VIDHYADEEP
UNIVERSITY
Holy Flame Of Knowledge

VIDHYADEEP UNIVERSITY INSTITUTE OF B.Sc. IT & BCA	
NAME :-	
SUBJECT :-	**ENROLLMENT :-**
SUBMIT DATE :-	**DEPARTMENT :-**

SR NO	PROBLEMS	DATE	SIGN
1	Stack implementation		
2	Operation of stack		
3	Operation of Queue		
4	Multiple stack		
5	Circular queue		
6	De-queue		
7	Implementation of Queue		

1) Stack implementation

```
#include <stdio.h>

#include <stdbool.h>

#define MAX 100  // Maximum size of the stack

// Stack structure

typedef struct {

    int items[MAX];

    int top;

} Stack;

// Function to initialize the stack

void initStack(Stack *s) {

    s->top = -1;  // Indicates an empty stack

}

// Function to check if the stack is empty

bool isEmpty(const Stack *s) {

    return s->top == -1;

}

// Function to check if the stack is full

bool isFull(const Stack *s) {

    return s->top == MAX - 1;

}

// Function to push an element onto the stack

void push(Stack *s, int value) {

    if (isFull(s)) {

        printf("Stack overflow\n");
        return;
    }
    s->items[++s->top] = value;
}
// Function to pop an element from the stack
int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack underflow\n");
        return -1;  // Return -1 to indicate an error
    }
    return s->items[s->top--];
}
// Function to get the top element of the stack
int peek(const Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty\n");
        return -1;  // Return -1 to indicate an error
    }
    return s->items[s->top];
}
// Main function
int main() {
    Stack myStack;
    initStack(&myStack);  // Initialize the stack
    // Push elements onto the stack
    push(&myStack, 10);
    push(&myStack, 20);
    push(&myStack, 30);

    // Print and pop elements from the stack
    while (!isEmpty(&myStack)) {
        printf("Top element: %d\n", peek(&myStack));
        pop(&myStack);
    }
    return 0;
}
```

2) Operation of stack

- **Push 2. Pop 3. Is Empty 4. Is Full**

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX 100  // Maximum size of the stack

// Stack structure
typedef struct {
    int items[MAX];
    int top;
} Stack;

// Function to initialize the stack
void initStack(Stack *s) {
    s->top = -1;  // Indicates an empty stack
}

// Function to check if the stack is empty
bool isEmpty(Stack *s) {
    return s->top == -1;
}

// Function to check if the stack is full
```

```
bool isFull(Stack *s) {
    return s->top == MAX - 1;
}

// Function to push an element onto the stack
void push(Stack *s, int value) {
    if (isFull(s)) {
        printf("Stack overflow\n");
        return;
    }
    s->items[++s->top] = value;
    printf("Pushed %d onto the stack\n", value);
}

// Function to pop an element from the stack
int pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack underflow\n");
        return -1;  // Return -1 to indicate error
    }
    return s->items[s->top--];
}

// Function to get the top element of the stack
int peek(Stack *s) {
    if (isEmpty(s)) {
        printf("Stack is empty\n");
        return -1;  // Return -1 to indicate error
    }
    return s->items[s->top];
}

// Function to get the size of the stack
int size(Stack *s) {
    return s->top + 1;
}

// Main function
int main() {
    Stack s;
    initStack(&s);  // Initialize the stack

    // Push elements onto the stack
    push(&s, 5);
    push(&s, 10);
    push(&s, 15);

    // Display the stack state before popping
    printf("\nStack before pop operations:\n");
    printf("Top element: %d\n", peek(&s));  // Should print 15
    printf("Stack size: %d\n", size(&s));  // Should print 3

    // Pop the top element from the stack
    printf("\nPopped element: %d\n", pop(&s));  // Should print 15

    // Display the stack state after one pop
    printf("\nStack after one pop operation:\n");
    if (!isEmpty(&s)) {
        printf("Top element: %d\n", peek(&s));  // Should print 10
    }
    printf("Stack size: %d\n", size(&s));  // Should print 2

    // Perform another pop operation
    printf("\nPopped element: %d\n", pop(&s));  // Should print 10

    // Display the state of the stack after another pop
    printf("\nStack after another pop operation:\n");
    if (!isEmpty(&s)) {
        printf("Top element: %d\n", peek(&s));  // Should print 5
    }
    printf("Stack size: %d\n", size(&s));  // Should print 1

    // Perform a final pop operation
    printf("\nPopped element: %d\n", pop(&s));  // Should print 5
```

