows the user to navigate both forward and backward through the locations. Implement the following to:   1. **Add** a new location after the current location. 2. **Remove** a specific location by its **Location Name**. 3. **Print** all the locations from the most recently visited back to the first location. |
| **Question # 4** |
| In an e-commerce platform, users can add items to their cart, and later remove or modify them. Each item in the cart is stored as a node in a doubly linked list, where each node contains the following fields:   * **Item ID** (a unique identifier for the item), * **Item Name** (the name of the item), * **Quantity** (the quantity of the item in the cart), * **Price** (the price of the item), * **Description** (additional details about the item), * **Previous** (a pointer to the previous item in the cart), * **Next** (a pointer to the next item in the cart).   Implement the following to:   1. **Add** a new item after the current item in the cart. 2. **Remove** the current item from the cart. 3. **Move** to the previous or next item in the cart. |
| **Question # 5** |
| A ticketing system for a theme park uses a circular linked list to manage visitors waiting for a ride. Each visitor is stored as a node in the list, where each node contains the following fields:   * **Visitor ID** (a unique identifier for the visitor), * **Visitor Name** (the name of the visitor), * **Ticket Type** (type of ticket, such as VIP or General), * **Ride Count** (how many rides the visitor has completed), * **Next** (a pointer to the next visitor in the queue, forming a circular linked list).   When a visitor completes their turn on a ride, they are moved to the end of the line, forming a loop. Implement the following to:   1. **Add** a visitor to the end of the circular linked list . 2. **Remove** the visitor at the front of the circular linked list after they have completed their turn. 3. Simulate the process of moving visitors to the end of the queue after their turn on the ride. |
| **Question # 6** |
| In a music playlist application, songs are stored in a circular linked list, where the playlist automatically loops back to the first song after the last song. Each song is stored as a node, where each node contains the following fields:   * **Song ID** (a unique identifier for the song), * **Song Title** (the name of the song), * **Artist** (the name of the artist or band), * **Duration** (the song's length in minutes and seconds), * **Next** (a pointer to the next song in the playlist, forming a circular linked list).   Implement the following to:   1. **Add** a new song to the playlist after the current song. 2. **Remove** the current song from the playlist. 3. **Traverse and play** the next song, ensuring that the playlist loops back to the first song after reaching the last one. |
| **Question # 7** |
| You are working for a company that manages flight data for multiple airports. The company stores information about flights between different airports, but only a small percentage of the potential routes between these airports have actual flights, making the data sparse. You need to implement a system to efficiently store, manipulate, and process this sparse flight data using a sparse matrix represented as a linked list.  Each entry in the sparse matrix will store the number of flights between two airports (represented by rows and columns of the matrix). The system should store only non-zero values (i.e., existing flights), as the majority of routes have no flights.  Implement the following operations:   1. **Store flight data**: Input and store only non-zero entries in the matrix using a linked list. 2. **Transpose the flight matrix**: Implement a function to transpose the matrix, where rows represent departure airports and columns represent arrival airports. The transpose flips the matrix, showing flights arriving at each airport instead of departing.   **Add flight data from two matrices**: Combine the flight data for two different weeks., Adding the flight data for two different weeks gives the total number of flights across both weeks.   1. **Multiply flight matrices**: Multiply two flight matrices to analyze connecting flights between airports via intermediate airports. This can help analyze indirect flight connections, i.e., flights from Airport A to Airport C via Airport B. This could be useful for determining connecting flights between airports. 2. Finally, write a function to print the matrix in a readable format, showing all airports and the number of flights between them.   **Additional Explanation**:  2D array is used to represent a sparse matrix in which there are three rows named as   * **Row:**Index of row, where non-zero element is located * **Column:**Index of column, where non-zero element is located * **Value:**Value of the non-zero element located at index – (row, column)   Sparse Matrix Array Representation  In the linked list, **each node has four fields**. These four fields are defined as:   * **Row:**Index of row, where non-zero element is located * **Column:**Index of column, where non-zero element is located * **Value:**Value of the non-zero element located at index – (row, column) * **Next node:**Address of the next node     Now Sparse matrix can be represented using linked list as follows:    **2nd Method**  One of the possible representation of sparse matrix is List of Lists (LIL). In this representation, one list is used to represent the rows, and each row contains the list of triples: Column index, Value (non – zero element) and address field, for non – zero elements.    For the best performance both lists should be stored in order of ascending keys. Now sparse matrix can be represented as: |

The following are rubrics to assess the implementations for each of the problems described above. The rubrics focus on correctness, functionality, efficiency, and code structure.

