INDIAN INSTITUTE OF INFORMATION TECHNOLOGY, NAGPUR
Department of Computer Science and Engineering
CSL 210: Data Structures with Applications

Date: 27/08/2024 (Tuesday)
Time: 3:00 to 04:00 PM

Duration: 1 Hour | Semester -III(CSE,AIML,HCI, DSA), V Sem ECE | Max. Marks: **15M**

Important Instructions:

- All the questions are compulsory.

Q1 [2+2+1]M [CO1]

A smart city traffic management system represents traffic density at each intersection of two roads, as a matrix A of size m×n, where each element A[i][j] indicates the traffic density at a specific intersection. Most intersections zero traffic, whereas very few key intersections experience significant traffic, represented by non-zero values in the matrix.
The system needs to efficiently store, update, and analyse traffic data in real-time, handling frequent updates as traffic conditions change.
Solution:

(a) Which data structure would you prefer and why? Discuss the advantages and disadvantages of your chosen data structure over other available data structures.

Data Structure	**Advantages**	**Disadvantages**
Singly Linked List	Simple, dynamic size, efficient insertions/deletions	Inefficient traversal, memory overhead, limited direct access
Multi-Linked List	Efficient traversal, dynamic updates, memory efficient	Complex implementation

i) Singly Linked List: Simple and easy for scenarios where updates are needed, but less efficient for complex analyses.

ii)Multi-Linked List: Optimal for real-time updates and flexible traversal, balancing memory efficiency with dynamic capabilities.

(b) Based on your chosen data structure, describe how you would handle updating the traffic density at a specific intersection (i,j) when the value increases from zero to a non-zero value, or vice versa.

Insert a Node (Zero to Non-Zero):

When the traffic density at (i,j) increases from zero to a non-zero value, a new node needs to be inserted into the multi-linked list at the correct position in both the row and column.

Steps:

```
// Function to insert a new node into a sparse matrix
Function InsertNode(rowHeader, columnHeader, i, j, value):
    // Step 1: Locate the Position in the Row List
    currentRowNode = rowHeader[i]
    While currentRowNode.right != NULL AND currentRowNode.column < j:
        currentRowNode = currentRowNode.right
    // Step 2: Locate the Position in the Column List
    currentColumnNode = columnHeader[j]
    While currentColumnNode.down != NULL AND currentColumnNode.row < i:
        currentColumnNode = currentColumnNode.down

    // Step 3: Insert the New Node
    newNode = CreateNode(i, j, value)

    // Adjust right pointer of the previous node in the row
    newNode.right = currentRowNode.right
    currentRowNode.right = newNode

    // Adjust down pointer of the previous node in the column
    newNode.down = currentColumnNode.down
    currentColumnNode.down = newNode
End Function
```

Delete a Node (Zero to Non-Zero):

When the traffic density at (i,j) decreases from non-zero to a zero value, a node needs to be deleted from the multi-linked list from the correct position in both the row and column.

```
// Function to delete a node in a sparse matrix
Function DeleteNode(rowHeader, columnHeader, i, j):
    // Step 1: Locate the Position in the Row List
    currentRowNode = rowHeader[i]
    previousRowNode = NULL

    While currentRowNode != NULL AND currentRowNode.column < j:
        previousRowNode = currentRowNode
        currentRowNode = currentRowNode.right

    // If the node does not exist, nothing to delete
    If currentRowNode == NULL OR currentRowNode.column != j:
        Return

    // Step 2: Locate the Position in the Column List
    currentColumnNode = columnHeader[j]
    previousColumnNode = NULL
```

```
        While currentColumnNode != NULL AND currentColumnNode.row < i:
            previousColumnNode = currentColumnNode
            currentColumnNode = currentColumnNode.down

        // Step 3: Delete the Node
        // Adjust the right pointer of the previous node in the row
        If previousRowNode == NULL:
            rowHeader[i] = currentRowNode.right
        Else:
            previousRowNode.right = currentRowNode.right

        // Adjust the down pointer of the previous node in the column
        If previousColumnNode == NULL:
            columnHeader[j] = currentColumnNode.down
        Else:
            previousColumnNode.down = currentColumnNode.down

        // Optionally, free the memory of the node if required
        Free(currentRowNode)
    End Function
```

