Vivekanand Education Society's Institute of Technology, Chembur, Mumbai,
Department Of Artificial Intelligence & Data Science
Year : 2023-24 (ODD Sem)
MID TERM TEST

Class : D6AD	**Division: A/B**
Semester: III	**Subject: Data Structures**
Date: 03/09/2024	**Time: 9:00AM-10:00AM**

Q1 a. Evaluate the role of data structures in enhancing the efficiency of algorithms and software design

Data structures play a crucial role in enhancing the efficiency of algorithms and software design by providing organized and optimized ways to store and manage data. Efficient data structures, such as arrays, linked lists, trees, and hash tables, enable faster data retrieval, manipulation, and storage, which directly improves algorithm performance. Proper use of data structures reduces time and space complexity, allowing software to handle larger datasets and perform operations more quickly and effectively.

Q1 b. Define Abstract Data Type (ADT) and give an example of an ADT in programming.

Abstract Data Type (ADT) is a theoretical concept that defines a data structure by its behavior (operations) rather than its implementation. An ADT specifies *what* operations can be performed and *what* the expected outcomes are, without detailing *how* these operations are implemented.

Example: A **Stack** is a common ADT. It allows operations such as:

1. **Push** (inserting an element)
2. **Pop** (removing the top element)
3. **Peek** (viewing the top element)

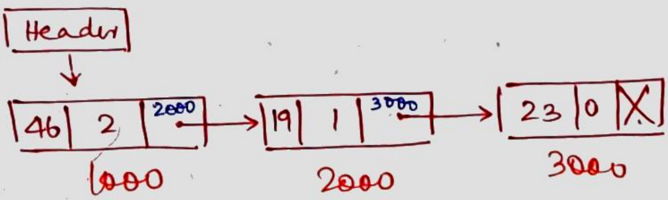
The underlying implementation (using arrays or linked lists) is hidden from the user, focusing only on the operations.

Q1 c. Explain why self-referencing nodes are important in the structure of a linked list.

Self-referencing nodes are essential in a linked list because they enable the dynamic linking of elements. Each node in a linked list contains a reference (or pointer) to the next node, allowing the list to grow or shrink as needed without predefined limits. This flexible structure facilitates efficient insertion and deletion of elements, making it ideal for dynamic data management compared to static arrays.

Q1 d. Represent the given Polynomial using a Singly Linked List

$p = 46\,x^2 + 19\,x + 23$



Q1 e. Explain how to add an element to both the front and the rear of a double-ended queue implemented using a linked list.

In a double-ended queue (deque) implemented using a linked list, you can add elements to both the front and the rear as follows:

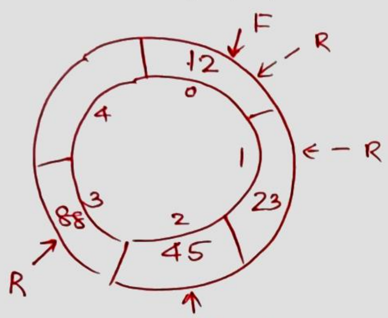
1. **Adding to the Front:**
   - Create a new node containing the element.
   - Set the new node's next pointer to the current head node.
   - Update the head pointer to this new node.
   - If the deque was empty, also set the tail pointer to this new node.
2. **Adding to the Rear:**
   - Create a new node containing the element.
   - Set the current tail node's next pointer to the new node.
   - Update the tail pointer to this new node.
   - If the deque was empty, also set the head pointer to this new node.

Q1 f. Describe the operations involved in a priority queue and how elements are prioritized.

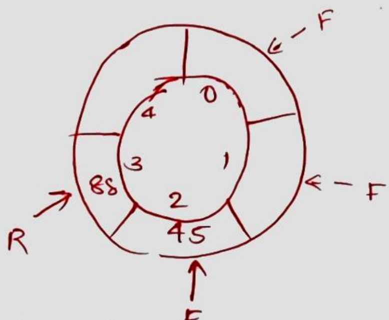
A **priority queue** is a special type of data structure where elements are processed based on their priority rather than their insertion order. The key operations involved are:

1. **Insertion (Enqueue):** Elements are added to the queue with an associated priority. Elements with higher priority are given preference over those with lower priority.
2. **Deletion (Dequeue):** The element with the highest priority is removed from the queue. If two elements have the same priority, they are processed in the order they were added.

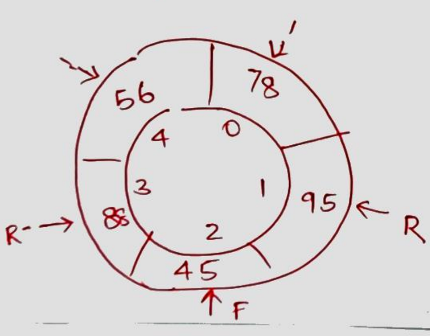
Q2 a. Illustrate the working of Circular Queue for the given scenario. Declare a CQ of size 5, Enqueue the elements 12, 23,45, 88. Dequeue two elements from the CQ, Enqueue the elements 56, 78, 95, 17.



