EXPERIMENT – 7 QUEUES:

**Description:**

A queue is a fundamental data structure in computer science and programming that operates on the principle of "first-in, first-out" (FIFO). It can be visualized as a linear collection of elements, much like people waiting in a line. Elements are added to the back of the queue, and they are removed from the front of the queue, ensuring that the item that has been waiting the longest is the first to be processed.

**ADVANTAGES:**

1. FIFO Order: The primary advantage of a queue is that it enforces the FIFO order, ensuring that items are processed in the order they were added. This property is essential in many real-world scenarios, such as task scheduling and print job management.

2. Simple and Efficient: Queue operations (enqueue and dequeue) are simple and can be implemented efficiently. Both insertion and removal of elements from the queue typically have constant time complexity, O(1), when using appropriate data structures.

3. Synchronization: Queues are useful for synchronization between different parts of a program or in multi-threaded environments. They help control access to shared resources in a thread-safe manner.

4Breadth-First Search (BFS) Queues are a key component in graph algorithms like BFS, where you need to explore nodes layer by layer.

5.\*Buffering and Queuing: Queues are commonly used in systems to buffer data or events. For example, message queuing systems ensure that messages are processed in the order they are received.

6. Task Management: In task management systems, queues can be used to manage a list of tasks, ensuring that tasks are executed in a well-defined order.

**Disadvantages**

1. \*\*Limited Functionality\*\*: Queues are specialized for maintaining elements in a specific order. They may not be the best choice for applications that require more complex operations, such as searching for elements by key or priority.

2. Fixed Size (in some implementations): Some queue implementations, like arrays, have a fixed size. Once the queue is full, you cannot add more elements without resizing the data structure, which can be an inefficient operation.

3. Wasted Memory: In a linked list implementation, each element in the queue typically contains a reference to the next element, which can lead to some memory overhead. This overhead can be significant when dealing with a large number of elements.

4. No Random Access: Unlike arrays, you cannot directly access elements in a queue by index. You must dequeue elements one by one from the front to access specific elements, which can be inefficient if you need to access elements at arbitrary positions.

5.Complexity in Priority Queues: While priority queues are a variant of queues, they can be more complex to implement and maintain, especially if the priority order changes frequently.

6. Not Ideal for LIFO Operations: If you need to perform last-in, first-out (LIFO) operations, a stack data structure would be more suitable than a queue.

**APPLICATIONS:**

1. Task Scheduling

2. Print Job Management

3. Breadth-First Search (BFS)

4. Message Queues

5. Task Management Systems

6. Call Center Systems

7. Print Queue Management

8. Buffering and Caching

9. Web Servers

10. Task Processing Pipelines

11. Resource Sharing and Synchronization

12. Simulation and Modeling

13. Print Servers

14. Order Processing Systems

15. Call Routing and Voicemail Systems

**OPERATIONS ON QUEUES USING ARRAYS:**

1)Enqueue:

Function enqueue(data):

if rear == max\_size - 1:

print "Queue is full. Cannot insert."

return

if front == -1:

front = 0

increment rear

queue[rear] = data

print "Inserted: data"

2)Dequeue:

Function dequeue():

if front == -1:

print "Queue is empty. Cannot delete."

return

deletedItem = queue[front]

if front == rear:

front = rear = -1

else:

increment front

print "Deleted: deletedItem"

3)Size:

Function size():

if front == -1:

return 0

return rear - front + 1

4)Display:

Function display():

if front == -1:

print "Queue is empty."

return

for i from front to rear:

print queue[i]

5)Front:

Function getFront():

if front == -1:

print "Queue is empty."

return -1

return queue[front]

6)Rear:

Function getRear():

if rear == -1:

print "Queue is empty."

return -1

return queue[rear]

7)Empty:

Function isEmpty():

if front == -1:

return true

else:

return false

8)Full:

Function isFull():

if rear == max\_size - 1:

return true

else:

return false

**Program to perform queues using arrays :**

#include <stdio.h>

#include <stdlib.h>

int \*queue;

int front = -1;

int rear = -1;

int max\_size;

// Function to insert an element into the queue

void enqueue(int data) {

if (rear == max\_size - 1) {

printf("Queue is full. Cannot insert.\n");

return;

