Digital Assessment – 1

Name : Ashmit Sharma

Registration Number: 24BBS0041

CODES:

Q1

```
#include <stdio.h>
#define MAX 5

int stack[MAX];
int top = -1;

void push(int element) {
   if (top == MAX - 1) {
      printf("Stack Overflow! Cannot push %d onto the stack.\n", element);
   } else {
      stack[++top] = element;
      printf("Pushed %d onto the stack.\n", element);
   }
}

void pop() {
   if (top == -1) {
      printf("Stack Underflow! Cannot pop from an empty stack.\n");
   } else {
      printf("Popped %d from the stack.\n", stack[top--]);
   }
}

void display() {
```

```
    if (top == -1) {
        printf("The stack is empty.\n");
    } else {
        printf("Stack elements: ");
        for (int i = top; i >= 0; i--) {
            printf("%d ", stack[i]);
        }
        printf("\n");
    }
}

int main() {
    int choice, element;

    do {
        printf("\nStack Operations:\n");
        printf("1. PUSH\n");
        printf("2. POP\n");
        printf("3. DISPLAY\n");
        printf("4. EXIT\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the element to push: ");
                scanf("%d", &element);
                push(element);
                break;

            case 2:
                pop();
                break;
```

```
            case 3:
                display();
                break;

            case 4:
                printf("Exiting program.\n");
                break;

            default:
                printf("Invalid choice! Please try again.\n");
        }
    } while (choice != 4);

    return 0;
}
```

Q2:

```
#include <stdio.h>
#define MAX 5

int queue[MAX];
int front = -1, rear = -1;

void enqueue(int element) {
    if ((rear + 1) % MAX == front) {
        printf("Queue Overflow! Cannot enqueue %d.\n", element);
    } else {
        if (front == -1) {
            front = 0;
        }
        rear = (rear + 1) % MAX;
```

```
        queue[rear] = element;
        printf("Enqueued %d into the queue.\n", element);
    }
}

void dequeue() {
    if (front == -1) {
        printf("Queue Underflow! Cannot dequeue from an empty queue.\n");
    } else {
        printf("Dequeued %d from the queue.\n", queue[front]);
        if (front == rear) {
            front = rear = -1;
        } else {
            front = (front + 1) % MAX;
        }
    }
}

void display() {
    if (front == -1) {
        printf("The queue is empty.\n");
    } else {
        printf("Queue elements: ");
        for (int i = front; i != rear; i = (i + 1) % MAX) {
            printf("%d ", queue[i]);
        }
        printf("%d\n", queue[rear]);
    }
}

int main() {
    int choice, element;
```

```
do {
    printf("\nQueue Operations:\n");
    printf("1. ENQUEUE\n");
    printf("2. DEQUEUE\n");
    printf("3. DISPLAY\n");
    printf("4. EXIT\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Enter the element to enqueue: ");
            scanf("%d", &element);
            enqueue(element);
            break;

        case 2:
            dequeue();
            break;

        case 3:
            display();
            break;

        case 4:
            printf("Exiting program.\n");
            break;

        default:
            printf("Invalid choice! Please try again.\n");
    }
} while (choice != 4);
```

```
    return 0;
}
```

Q3:

```
#include <stdio.h>
#define SIZE 5

int circularQueue[SIZE];
int front = -1, rear = -1;

int isFull() {
    return (front == 0 && rear == SIZE - 1) || (front == rear + 1);
}

int isEmpty() {
    return front == -1;
}

void enqueue(int value) {
    if (isFull()) {
        printf("\nQueue is Full. Cannot enqueue %d.\n", value);
        return;
    }

    if (isEmpty()) {
        front = rear = 0;
    } else {
        rear = (rear + 1) % SIZE;
    }

    circularQueue[rear] = value;
    printf("\nEnqueued: %d\n", value);
}
```

```
void dequeue() {
    if (isEmpty()) {
        printf("\nQueue is Empty. Cannot dequeue.\n");
        return;
    }

    int value = circularQueue[front];

    if (front == rear) {
        front = rear = -1; // Queue becomes empty
    } else {
        front = (front + 1) % SIZE;
    }

    printf("\nDequeued: %d\n", value);
}

void display() {
    if (isEmpty()) {
        printf("\nQueue is Empty.\n");
        return;
    }

    printf("\nCircular Queue: ");
    int i = front;
    while (1) {
        printf("%d ", circularQueue[i]);
        if (i == rear) {
            break;
        }
        i = (i + 1) % SIZE;
    }
```

```
    printf("\n");
}

int main() {
    int choice, value;

    do {
        printf("\nCircular Queue Operations:\n");
        printf("1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the value to enqueue: ");
                scanf("%d", &value);
                enqueue(value);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("\nExiting...\n");
                break;
            default:
                printf("\nInvalid choice! Please try again.\n");
        }
    } while (choice != 4);

    return 0;
```

```
}
```