(c) Propose an algorithm to calculate the total traffic for a specific row i or column j. (stepwise)

```
Function: calculateRowTotal(matrix, row)

    Initialize total = 0.

    Set current = matrix.rowHeads[row].

    Traverse the row:

        While current is not NULL:

                Add current.value to total.

                Move to the next node: current = current.right.

    Return total.

    Function: calculateColumnTotal(matrix, col)
```

```
Initialize total = 0.

Set current = matrix.colHeads[col].

Traverse the column:

    While current is not NULL:

        Add current.value to total.

        Move to the next node: current = current.down.

Return total
```

Q2

In many operating systems, processes are managed using a priority queue, where each process is assigned a priority level that determines the order in which it should be executed. In some scenarios, the priority of a process may change dynamically during its lifecycle.

Suppose you are using a linked list-based priority queue, where processes are stored in a singly or doubly linked list, sorted by their priority levels. Each node in the linked list stores a process number and its priority (both are integers).

(a) How would you modify the existing linked list-based priority queue to efficiently handle changes in the priority of processes?

For example, a process having a priority 3, may change to 5 or 1. (Higher number indicates higher priority).

Write a pseudocode for the same.

(b) Can you use additional data structure with the existing linked list that might be necessary to support these dynamic priority adjustments efficiently. Explain how these changes will impact the overall performance of the priority queue operations.

[3+2]M [CO1]

Sol

a) Logic is as below-

1. **Initialization:**
   - Use a doubly linked list to store processes sorted by priority.
   - Maintain a hashmap or array of lists for quick access to nodes using process numbers.
2. **Insert Process:**
   - Create a new node with process number and priority.
   - Find the correct position based on priority and insert it there.
3. **Change Priority:**
   - Locate the node using the hash map.
   - Remove the node from its current position.
   - Update its priority.
   - Reinsert it at the correct position based on the new priority.
4. **Remove Process:**
   - Locate the node using the hash map.
   - Remove it from the list and update pointers.

b) Priority Buckets (Array of Lists):

	● **Modification:** Use an array of doubly linked lists, where each index corresponds to a priority level, and each list contains nodes of that priority. ● **Impact:** This approach allows O(1) access to all nodes of a given priority and O(1) time complexity for inserting or removing a node from a priority bucket. When a node's priority changes, it can be moved from one list to another in constant time. The challenge with this approach is handling a wide range of priority values, which may require a dynamic resizing mechanism. **OR** **Hash Map for Direct Access:** ● **Modification:** Maintain a hash map (or dictionary) that maps each process ID to its corresponding node in the linked list. ● **Impact:** This enables O(1) access to nodes, allowing quick updates to node priorities without needing to traverse the list to find the node. This significantly reduces the time needed to locate a node when changing its priority.	
Q3	You are given an array of strings *nums*, where each string is of equal length and consists of only digits. Trim each string from LSB position with length 1, 2, ..until maxlength of the strings, and store them in separate arrays. Example: nums = ["102","473","251","814"] Length 1: Trim1 = [“2”, “3”, “1”, “4”] Length 2: Trim2 = [“02”, “73”, “51”, “14”] Length 3: Trim3 = ["102","473","251","814"] Write a pseudocode to sort each trimmed array using non-comparison based sorting algorithm, and return kth smallest element, where k will be entered by the user. Print the kth largest element for each trimmed array separately. Solution: Function to TRIM (1M) + Radix/counting/bucket sort (3M) + kth smallest/largest (1M)	[5 M] [CO1]

****