(a) After enqueuing 4 elements in CQ.



(b) After dequeuing 2 elemts from CQ.



(c) Out of 4 elements to be enqueued 3 elements were inserted.
Enqueue 17. : CQ is Full.

Q2 b. Convert the given Infix Expression to Postfix and evaluate the Postfix Expression.
Infix Expression : 10 + 3 ^ 5 / (16 - 4)

Input String	Output Stack	Operator Stack
10 + 3 ^ 5 / (16 - 4)	1	
10 + 3 ^ 5 / (16 - 4)	10	
10 + 3 ^ 5 / (16 - 4)	10	
10 + 3 ^ 5 / (16 - 4)	10	+
10 + 3 ^ 5 / (16 - 4)	10	+
10 + 3 ^ 5 / (16 - 4)	10 3	+
10 + 3 ^ 5 / (16 - 4)	10 3	+
10 + 3 ^ 5 / (16 - 4)	10 3	+^
10 + 3 ^ 5 / (16 - 4)	10 3	+^
10 + 3 ^ 5 / (16 - 4)	10 3 5	+^
10 + 3 ^ 5 / (16 - 4)	10 3 5	+^
10 + 3 ^ 5 / (16 - 4)	10 3 5 ^	+/
10 + 3 ^ 5 / (16 - 4)	10 3 5 ^	+/
10 + 3 ^ 5 / (16 - 4)	10 3 5 ^	+/(
10 + 3 ^ 5 / (16 - 4)	10 3 5 ^ 1	+/(
10 + 3 ^ 5 / (16 - 4)	10 3 5 ^ 16	+/(
10 + 3 ^ 5 / (16 - 4)	10 3 5 ^ 16	+/(
10 + 3 ^ 5 / (16 - 4)	10 3 5 ^ 16	+/(-
10 + 3 ^ 5 / (16 - 4)	10 3 5 ^ 16	+/(-
10 + 3 ^ 5 / (16 - 4)	10 3 5 ^ 16 4	+/(-
10 + 3 ^ 5 / (16 - 4)	10 3 5 ^ 16 4-	+/
10 + 3 ^ 5 / (16 - 4)	10 3 5 ^ 16 4-/+	

Eval. of Postfix

10 3 5 ^ 16 4 - / +

10 243 16 4 - / +

10 243 12 / +

10 20.25 +

30.25

Input ~~String~~ character	Stack	O/P
10	10	30.25
3	10 3	
5	10 3 5	
^	10 243	
16	10 243 16	
4	10 243 16 4	
-	10 243 12	
/	10 ~~243~~ 20.25	
+	30.25.	

Q3 a. Write a Function in C to implement the following for Singly Circular Linked List.
(i) Insert a node in the start the list
(ii) Delete a node given its value.

```
// Function to insert a node at the start of the circular singly linked list
void insertAtStart(struct Node** head, int data) {
    // Allocate memory for the new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    // If the list is empty
    if (*head == NULL) {
        newNode->next = newNode; // Point to itself, as it's the only node
        *head = newNode;
    } else {
        // Traverse to the last node to maintain the circular property
        struct Node* temp = *head;
        while (temp->next != *head) {
            temp = temp->next;
        }
        // Insert the new node at the beginning
        newNode->next = *head;
        temp->next = newNode;  // Update the last node's next pointer
        *head = newNode;       // Update head to the new node
    }
}
// Function to delete a node given its value
void deleteNode(struct Node** head, int key) {
    // If the list is empty, return
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node *current = *head, *prev = NULL;
    // Case 1: If the node to be deleted is the head node
    if (current->data == key) {
        // If there is only one node in the list
        if (current->next == *head) {
            free(current); // Free the only node
            *head = NULL;  // The list becomes empty
            return;
        }
        // If there are more than one nodes, find the last node
        struct Node* temp = *head;
        while (temp->next != *head) {
            temp = temp->next;
        }
        temp->next = current->next; // Update last node's next pointer
        *head = current->next;      // Update head to the next node
        free(current);              // Free the old head node
        return;
    }
```

```
    // Case 2: If the node to be deleted is not the head node
    while (current->next != *head && current->data != key) {
        prev = current;
        current = current->next;
    }
    // If the node with the given key is found
    if (current->data == key) {
        prev->next = current->next; // Update the previous node's next pointer
        free(current);          // Free the current node
    } else {
        printf("Node with value %d not found.\n", key);
    }
}
```