Q4:

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

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

struct Node* head = NULL;

void insertAtBeginning(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = head;
   head = newNode;
   printf("Inserted %d at the beginning.\n", data);
}

void insertAtEnd(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;

   if (head == NULL) {
      head = newNode;
   } else {
      struct Node* temp = head;
      while (temp->next != NULL) {
         temp = temp->next;
      }
```

```
        temp->next = newNode;
    }
    printf("Inserted %d at the end.\n", data);
}

void insertAtPosition(int data, int position) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;

    if (position == 1) {
        newNode->next = head;
        head = newNode;
        printf("Inserted %d at position %d.\n", data, position);
        return;
    }

    struct Node* temp = head;
    for (int i = 1; i < position - 1; i++) {
        if (temp == NULL) {
            printf("Position out of bounds.\n");
            free(newNode);
            return;
        }
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Position out of bounds.\n");
        free(newNode);
        return;
    }

    newNode->next = temp->next;
```

```
    temp->next = newNode;
    printf("Inserted %d at position %d.\n", data, position);
}

void deleteFromBeginning() {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }

    struct Node* temp = head;
    head = head->next;
    printf("Deleted %d from the beginning.\n", temp->data);
    free(temp);
}

void deleteFromEnd() {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }

    if (head->next == NULL) {
        printf("Deleted %d from the end.\n", head->data);
        free(head);
        head = NULL;
        return;
    }

    struct Node* temp = head;
    while (temp->next->next != NULL) {
        temp = temp->next;
    }
```

```
    printf("Deleted %d from the end.\n", temp->next->data);
    free(temp->next);
    temp->next = NULL;
}

void deleteFromPosition(int position) {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }

    if (position == 1) {
        struct Node* temp = head;
        head = head->next;
        printf("Deleted %d from position %d.\n", temp->data, position);
        free(temp);
        return;
    }

    struct Node* temp = head;
    for (int i = 1; i < position - 1; i++) {
        if (temp == NULL || temp->next == NULL) {
            printf("Position out of bounds.\n");
            return;
        }
        temp = temp->next;
    }

    if (temp->next == NULL) {
        printf("Position out of bounds.\n");
        return;
    }
```

```
    struct Node* toDelete = temp->next;
    temp->next = toDelete->next;
    printf("Deleted %d from position %d.\n", toDelete->data, position);
    free(toDelete);
}

void search(int data) {
    struct Node* temp = head;
    int position = 1;

    while (temp != NULL) {
        if (temp->data == data) {
            printf("Element %d found at position %d.\n", data, position);
            return;
        }
        temp = temp->next;
        position++;
    }

    printf("Element %d not found in the list.\n", data);
}

void display() {
    if (head == NULL) {
        printf("The list is empty.\n");
        return;
    }

    printf("List elements: ");
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
```

```
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    int choice, data, position;

    do {
        printf("\nSingly Linked List Operations:\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Insert at Position\n");
        printf("4. Delete from Beginning\n");
        printf("5. Delete from End\n");
        printf("6. Delete from Position\n");
        printf("7. Search\n");
        printf("8. Display\n");
        printf("9. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the data to insert: ");
                scanf("%d", &data);
                insertAtBeginning(data);
                break;

            case 2:
                printf("Enter the data to insert: ");
                scanf("%d", &data);
                insertAtEnd(data);
```

```
            break;

        case 3:
            printf("Enter the data to insert: ");
            scanf("%d", &data);
            printf("Enter the position to insert: ");
            scanf("%d", &position);
            insertAtPosition(data, position);
            break;

        case 4:
            deleteFromBeginning();
            break;

        case 5:
            deleteFromEnd();
            break;

        case 6:
            printf("Enter the position to delete: ");
            scanf("%d", &position);
            deleteFromPosition(position);
            break;

        case 7:
            printf("Enter the element to search: ");
            scanf("%d", &data);
            search(data);
            break;

        case 8:
            display();
            break;
```

```
        case 9:
            printf("Exiting program.\n");
            break;

        default:
            printf("Invalid choice! Please try again.\n");
    }
  } while (choice != 9);

