DATA STRUCTURE NOTES

Unit2: Queue & Linked List:

Queue ek **linear data structure** hai jo **FIFO (First In, First Out)** principle follow karta hai. Matlab jo element sabse pehle insert kiya gaya hoga, wahi sabse pehle remove hoga.

👉 Isse aap real-life example se samajh sakte hain:
Jaise ticket counter par line (queue) me jo person pehle aata hai, use pehle ticket milta hai, aur jo baad me aata hai, use apne turn ka wait karna padta hai.

Queue ke Basic Operations

1. **Enqueue (Insertion)** → Queue ke **rear** (pichle end) par naya element add karna.
2. **Dequeue (Deletion)** → Queue ke **front** (aage wale end) se element remove karna.
3. **Peek/Front** → Queue ke front element ko bina remove kiye dekhna.
4. **isEmpty** → Check karna ki queue khaali hai ya nahi.
5. **isFull** → Check karna ki queue full hai ya nahi (agar array based implementation ho).

Queue ke Types

1. **Simple Queue (Linear Queue)** → Normal FIFO structure.
2. **Circular Queue** → Queue ka circular representation jisme space efficiently use hoti hai.
3. **Priority Queue** → Elements ko unki priority ke basis par serve kiya jata hai (FIFO nahi hota).
4. **Deque (Double Ended Queue)** → Insertion aur Deletion dono end se allowed hota hai.

Queue Representation

- **Array based**
- **Linked list based**

Queue using Array

Isme ek fixed-size array hota hai jisme hum front aur rear pointers ka use karke elements insert/delete karte hain.

Code:

```
#include <stdio.h>

#define SIZE 5
```

```
int queue[SIZE];
int front = -1, rear = -1;
// Enqueue (insert)
void enqueue(int value) {
    if (rear == SIZE - 1) {
        printf("Queue is Full!\n");
    } else {
        if (front == -1) front = 0;
        rear++;
        queue[rear] = value;
        printf("%d inserted into queue\n", value);
    }
}
// Dequeue (delete)
void dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue is Empty!\n");
    } else {
        printf("%d deleted from queue\n", queue[front]);
        front++;
    }
}
// Display
void display() {
    if (front == -1 || front > rear) {
        printf("Queue is Empty!\n");
    } else {
        printf("Queue elements: ");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
```

```
        }
        printf("\n");
    }
}
int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    dequeue();
    display();
    return 0;
}
```

Queue using Linked List

Isme har element ek **node** ke form me hota hai jisme data aur pointer (next) hota hai. Queue ke liye hum front aur rear pointer maintain karte hain.

Code:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node *front = NULL, *rear = NULL;
// Enqueue (insert)
void enqueue(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (rear == NULL) {
```

```
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
    printf("%d inserted into queue\n", value);
}
// Dequeue (delete)
void dequeue() {
    if (front == NULL) {
        printf("Queue is Empty!\n");
        return;
    }
    struct Node* temp = front;
    printf("%d deleted from queue\n", front->data);
    front = front->next;
    if (front == NULL) {
        rear = NULL; // Queue empty ho gaya
    }
    free(temp);
}
// Display
void display() {
    if (front == NULL) {
        printf("Queue is Empty!\n");
        return;
    }
    struct Node* temp = front;
    printf("Queue elements: ");
    while (temp != NULL) {
```

```
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    dequeue();
    display();
    return 0;
}
```

Array Queue: Simple hota hai lekin fixed size ka problem hota hai.
Linked List Queue: Flexible hota hai, dynamic memory ke saath grow/shrink karta hai.

Dequeue:

Dequeue (Double Ended Queue)

Dequeue ek **special type ka queue** hota hai jisme **insertion aur deletion dono ends (front aur rear) se kar sakte hain.**
Isse **Double Ended Queue** ya **Deque** bhi kehte hain.

Normal Queue me:

- Insertion hamesha **rear** se hoti hai.
- Deletion hamesha **front** se hoti hai.

Dequeue me:

- Insertion **rear aur front dono se** ho sakti hai.
- Deletion **rear aur front dono se** ho sakti hai.

Types of Dequeue

1. **Input Restricted Deque**
    - Insertion sirf **rear end** par allowed hai.

- Deletion dono ends (front aur rear) se allowed hai.

2. **Output Restricted Deque**
   - Deletion sirf **front end** se allowed hai.
   - Insertion dono ends (front aur rear) se allowed hai.

Dequeue ke Operations

1. **InsertFront(x)** → Front par element add karna.
2. **InsertRear(x)** → Rear par element add karna.
3. **DeleteFront()** → Front se element remove karna.
4. **DeleteRear()** → Rear se element remove karna.
5. **PeekFront() / PeekRear()** → Element dekhna bina remove kiye.

Representation

- **Array based Deque** (fixed size, overflow ho sakta hai)
- **Doubly Linked List based Deque** (dynamic size, jyada flexible)

Example (Array based representation):

Initial: [ ] (empty)

- InsertRear(10) → [10]
- InsertRear(20) → [10, 20]
- InsertFront(5) → [5, 10, 20]
- DeleteRear() → [5, 10]
- DeleteFront() → [10]

1. **Deque using Array (C Language)**