```
    // Check if the stack is empty after the final pop
    printf("\nStack after final pop operation:\n");
    if (isEmpty(&s)) {
        printf("The stack is empty.\n");
    } else {
        printf("Top element: %d\n", peek(&s));
        printf("Stack size: %d\n", size(&s));
    }

    return 0;
}
```

3)Operation of Queue

1.Push 2. Pop 3. Is Empty 4. Is Full

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct {
    int *data;
    int front;
    int rear;
    int capacity;
    int count;
} CircularQueue;

// Function prototypes
CircularQueue* createQueue(int size);
bool enqueue(CircularQueue *queue, int value);
int dequeue(CircularQueue *queue);
bool isEmpty(CircularQueue *queue);
bool isFull(CircularQueue *queue);
void printQueue(CircularQueue *queue);
void freeQueue(CircularQueue *queue);

CircularQueue* createQueue(int size) {
    CircularQueue *queue = (CircularQueue *)malloc(sizeof(CircularQueue));
    queue->data = (int *)malloc(size * sizeof(int));

    queue->front = 0;

    queue->rear = -1;

    queue->capacity = size;

    queue->count = 0;

    return queue;

}


bool enqueue(CircularQueue *queue, int value) {

    if (isFull(queue)) {

        fprintf(stderr, "Queue is full\n");

        return false;

    }

    queue->rear = (queue->rear + 1) % queue->capacity;

    queue->data[queue->rear] = value;

    queue->count++;

    return true;

}


int dequeue(CircularQueue *queue) {

    if (isEmpty(queue)) {

        fprintf(stderr, "Queue is empty\n");

        return -1;

    }

    int value = queue->data[queue->front];
    queue->front = (queue->front + 1) % queue->capacity;

    queue->count--;

    return value;

}


bool isEmpty(CircularQueue *queue) {

    return queue->count == 0;

}


bool isFull(CircularQueue *queue) {

    return queue->count == queue->capacity;

}


void printQueue(CircularQueue *queue) {

    if (isEmpty(queue)) {

        printf("Queue is empty\n");

        return;

    }

    printf("Queue elements: ");

    for (int i = 0; i < queue->count; ++i) {

        printf("%d ", queue->data[(queue->front + i) % queue->capacity]);

    }

    printf("\n");

}
```

```
void freeQueue(CircularQueue *queue) {
    free(queue->data);
    free(queue);
}

int main() {
    CircularQueue *cq = createQueue(5); // Queue with capacity 5

    // Enqueue some values
    enqueue(cq, 10);
    enqueue(cq, 20);
    enqueue(cq, 30);
    enqueue(cq, 40);
    enqueue(cq, 50);

    // Print queue
    printQueue(cq);

    // Dequeue some values
    printf("Dequeued: %d\n", dequeue(cq));
    printf("Dequeued: %d\n", dequeue(cq));

    // Print queue after dequeuing
    printQueue(cq);
    // Enqueue more values

    enqueue(cq, 60);

    enqueue(cq, 70);



    // Print queue after enqueuing more values

    printQueue(cq);



    // Free the allocated memory

    freeQueue(cq);



    return 0;

}
```

4) Multiple stack

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>


#define MAX 100  // Maximum size of the queue


// Queue structure

typedef struct {

   int items[MAX];

   int front, rear, size;

} Queue;


// Function to initialize the queue

void initQueue(Queue *q) {

   q->front = 0;

   q->rear = -1;

   q->size = 0;

}


// Function to check if the queue is empty

bool isEmpty(Queue *q) {

   return q->size == 0;

}
```

```
// Function to check if the queue is full
bool isFull(Queue *q) {
    return q->size == MAX;
}

// Function to add an element to the queue
void enqueue(Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue is full\n");
        return;
    }
    q->rear = (q->rear + 1) % MAX;
    q->items[q->rear] = value;
    q->size++;
    printf("Enqueued %d\n", value);
}

// Function to remove an element from the queue
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return -1;  // Return -1 to indicate error
    }
    int value = q->items[q->front];
    q->front = (q->front + 1) % MAX;
    q->size--;

    return value;

}


// Function to get the front element of the queue

int front(Queue *q) {

    if (isEmpty(q)) {

        printf("Queue is empty\n");

        return -1;  // Return -1 to indicate error

    }

    return q->items[q->front];

}


// Function to get the size of the queue

int size(Queue *q) {

    return q->size;

}


// Main function

int main() {

    Queue q;

    initQueue(&q);  // Initialize the queue


    // Enqueue elements into the queue

    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);

    // Display the queue state
    printf("\nQueue state:\n");
    printf("Front element: %d\n", front(&q));  // Should print 10
    printf("Queue size: %d\n", size(&q));  // Should print 3

    // Dequeue an element from the queue
    printf("\nDequeued element: %d\n", dequeue(&q));  // Should print 10