  return 0;
}
```

Q5:

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

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

struct Node *head = NULL;

// Function to create a new node
struct Node* createNode(int value) {
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  newNode->prev = NULL;
  newNode->next = NULL;
  return newNode;
```

```
}

void insertAtBeginning(int value) {
    struct Node* newNode = createNode(value);
    if (head == NULL) {
        head = newNode;
    } else {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
    printf("Inserted %d at the beginning.\n", value);
}

void insertAtEnd(int value) {
    struct Node* newNode = createNode(value);
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
    printf("Inserted %d at the end.\n", value);
}

void insertAtPosition(int value, int position) {
    struct Node* newNode = createNode(value);
    if (position == 1) {
        insertAtBeginning(value);
```

```
        return;
    }

    struct Node* temp = head;
    for (int i = 1; i < position - 1; i++) {
        if (temp == NULL) {
            printf("Position out of bounds.\n");
            return;
        }
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Position out of bounds.\n");
        return;
    }

    newNode->next = temp->next;
    if (temp->next != NULL) {
        temp->next->prev = newNode;
    }
    temp->next = newNode;
    newNode->prev = temp;

    printf("Inserted %d at position %d.\n", value, position);
}

void deleteAtBeginning() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
```

```
    struct Node* temp = head;
    head = head->next;
    if (head != NULL) {
        head->prev = NULL;
    }
    printf("Deleted %d from the beginning.\n", temp->data);
    free(temp);
}

void deleteAtEnd() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }

    if (temp->prev != NULL) {
        temp->prev->next = NULL;
    } else {
        head = NULL;
    }

    printf("Deleted %d from the end.\n", temp->data);
    free(temp);
}

void deleteAtPosition(int position) {
    if (head == NULL) {
        printf("List is empty.\n");
```

```
        return;
    }

    if (position == 1) {
        deleteAtBeginning();
        return;
    }

    struct Node* temp = head;
    for (int i = 1; i < position; i++) {
        if (temp == NULL) {
            printf("Position out of bounds.\n");
            return;
        }
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Position out of bounds.\n");
        return;
    }

    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    }

    printf("Deleted %d from position %d.\n", temp->data, position);
    free(temp);
}
```

```
void search(int value) {
    struct Node* temp = head;
    int position = 1;
    while (temp != NULL) {
        if (temp->data == value) {
            printf("Element %d found at position %d.\n", value, position);
            return;
        }
        temp = temp->next;
        position++;
    }
    printf("Element %d not found in the list.\n", value);
}

void display() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    printf("Doubly Linked List: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    int choice, value, position;

    do {
```

```
printf("\nDoubly Linked List Operations:\n");
printf("1. Insert at Beginning\n");
printf("2. Insert at End\n");
printf("3. Insert at Position\n");
printf("4. Delete at Beginning\n");
printf("5. Delete at End\n");
printf("6. Delete at Position\n");
printf("7. Search\n");
printf("8. Display\n");
printf("9. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
  case 1:
    printf("Enter value to insert at beginning: ");
    scanf("%d", &value);
    insertAtBeginning(value);
    break;
  case 2:
    printf("Enter value to insert at end: ");
    scanf("%d", &value);
    insertAtEnd(value);
    break;
  case 3:
    printf("Enter value to insert: ");
    scanf("%d", &value);
    printf("Enter position: ");
    scanf("%d", &position);
    insertAtPosition(value, position);
    break;
  case 4:
    deleteAtBeginning();
```

```
            break;
        case 5:
            deleteAtEnd();
            break;
        case 6:
            printf("Enter position to delete: ");
            scanf("%d", &position);
            deleteAtPosition(position);
            break;
        case 7:
            printf("Enter value to search: ");
            scanf("%d", &value);
            search(value);
            break;
        case 8:
            display();
            break;
        case 9:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice! Please try again.\n");
      }
   } while (choice != 9);

   return 0;
}
```

DSA

Digital Assessment – 1

NAME :- Ashmit Sharma
Reg no. :- 24BBS0041

Q1

Algorithm :

1. Start
2. Initialize an empty stack and a variable top = -1 to indicate the top of the stack.
3. Display Menu options:
   (a) PUSH an element into the stack
   (b) POP an element from the stack
   (c) Display all elements of the stack
   (d) Exit the program
4. Loop until user selects exit:
   a. Prompt the user to enter their choice.
   b. Based on the choice, perform the following actions:

   Case 1: PUSH()
   - Check if the stack is full (i.e top == size - 1).
   - If full, display "Stack Overflow" and return to the menu.
   - Otherwise, prompt the user to enter an element

- Increment top by 1 and add the element to stack [top]

Case 2: POP()

- Check if the stack is empty (i.e. top == -1)
- If empty, display "Stack Underflow" and return to the menu.
- Otherwise, display the element at stack[top] and decrement top by 1.