```
#include <stdio.h>

#define SIZE 5

int deque[SIZE];

int front = -1, rear = -1;

// Insert at front

void insertFront(int x) {
```

```
    if ((front == 0 && rear == SIZE - 1) || (front == rear + 1)) {
        printf("Deque is Full!\n");
        return;
    }
    if (front == -1) { // empty
        front = rear = 0;
    } else if (front == 0) {
        front = SIZE - 1;
    } else {
        front--;
    }
    deque[front] = x;
    printf("%d inserted at front\n", x);
}
// Insert at rear
void insertRear(int x) {
    if ((front == 0 && rear == SIZE - 1) || (front == rear + 1)) {
        printf("Deque is Full!\n");
        return;
    }
    if (front == -1) { // empty
        front = rear = 0;
    } else if (rear == SIZE - 1) {
        rear = 0;
    } else {
        rear++;
    }
    deque[rear] = x;
    printf("%d inserted at rear\n", x);
}
```

```
// Delete from front
void deleteFront() {
    if (front == -1) {
        printf("Deque is Empty!\n");
        return;
    }
    printf("%d deleted from front\n", deque[front]);
    if (front == rear) { // only one element
        front = rear = -1;
    } else if (front == SIZE - 1) {
        front = 0;
    } else {
        front++;
    }
}
// Delete from rear
void deleteRear() {
    if (front == -1) {
        printf("Deque is Empty!\n");
        return;
    }
    printf("%d deleted from rear\n", deque[rear]);
    if (front == rear) { // only one element
        front = rear = -1;
    } else if (rear == 0) {
        rear = SIZE - 1;
    } else {
        rear--;
    }
}
```

```
// Display
void display() {
    if (front == -1) {
        printf("Deque is Empty!\n");
        return;
    }
    printf("Deque elements: ");
    int i = front;
    while (1) {
        printf("%d ", deque[i]);
        if (i == rear) break;
        i = (i + 1) % SIZE;
    }
    printf("\n");
}
int main() {
    insertRear(10);
    insertRear(20);
    insertFront(5);
    display();
    deleteFront();
    deleteRear();
    display();
    return 0;
}
```

2. Deque using Doubly Linked List (C Language)

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
```

```
    struct Node *prev, *next;
};
struct Node *front = NULL, *rear = NULL;
// Insert at front
void insertFront(int x) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = x;
    newNode->prev = NULL;
    newNode->next = front;
    if (front == NULL) {
        front = rear = newNode;
    } else {
        front->prev = newNode;
        front = newNode;
    }
    printf("%d inserted at front\n", x);
}
// Insert at rear
void insertRear(int x) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = x;
    newNode->next = NULL;
    newNode->prev = rear;
    if (rear == NULL) {
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
    printf("%d inserted at rear\n", x);
```

```
}
// Delete from front
void deleteFront() {
    if (front == NULL) {
        printf("Deque is Empty!\n");
        return;
    }
    struct Node* temp = front;
    printf("%d deleted from front\n", front->data);
    front = front->next;
    if (front != NULL) {
        front->prev = NULL;
    } else {
        rear = NULL;
    }
    free(temp);
}
// Delete from rear
void deleteRear() {
    if (rear == NULL) {
        printf("Deque is Empty!\n");
        return;
    }
    struct Node* temp = rear;
    printf("%d deleted from rear\n", rear->data);
    rear = rear->prev;
    if (rear != NULL) {
        rear->next = NULL;
    } else {
        front = NULL;
```

```
    }
    free(temp);
}
// Display
void display() {
    if (front == NULL) {
        printf("Deque is Empty!\n");
        return;
    }
    struct Node* temp = front;
    printf("Deque elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
int main() {
    insertRear(10);
    insertRear(20);
    insertFront(5);
    display();
    deleteFront();
    deleteRear();
    display();
    return 0;
}
```

Array Deque → Fast but fixed size hota hai.
Linked List Deque → Dynamic hota hai, size ka limitation nahi hota.