    // Display the queue state after one dequeue
    printf("\nQueue state after one dequeue:\n");
    if (!isEmpty(&q)) {
        printf("Front element: %d\n", front(&q));  // Should print 20
    }
    printf("Queue size: %d\n", size(&q));  // Should print 2

    // Perform another dequeue operation
    printf("\nDequeued element: %d\n", dequeue(&q));  // Should print 20

    // Display the queue state after another dequeue
    printf("\nQueue state after another dequeue:\n");
    if (!isEmpty(&q)) {
        printf("Front element: %d\n", front(&q));  // Should print 30
```

```c
    }
    printf("Queue size: %d\n", size(&q));  // Should print 1

    // Perform a final dequeue operation
    printf("\nDequeued element: %d\n", dequeue(&q));  // Should print 30

    // Check if the queue is empty after the final dequeue
    printf("\nQueue state after final dequeue:\n");
    if (isEmpty(&q)) {
        printf("The queue is empty.\n");
    } else {
        printf("Front element: %d\n", front(&q));
        printf("Queue size: %d\n", size(&q));
    }

    return 0;
}
```

5) Circular queue

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct {
    int *data;
    int front;
    int rear;
    int capacity;
    int count;
} CircularQueue;

// Function prototypes
CircularQueue* createQueue(int size);
bool enqueue(CircularQueue *queue, int value);
int dequeue(CircularQueue *queue);
bool isEmpty(CircularQueue *queue);
bool isFull(CircularQueue *queue);
void printQueue(CircularQueue *queue);
void freeQueue(CircularQueue *queue);

CircularQueue* createQueue(int size) {
    CircularQueue *queue = (CircularQueue *)malloc(sizeof(CircularQueue));
    queue->data = (int *)malloc(size * sizeof(int));

    queue->front = 0;

    queue->rear = -1;

    queue->capacity = size;

    queue->count = 0;

    return queue;

}


bool enqueue(CircularQueue *queue, int value) {

    if (isFull(queue)) {

        fprintf(stderr, "Queue is full\n");

        return false;

    }

    queue->rear = (queue->rear + 1) % queue->capacity;

    queue->data[queue->rear] = value;

    queue->count++;

    return true;

}


int dequeue(CircularQueue *queue) {

    if (isEmpty(queue)) {

        fprintf(stderr, "Queue is empty\n");

        return -1;

    }
    int value = queue->data[queue->front];

    queue->front = (queue->front + 1) % queue->capacity;

    queue->count--;

    return value;

}


bool isEmpty(CircularQueue *queue) {

    return queue->count == 0;

}


bool isFull(CircularQueue *queue) {

    return queue->count == queue->capacity;

}


void printQueue(CircularQueue *queue) {

    if (isEmpty(queue)) {

        printf("Queue is empty\n");

        return;

    }

    printf("Queue elements: ");

    for (int i = 0; i < queue->count; ++i) {

        printf("%d ", queue->data[(queue->front + i) % queue->capacity]);

    }

    printf("\n");
```

```
}

void freeQueue(CircularQueue *queue) {
    free(queue->data);
    free(queue);
}

int main() {
    CircularQueue *cq = createQueue(5); // Queue with capacity 5

    // Enqueue some values
    enqueue(cq, 10);
    enqueue(cq, 20);
    enqueue(cq, 30);
    enqueue(cq, 40);
    enqueue(cq, 50);

    // Print queue
    printQueue(cq);

    // Dequeue some values
    printf("Dequeued: %d\n", dequeue(cq));
    printf("Dequeued: %d\n", dequeue(cq));
    // Print queue after dequeuing
    printQueue(cq);

    // Enqueue more values
    enqueue(cq, 60);
    enqueue(cq, 70);

    // Print queue after enqueuing more values
    printQueue(cq);

    // Free the allocated memory
    freeQueue(cq);

    return 0;
}
```