Case 3: Display()

- Check if the stack is empty (i.e, top == -1).
- If empty, display "Stack is empty"
- Otherwise, display all elements from stack [top] to stack [0].

Case 4: Exit

- Terminate the program

5. End of Loop
6. Stop.

```
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter the element to push: 10
Pushed 10 onto the stack.

Stack Operations:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 2
Popped 10 from the stack.

Stack Operations:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter the element to push: 20
Pushed 20 onto the stack.

Stack Operations:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 3
Stack elements: 20

Stack Operations:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice:
```

```
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter the element to push: 20
Pushed 20 onto the stack.

Stack Operations:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter the element to push: 25
Pushed 25 onto the stack.

Stack Operations:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter the element to push: 30
Stack Overflow! Cannot push 30 onto the stack.

Stack Operations:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 3
Stack elements: 25 20 15 10 5

Stack Operations:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice:
```

```
Stack Operations:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 2
Stack Underflow! Cannot pop from an empty stack.

Stack Operations:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 3
The stack is empty.

Stack Operations:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter the element to push: 50
Pushed 50 onto the stack.

Stack Operations:
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 4
Exiting program.
PS C:\Users\Hp\.vscode>
```

Q2

Algorithms

1. Start
2. Initialize
   (a) An array queue[] of fixed size
   (b) front = -1 and rear = -1 to track the front and rear of the queue.
3. Display Menu:
   (a) Enqueue
   (b) Dequeue
   (c) Display
   (d) Exit
4. Take the user's choice as input
5. Perform operations based on the user's choice:
   (a) Enqueue():
   - Check if the queue is full (rear == SIZE-1):
   - If true, display "Queue Overflow" and return to the menu.
   - Otherwise:
     - If the queue is empty (front == -1), set front=0.
     - Increment rear by 1 and insert the element at queue[rear].

⑤ Dequeue():

- Check if the queue is empty (front == -1 or front > rear):
  - If true, display "Queue Underflow" and return to the menu.
  - otherwise:
    - Display the element at queue[front].
    - Increment front by 1
    - If front > rear, reset front = -1 (queue is empty),

⑥ Display()

- check if the queue is empty (front == -1 or front > rear):
  - If true, display "Queue is empty."
  - Otherwise:
    - Traverse and print elements from queue[front] to queue[rear].

④ Exit

- exit the program.

6. End

```
Queue Operations:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter the element to enqueue: 10
Enqueued 10 into the queue.

Queue Operations:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter the element to enqueue: 20
Enqueued 20 into the queue.

Queue Operations:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice: 2
Dequeued 10 from the queue.

Queue Operations:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice: 3
Queue elements: 20

Queue Operations:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
```

```
Enter your choice: 1
Enter the element to enqueue: 3
Enqueued 3 into the queue.

Queue Operations:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter the element to enqueue: 4
Enqueued 4 into the queue.

Queue Operations:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter the element to enqueue: 5
Enqueued 5 into the queue.

Queue Operations:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter the element to enqueue: 6
Queue Overflow! Cannot enqueue 6.

Queue Operations:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice: 3
Queue elements: 1 2 3 4 5
```

```
Queue Operations:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice: 2
Queue Underflow! Cannot dequeue from an empty queue.

Queue Operations:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice: 3
The queue is empty.

Queue Operations:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter the element to enqueue: 30
Enqueued 30 into the queue.

Queue Operations:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice: 3
Queue elements: 30

Queue Operations:
1. ENQUEUE
2. DEQUEUE
3. DISPLAY
4. EXIT
Enter your choice:
```

Q3

Algorithm

1. Start
2. Initialize:
   - queue[] as an array of fixed size SIZE.
   - front = -1 and rear = -1 to track the front and rear of the queue
3. Display menu:
   - (a) Enqueue
   - (b) Dequeue
   - (c) Display
   - (d) Exit
4. Repeat until the user chooses to exit:
   - Take the user's choice as input
5. Perform operations based on the user's choice:
   - (a) Enqueue():
     - Check if the queue is full:
       - Condition (rear + 1) % SIZE == front
       - If true, display "Queue Overflow" and return to the menu
       - Otherwise:

- If the queue is empty (front == -1), set front = 0.
- Increased rear using the formula rear = (rear+1) % SIZE.
- Insert the element at queue[rear].