Priority Queue:

Priority Queue kya hota hai?

Priority Queue ek **abstract data type** hai jo queue ki tarah hi hota hai, lekin yaha elements **priority ke basis par dequeue (delete)** hote hain.

- Har element ke sath ek **priority value** hoti hai.
- Jis element ki **highest priority** hai, wo sabse pehle nikalta hai (FIFO nahi).
- Agar do elements ki priority same hai, toh wo normal queue ke order me nikalte hain.

Representation:

1. **Array based**
2. **Linked list based**
3. **Heap based (best performance)**

Priority Queue – Recursive Definition

Problem Definition

Priority Queue elements ka ek collection hai, jisme har element ke sath ek priority hoti hai.

- **Insert(x, p):** element x with priority p ko queue me daalo.
- **Delete():** sabse highest-priority element (chhoti priority value) ko remove karo.

Recursive Idea (Simple Problem → Smaller Subproblem)

1. **InsertRecursive(Q, x, p):**
    - Agar queue khaali hai → ek new node banao aur return karo.
    - Agar current node ki priority > p → x is position pe aayega.
    - Nahi toh → baki list pe recursively insert karo.
2. **DeleteRecursive(Q):**
    - Agar queue khaali hai → kuch mat karo.
    - Warna front node delete karo aur uske next ko return karo.

Recursive Implementation in C (Linked List Based):

```
#include <stdio.h>
#include <stdlib.h>
// Node structure
struct Node {
```

```
    int data;
    int priority;
    struct Node* next;
};
// Recursive insert function
struct Node* insertRecursive(struct Node* head, int data, int priority) {
    // Base case: agar list empty hai ya new node ki priority chhoti hai
    if (head == NULL || priority < head->priority) {
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = data;
        newNode->priority = priority;
        newNode->next = head;
        return newNode;
    }
    // Recursive case
    head->next = insertRecursive(head->next, data, priority);
    return head;
}
// Recursive delete (remove highest priority element)
struct Node* deleteRecursive(struct Node* head) {
    if (head == NULL) {
        printf("Priority Queue Empty!\n");
        return NULL;
    }
    printf("Deleted: %d (P=%d)\n", head->data, head->priority);
    struct Node* temp = head->next;
    free(head);
    return temp; // remaining queue
}
// Recursive display
```

```
void displayRecursive(struct Node* head) {
    if (head == NULL) {
        printf("\n");
        return;
    }
    printf("(%d, P=%d) ", head->data, head->priority);
    displayRecursive(head->next); // recurse for next
}
// Driver code
int main() {
    struct Node* pq = NULL;
    pq = insertRecursive(pq, 10, 2);
    pq = insertRecursive(pq, 5, 1);
    pq = insertRecursive(pq, 20, 3);
    pq = insertRecursive(pq, 15, 2);
    printf("Priority Queue: ");
    displayRecursive(pq);
    pq = deleteRecursive(pq);
    printf("After Deletion: ");
    displayRecursive(pq);
    return 0;
}
```

Dry Run Example

Insert operations (recursive):

- Insert(10,2) → (10, P=2)
- Insert(5,1) → (5, P=1) → (10, P=2)
- Insert(20,3) → (5, P=1) → (10, P=2) → (20, P=3)
- Insert(15,2) → (5, P=1) → (10, P=2) → (15, P=2) → (20, P=3)

DeleteRecursive:

- Deletes (5, P=1) → Remaining (10, P=2) → (15, P=2) → (20, P=3)

Key Learning (Recursive Definition of Simple Problems)

1. **InsertRecursive** → problem ko todta hai into *insert into smaller list*.
2. **DeleteRecursive** → sirf front element ko remove karta hai aur baki list return karta hai.
3. **DisplayRecursive** → har node print karta hai aur chhoti list pe call lagata hai.

Ye ek **linked list based priority queue ka recursive implementation** hai jo **divide and recurse approach** use karta hai.