1. **To-Do List Task Management System (Singly Linked List)**

**Rubric:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Criteria | Excellent (4) | Good (3) | Fair (2) | Poor (1) |
| Add Task to End | Correctly adds task to end, maintains list structure | Adds task but with minor issues (e.g., edge cases) | Partially working, significant issues or edge cases missing | Fails to add tasks properly |
| Remove Task by Name | Correctly removes the task, handles all cases | Removes task but with minor issues in edge cases | Task removal has several flaws | Fails to remove tasks |
| Print in Reverse Order | Correctly prints tasks in reverse without altering list | Correctly prints in reverse but inefficient | Incorrect or inefficient print | Fails to print tasks |
| Code Structure & Efficiency | Well-structured, optimized, follows good practices | Structured code with minor inefficiencies | Code works but lacks structure | Poorly written or inefficient code |

1. **2. Patient Registration System (Singly Linked List)**

**Rubric:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Criteria | Excellent (4) | Good (3) | Fair (2) | Poor (1) |
| Add New Patient | Correctly registers and adds patient to end of the list | Registers patient, minor issues | Partially registers patient | Fails to register patient |
| Remove Patient (Discharge) | Correctly removes patient by registration number | Removes patient with minor issues | Flawed implementation, issues with removal | Fails to remove patient |
| Find Patient by Registration No | Finds and returns patient details correctly | Finds patient, minor flaws in edge cases | Partially works, significant issues | Fails to find patient |
| Code Structure & Efficiency | Optimized, clear, and easy-to-follow code | Structured with minor inefficiencies | Code works but not well-structured | Inefficient and disorganized code |

1. **3. Navigation System (Doubly Linked List)**

**Rubric:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Criteria | Excellent (4) | Good (3) | Fair (2) | Poor (1) |
| Add Location | Correctly adds a location after current, maintains structure | Adds location with minor issues | Adds location but with significant issues | Fails to add location |
| Remove Location by Name | Correctly removes the location, handles all cases | Removes location but with minor issues | Removal has several flaws | Fails to remove location |
| Print Locations in Reverse | Correctly prints all locations in reverse | Prints in reverse but inefficiently | Partially correct, but flawed logic | Fails to print in reverse |
| Code Structure & Efficiency | Well-structured, optimized, follows good practices | Minor inefficiencies in code structure | Code works but lacks structure | Poorly written or inefficient code |

1. **4. E-Commerce Cart System (Doubly Linked List)**

**Rubric:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Criteria | Excellent (4) | Good (3) | Fair (2) | Poor (1) |
| Add Item to Cart | Correctly adds item after current, maintains list structure | Adds item but minor issues | Significant issues with adding items | Fails to add items properly |
| Remove Current Item | Correctly removes the current item, handles edge cases | Removes item but minor issues | Partially works with significant issues | Fails to remove item |
| Move to Next/Previous Item | Correctly moves through list, circular behavior | Moves through list, but has inefficiencies | Significant flaws in traversal | Fails to navigate the list |
| Code Structure & Efficiency | Optimized, clear, and well-structured code | Minor inefficiencies, but structured | Functional but unoptimized or poorly structured code | Inefficient or disorganized code |

1. **5. Theme Park Ticketing System (Circular Linked List)**

**Rubric:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Criteria | Excellent (4) | Good (3) | Fair (2) | Poor (1) |
| Add Visitor to Queue | Correctly adds a visitor to the end of the queue | Adds visitor but with minor issues | Partially working, significant issues | Fails to add visitor |
| Remove Visitor from Queue | Correctly removes the first visitor after their turn | Removes visitor but has minor issues | Significant issues in removal | Fails to remove visitor |
| Simulate Moving Visitors | Accurately moves visitors to the back of the queue | Simulation mostly correct but inefficient | Partially simulates, significant issues | Fails to simulate visitor movement |
| Code Structure & Efficiency | Well-structured, efficient, and follows good practices | Structured code with minor inefficiencies | Code works but lacks structure | Poorly written or inefficient code |

1. **6. Music Playlist System (Circular Linked List)**

**Rubric:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Criteria | Excellent (4) | Good (3) | Fair (2) | Poor (1) |
| Add Song After Current | Correctly adds a song after the current song | Adds song but minor inefficiencies | Adds song but significant issues | Fails to add song |
| Remove Current Song | Correctly removes the current song and maintains structure | Removes song but has minor issues | Partially working, flawed removal | Fails to remove song |
| Play Next Song (Looping) | Correctly moves to the next song and loops after the last one | Traverses the list, minor inefficiencies | Partially correct, but inefficient | Fails to traverse or loop back |
| Code Structure & Efficiency | Optimized, clear, and easy-to-follow code | Structured with minor inefficiencies | Code works but lacks structure | Inefficient or disorganized code |

So then good luck and happy coding. ☺