(b) Dequeue():

- check if the queue is empty:
  - Condition : front == -1
  - If true, display "Queue Underflow" and return to the menu.
  - otherwise
    - Display the element at queue [front].
    - If front == rear, reset front = -1 and rear = -1 (queue become empty)
    - Otherwise, increment front using the formula front = (front+1) % SIZE

(c) Display()

- Check if the queue is empty:
  - Condition : front == -1
  - If true, display "Queue is empty"

- ⓒ Otherwise:
  - Traverse the queue from front to rear:
    - Use a loop with index i = front and iterate until i != rear:
      - Print queue[i]
      - Use the formula i = (i+1) % SIZE to move circularly.

ⓓ Exit:
- End the program.

6. End.

```
Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 10

Enqueued: 10

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 20

Enqueued: 20

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3

Circular Queue: 10 20

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 4

Exiting...
PS C:\Users\Hp\.vscode>
```

```
Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 50

Enqueued: 50

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 60

Queue is Full. Cannot enqueue 60.

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
Enter the value to enqueue: 70

Queue is Full. Cannot enqueue 70.

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 3

Circular Queue: 10 20 30 40 50
```

TERMINAL

```
PS C:\Users\Hp\.vscode> gcc c.c
c:/mingw/bin/../lib/gcc/mingw32/6.3.0/../../../../mingw32/bin/ld.exe: cannot open output file a.exe: Permission denied
collect2.exe: error: ld returned 1 exit status
PS C:\Users\Hp\.vscode> ./a.exe

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2

Queue is Empty. Cannot dequeue.

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 2

Queue is Empty. Cannot dequeue.

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice:
```

Q4

Algorithm

1. Start
2. Define a structure Node with:
   - A data field to store the value
   - A pointer field to point to the next node (next).
3. Initialize:
   - head = NULL (pointer to the start of the list)
4. Display menu:

(a) Insertion
- Beginning
- End
- At a given position

(b) Deletion
- Beginning
- End
- At a given position

(c) Search

(d) Exit

Repeat until the user chooses to exit!

- Take the user's choice as input

Operations:

(a) Insertion:

- At the Beginning:
  - ★ Create a new node.
  - ★ Set the new node's next pointer to head.
  - ★ Update head to point to the new node.
- At the End:
  - ★ Create a new node.
  - ★ If the list is empty (head == NULL), set head to the new node.
  - ★ Otherwise, traverse to the last node (temp->next == NULL)
  - ★ Set the last node's next pointer to the new node.
- At a Given Position:
  - ★ Take the position as input.
  - ★ If the position is 1, perform insertion at the beginning
  - ★ Otherwise, traverse to the (position-1)th node
  - ★ Create a new node and update the links:
    - □ Set the new node's next to the current node's next
    - □ Update the current node's next to the new node.

⑥ Deletion:

- From the Beginning:

  ★ If the list is empty (head == NULL), display "List is empty".

  ★ Otherwise, set head to head -> next and free the previous head.

- From the End:

  ★ If the list is empty (head == NULL), display "List is empty."

  ★ If the list has only one node, set head = NULL and free the node

  ★ Otherwise, traverse to the second-last node (temp -> next -> next == NULL)

  ★ Set temp -> next = NULL and free the last node.

- At a Given position

  ★ Take the position as input

  ★ If the position is 1, perform deletion at the beginning

  ★ Otherwise, traverse to the (position-1)th node

  ★ Update its next pointer to skip the target node:

    . temp -> next = temp -> next -> next.

  ★ Free the target node.

© Search:

- Take the element to search as input.
- Traverse the list while comparing each node's data with the given element.
- If found, display the position of the element.
- If not found, display "Element not found".