Q3 b. Write a program to implement Queue using Linked List

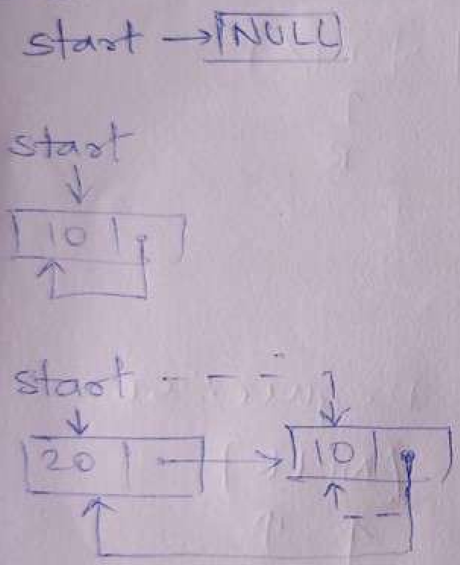
```
#include <stdio.h>
#include <stdlib.h>
// Define a node structure
struct Node {
    int data;
    struct Node* next;
};
// Define a queue structure
struct Queue {
    struct Node* front;
    struct Node* rear;
};
// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
// Function to create a new queue
struct Queue* createQueue() {
    struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
    q->front = q->rear = NULL;
    return q;
}
// Function to enqueue an element to the queue
void enqueue(struct Queue* q, int data) {
    struct Node* newNode = createNode(data);
    if (q->rear == NULL) {
        q->front = q->rear = newNode;
```

```c
        return;
    }
    q->rear->next = newNode;
    q->rear = newNode;
}
// Function to dequeue an element from the queue
int dequeue(struct Queue* q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return -1; // Return -1 if queue is empty
    }
    struct Node* temp = q->front;
    int data = temp->data;
    q->front = q->front->next;
    if (q->front == NULL) {
        q->rear = NULL;
    }
    free(temp);
    return data;
}
// Function to display the contents of the queue
void displayQueue(struct Queue* q) {
    struct Node* temp = q->front;
    if (temp == NULL) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
// Main function to test the queue implementation
int main() {
    struct Queue* q = createQueue();
    enqueue(q, 10);
    enqueue(q, 20);
    enqueue(q, 30);
    displayQueue(q);
    printf("Dequeued: %d\n", dequeue(q));
    printf("Dequeued: %d\n", dequeue(q));
```

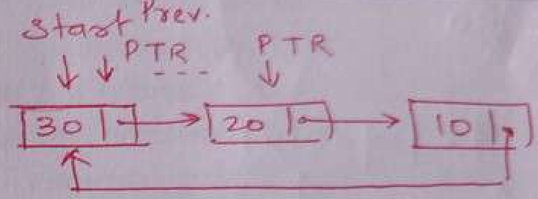
```
    displayQueue(q);
    enqueue(q, 40);
    enqueue(q, 50);
    displayQueue(q);
    // Freeing up memory (optional but good practice)
    while (q->front != NULL) {
        dequeue(q);
    }
    free(q);

    return 0;
}
```

Q3b. Q3a.

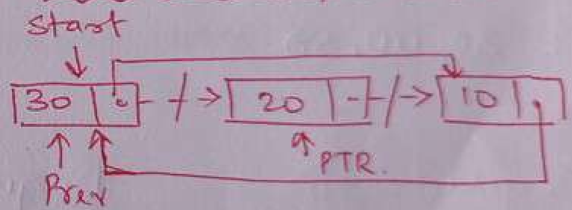
CLL



Insert @ Beg



Delete 20 from CLL

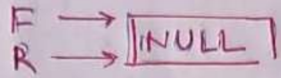


Prev. NEXT = PTR. NEXT

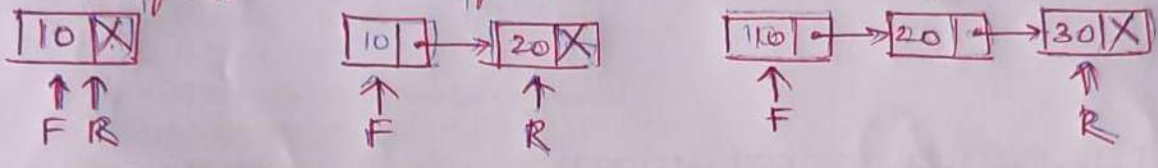
Delete a specific node

Q3b.

Initial



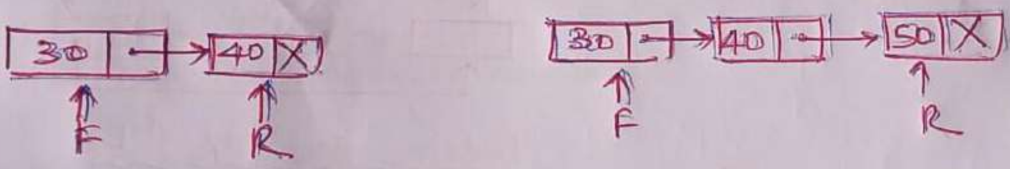
Enque 10 Enque 20 Enque 30



Queue: 10, 20, 30

Deque: 10 Deque: 20

Queue: 30



Queue: 30, 40, 50

Deque: 30 Deque: 40 Deque: 50