Advantages and Limitations of Recursion:

◈ Advantages of Recursion

1. **Simplicity / Code clarity**
   - Complex problems ko chhoti sub-problems me tod kar easily likha ja sakta hai.
   - Example: factorial, Fibonacci series, tree traversal, graph traversal.
2. **Reduced code length**
   - Recursion se chhoti aur readable code milti hai (loops aur stacks manually likhne ki zarurat nahi).
3. **Natural fit for hierarchical problems**
   - Jisme problem naturally divide hoti hai (Tree, Graph, Divide-and-Conquer algorithms).
4. **Backtracking problems me useful**
   - Jaise Maze solving, N-Queens, Sudoku, DFS etc.
5. **Mathematical definition ke saath match karta hai**

   Example:

   Factorial(n) = n * Factorial(n-1)

   Fibonacci(n) = Fib(n-1) + Fib(n-2)

◈ Limitations of Recursion

1. **Overhead of function calls**
   - Har recursive call ke liye stack frame banta hai → memory overhead badh jata hai.
2. **Stack Overflow Risk**
   - Agar recursion properly terminate na ho (base case missing ho) → infinite recursion aur crash ho sakta hai.
3. **Less Efficient**

- Recursive calls me baar-baar function call overhead hota hai.
- Example: Fibonacci recursive implementation → exponential time (O(2^n)).

4. **Difficult to Debug**
   - Recursive functions ko trace karna aur samajhna mushkil ho jata hai for beginners.
5. **Iterative Solution Faster Ho Sakti Hai**
   - Kai problems (like factorial, sum of numbers) simple loops se efficient tarike se solve ho jati hain.

🔷 **Summary**

✅ **Use Recursion** jab:

- Problem naturally recursive ho (tree traversal, divide and conquer).
- Code readability important ho.

❌ **Avoid Recursion** jab:

- Performance/memory critical ho.
- Iterative solution easily available aur fast ho.

Linked list singly,doubly and circular lists (Array and linked representation):

🔷 1. Singly Linked List (SLL)

👉 Har node me **data + pointer (next)** hota hai jo agle node ko point karta hai.
👉 Last node ka next = NULL.

Structure:

Head → [Data | Next] → [Data | Next] → [Data | NULL]

C Structure:

```
struct Node {
    int data;
    struct Node* next;
};
```

✅ **Advantages**: Memory dynamic, insertion/deletion easy.
❌ **Limitations**: Backward traversal possible nahi.

🔷 2. Doubly Linked List (DLL)

👉 Har node me **3 parts** hote hain:

- prev (pichle node ka address)
- data
- next (agle node ka address)

Structure:

NULL ← [Prev | Data | Next] ↔ [Prev | Data | Next] ↔ [Prev | Data | NULL]

C Structure:

```
struct Node {
   int data;
   struct Node* prev;
   struct Node* next;
};
```

✅ **Advantages**: Forward + Backward traversal, deletion easy.
❌ **Limitations**: Extra memory (prev pointer).

🔷 3. Circular Linked List (CLL)

(a) Singly Circular Linked List

👉 Last node ka next phir se **head** ko point karta hai.

Mathematics:

Head → [Data | Next] → [Data | Next] → … → [Data | Head]

C Structure:

```
struct Node {
   int data;
   struct Node* prev;
   struct Node* next;
};
```


✅ **Advantages**: Har node se full traversal possible (cycle).
❌ **Limitations**: Implementation complex.

◈ Array vs Linked Representation

Feature	Array Representation	Linked Representation
Storage	Continuous memory block	Dynamic (node wise, scattered memory)
Size	Fixed (declare karte time decide hota hai)	Flexible (runtime me grow/shrink hoti hai)
Insertion/Deletion	Costly (shift elements karne padte hain)	Easy (sirf pointer update karna hota hai)
Traversal	By index (arr[i])	Sequential using pointers
Memory Usage	No extra pointer storage	Extra memory (next/prev pointers)
Cache Performance	Better (continuous storage)	Poorer (scattered memory)

◈ Example: Array vs Linked Representation

Array Representation (Singly)

C:

```
int arr[5] = {10, 20, 30, 40, 50};

// arr[0] → 10, arr[1] → 20 ...
```

Linked Representation (Singly):

```
struct Node {
   int data;
   struct Node* next;
};

// Head → [10|*] → [20|*] → [30|NULL]
```

✅ Summary

- **SLL** → Simple, one-way traversal.
- **DLL** → Two-way traversal, but more memory.
- **CLL** → Traversal cycle banata hai, kisi bhi node se start possible.
- **Array vs Linked** → Array fixed aur fast random access; Linked flexible aur insertion/deletion efficient.