(d) Exit

6. Exit

(a) Terminate the program.

```
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter your choice: 1
Enter the data to insert: 10
Inserted 10 at the beginning.

Singly Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter your choice: 1
Enter the data to insert: 20
Inserted 20 at the beginning.

Singly Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter your choice: 3
Enter the data to insert: 15
Enter the position to insert: 2
Inserted 15 at position 2.
```

```
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter your choice: 1
Enter the data to insert: 10
Inserted 10 at the beginning.

Singly Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter your choice: 2
Enter the data to insert: 15
Inserted 15 at the end.

Singly Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter your choice: 7
Enter the element to search: 15
Element 15 found at position 2.
```

```
Singly Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter your choice: 4
List is empty. Cannot delete.

Singly Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
Enter your choice: 3
Enter the data to insert: 30
Enter the position to insert: 2
Position out of bounds.

Singly Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete from Beginning
5. Delete from End
6. Delete from Position
7. Search
8. Display
9. Exit
```

Q5

Algorithm:

1. Start
2. Define a structure Node with:
   - A data field to store the value
   - A prev pointer to point to the previous node
   - A next pointer to point to the next node
3. Initialize
   - head = NULL
4. Display menu:
   (a) Insertion
      - Beginning
      - End
      - At a given position
   (b) Deletion
      - Beginning
      - End
      - At a given position
   (c) Search
   (d) Exit

5. Operations

(A) Insertion

(a) At the Beginning:

* Create a node (new).
* Set the new node's next pointer to head.
* If the list is not empty (head != NULL), set head->prev to the new node
* Update head to point to the new node

(b) At the End:

* Create a new node
* If the list is empty (head == NULL), set head to the new node.
* Otherwise, traverse to the last node (temp->next == NULL).
* Set the last node's next pointer to the new node and set the new node's prev to the last node

(c) At a Given Position

* Take the position as input
* If the position is 1, perform insertion at the beginning.
* Otherwise traverse to (position-1)th node
* Create a new node and update the links
  - Set new node's next to current node's next.

- Set the current node's next's prev to the new node
- Update the current node's next's ~~prev~~ to the new node
- Set the new node's prev to the current node

(b)

(a) From the Beginning

* If the list is empty (head == NULL), display "List is empty."
* Otherwise:
  - Set head to head -> next
  - If head != NULL, set head -> prev = NULL.
  - Free the previous head

(b) From the End:

* If the list is empty (head == NULL), display "List is Empty"
* If the list has only one node (head -> next == NULL) set head = NULL and free the node
* Otherwise, traverse to the second-last node (temp -> next -> next == NULL).
* Set temp -> next -> NULL and free the last node

(c) At a Given Position:

* Take the position as input
* If the position is 1, perform deletion at the beginning
* otherwise:
  - Traverse to the (position − 1)th node
  - Update the links to skip the target node:
    - temp → next = temp → next → next.
    - If temp → next != NULL, set temp → next → prev = temp.
    - Free the target node

(C) Search:

- Take the element to search as input
- Traverse the list starting from head:
  * Compare each node's data with the given element
  * If found, display the position of the element
  * If not found, display "Element not found".

(D) Exit:

- Terminate the program.

6. End.

```
3. Insert at Position
4. Delete at Beginning
5. Delete at End
6. Delete at Position
7. Search
8. Display
9. Exit
Enter your choice: 2
Enter value to insert at end: 20
Inserted 20 at the end.

Doubly Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete at Beginning
5. Delete at End
6. Delete at Position
7. Search
8. Display
9. Exit
Enter your choice: 3
Enter value to insert: 15
Enter position: 2
Inserted 15 at position 2.

Doubly Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete at Beginning
5. Delete at End
6. Delete at Position
7. Search
8. Display
9. Exit
Enter your choice: 8
Doubly Linked List: 10 15 20
```

```
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete at Beginning
5. Delete at End
6. Delete at Position
7. Search
8. Display
9. Exit
Enter your choice: 2
Enter value to insert at end: 20
Inserted 20 at the end.

Doubly Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete at Beginning
5. Delete at End
6. Delete at Position
7. Search
8. Display
9. Exit
Enter your choice: 4
Deleted 10 from the beginning.

Doubly Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete at Beginning
5. Delete at End
6. Delete at Position
7. Search
8. Display
9. Exit
Enter your choice: 5
Deleted 20 from the end.
```

TERMINAL

```
3. Insert at Position
4. Delete at Beginning
5. Delete at End
6. Delete at Position
7. Search
8. Display
9. Exit
Enter your choice: 2
Enter value to insert at end: 15
Inserted 15 at the end.

Doubly Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete at Beginning
5. Delete at End
6. Delete at Position
7. Search
8. Display
9. Exit
Enter your choice: 7
Enter value to search: 15
Element 15 found at position 3.

Doubly Linked List Operations:
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete at Beginning
5. Delete at End
6. Delete at Position
7. Search
8. Display
9. Exit
Enter your choice: 7
Enter value to search: 25
Element 25 not found in the list.
```