6) De queue

```
#include <stdio.h>

#include <stdlib.h>


typedef struct Deque {

    int* data;

    int front;

    int back;

    int size;

    int capacity;

} Deque;


Deque* createDeque(int initialCapacity) {

    Deque* dq = (Deque*)malloc(sizeof(Deque));

    dq->data = (int*)malloc(sizeof(int) * initialCapacity);

    dq->front = 0;

    dq->back = 0;

    dq->size = 0;

    dq->capacity = initialCapacity;

    return dq;

}


void resizeDeque(Deque* dq) {

    int newCapacity = dq->capacity * 2;
    int* newData = (int*)malloc(sizeof(int) * newCapacity);
    for (int i = 0; i < dq->size; i++) {
        newData[i] = dq->data[(dq->front + i) % dq->capacity];
    }
    free(dq->data);
    dq->data = newData;
    dq->front = 0;
    dq->back = dq->size;
    dq->capacity = newCapacity;
}

void pushBack(Deque* dq, int value) {
    if (dq->size == dq->capacity) {
        resizeDeque(dq);
    }
    dq->data[dq->back] = value;
    dq->back = (dq->back + 1) % dq->capacity;
    dq->size++;
}

void pushFront(Deque* dq, int value) {
    if (dq->size == dq->capacity) {
        resizeDeque(dq);
    }
    dq->front = (dq->front - 1 + dq->capacity) % dq->capacity;

    dq->data[dq->front] = value;

    dq->size++;

}


void popBack(Deque* dq) {

    if (dq->size > 0) {

        dq->back = (dq->back - 1 + dq->capacity) % dq->capacity;

        dq->size--;

    }

}


void popFront(Deque* dq) {

    if (dq->size > 0) {

        dq->front = (dq->front + 1) % dq->capacity;

        dq->size--;

    }

}


void printDeque(Deque* dq) {

    printf("Deque: ");

    for (int i = 0; i < dq->size; i++) {

        printf("%d ", dq->data[(dq->front + i) % dq->capacity]);

    }
    printf("\n");
}

void freeDeque(Deque* dq) {
    free(dq->data);
    free(dq);
}

int main() {
    Deque* dq = createDeque(4);

    // Adding elements to the back
    pushBack(dq, 10);
    pushBack(dq, 20);
    pushBack(dq, 30);

    // Adding elements to the front
    pushFront(dq, 0);
    pushFront(dq, -10);

    // Print the deque
    printf("Deque after additions: ");
    printDeque(dq);
    // Remove elements from the back
    popBack(dq);

    // Remove elements from the front
    popFront(dq);

    // Print the deque after removals
    printf("Deque after removals: ");
    printDeque(dq);

    // Add more elements to both ends
    pushBack(dq, 40);
    pushFront(dq, -20);

    // Print the final state of the deque
    printf("Final deque: ");
    printDeque(dq);

    // Free allocated memory
    freeDeque(dq);

    return 0;
}
```

7) Implementation of Queue

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>


#define MAX 100  // Maximum size of the queue


// Queue structure

typedef struct {

    int items[MAX];

    int front, rear, size;

} Queue;


// Function to initialize the queue

void initQueue(Queue *q) {

    q->front = 0;

    q->rear = -1;

    q->size = 0;

}


// Function to check if the queue is empty

bool isEmpty(Queue *q) {

    return q->size == 0;

}
```

```
// Function to check if the queue is full
bool isFull(Queue *q) {
    return q->size == MAX;
}

// Function to add an element to the queue
void enqueue(Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue is full\n");
        return;
    }
    q->rear = (q->rear + 1) % MAX;
    q->items[q->rear] = value;
    q->size++;
    printf("Enqueued %d\n", value);
}

// Function to remove an element from the queue
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return -1;  // Return -1 to indicate error
    }
    int value = q->items[q->front];
    q->front = (q->front + 1) % MAX;
    q->size--;
    return value;
}

// Function to get the front element of the queue
int front(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return -1;  // Return -1 to indicate error
    }
    return q->items[q->front];
}

// Function to get the size of the queue
int size(Queue *q) {
    return q->size;
}

// Main function
int main() {
    Queue q;
    initQueue(&q);  // Initialize the queue
```

```
// Enqueue elements into the queue
enqueue(&q, 10);
enqueue(&q, 20);
enqueue(&q, 30);

// Display the queue state
printf("\nQueue state:\n");
printf("Front element: %d\n", front(&q));  // Should print 10
printf("Queue size: %d\n", size(&q));  // Should print 3

// Dequeue an element from the queue
printf("\nDequeued element: %d\n", dequeue(&q));  // Should print 10

// Display the queue state after one dequeue
printf("\nQueue state after one dequeue:\n");
if (!isEmpty(&q)) {
   printf("Front element: %d\n", front(&q));  // Should print 20
}
printf("Queue size: %d\n", size(&q));  // Should print 2

// Perform another dequeue operation
printf("\nDequeued element: %d\n", dequeue(&q));  // Should print 20
```

```
    // Display the queue state after another dequeue
    printf("\nQueue state after another dequeue:\n");
    if (!isEmpty(&q)) {
        printf("Front element: %d\n", front(&q));  // Should print 30
    }
    printf("Queue size: %d\n", size(&q));  // Should print 1

    // Perform a final dequeue operation
    printf("\nDequeued element: %d\n", dequeue(&q));  // Should print 30

    // Check if the queue is empty after the final dequeue
    printf("\nQueue state after final dequeue:\n");
    if (isEmpty(&q)) {
        printf("The queue is empty.\n");
    } else {
        printf("Front element: %d\n", front(&q));
        printf("Queue size: %d\n", size(&q));
    }

    return 0;
}
```