}

if (front == -1) {

front = 0;

}

rear++;

queue[rear] = data;

printf("Inserted: %d\n", data);

}

// Function to delete an element from the queue

void dequeue() {

if (front == -1) {

printf("Queue is empty. Cannot delete.\n");

return;

}

int deletedItem = queue[front];

if (front == rear) {

front = rear = -1;

} else {

front++;

}

printf("Deleted: %d\n", deletedItem);

}

// Function to get the size of the queue

int size() {

if (front == -1) {

return 0;

}

return rear - front + 1;

}

// Function to display the elements of the queue

void display() {

if (front == -1) {

printf("Queue is empty.\n");

return;

}

printf("Queue elements: ");

for (int i = front; i <= rear; i++) {

printf("%d ", queue[i]);

}

printf("\n");

}

// Function to get the front element of the queue

int getFront() {

if (front == -1) {

printf("Queue is empty.\n");

return -1;

}

return queue[front];

}

// Function to get the rear element of the queue

int getRear() {

if (rear == -1) {

printf("Queue is empty.\n");

return -1;

}

return queue[rear];

}

// Function to check if the queue is empty

int isEmpty() {

return (front == -1);

}

// Function to check if the queue is full

int isFull() {

return (rear == max\_size - 1);

}

int main() {

int choice, data;

printf("Enter the maximum size of the queue: ");

scanf("%d", &max\_size);

// Dynamically allocate memory for the queue

queue = (int \*)malloc(max\_size \* sizeof(int));

while (1) {

printf("\nQueue Operations:\n");

printf("1. INSERT\n");

printf("2. DELETE\n");

printf("3. SIZE\n");

printf("4. Display\n");

printf("5. FRONT\n");

printf("6. REAR\n");

printf("7. IsEmpty\n");

printf("8. IsFull\n");

printf("9. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter data to insert: ");

scanf("%d", &data);

enqueue(data);

break;

case 2:

dequeue();

break;

case 3:

printf("Queue size: %d\n", size());

break;

case 4:

display();

break;

case 5:

printf("Front element: %d\n", getFront());

break;

case 6:

printf("Rear element: %d\n", getRear());

break;

case 7:

if (isEmpty()) {

printf("Queue is empty.\n");

} else {

printf("Queue is not empty.\n");

}

break;

case 8:

if (isFull()) {

printf("Queue is full.\n");

} else {

printf("Queue is not full.\n");

}

break;

case 9:

printf("Exiting program.\n");

free(queue);//Free the dynamically memory

return 0;

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

**OUTPUT:**

Enter the maximum size of the queue: 4

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 1

Enter data to insert: 29

Inserted: 29

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 2

Deleted: 29

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 3

Queue size: 0

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 4

Queue is empty.

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 5

Queue is empty.

Front element: -1

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 6

Queue is empty.

Rear element: -1

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 7

Queue is empty.

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 8

Queue is not full.

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 9

Exiting program.

Process returned 0 (0x0) execution time : 37.251 s

Press any key to continue.

**OPERATION OF QUEUES USING LINKED LISTS:**

1)Enqueue:

Function enqueue(data):

Create a new Node with the given data

If rear is NULL (queue is empty):

Set both front and rear to the new Node

Else:

Set the next of the current rear Node to the new Node

Update rear to the new Node

Print "Inserted: data"

2)Dequeue:

Function dequeue():

If front is NULL (queue is empty):

Print "Queue is empty. Cannot delete."

Return

Store the data of the front Node in deletedItem

If front is equal to rear (last element):

Set both front and rear to NULL

Else:

Update front to the next Node

Print "Deleted: deletedItem"

3)Size:

Function size():

Initialize count to 0

Start from the front Node

While the current Node is not NULL:

Increment count

Move to the next Node

Return count

4)Display:

Function display():

If front is NULL (queue is empty):

Print "Queue is empty."

Return

Create a current Node and set it to front

Print "Queue elements:"

While the current Node is not NULL:

Print the data of the current Node

Move to the next Node

Print a newline

5)Front:

Function getFront():

If front is NULL (queue is empty):

Print "Queue is empty."

Return -1

Return the data of the front Node

6)Rear:

Function getRear():

If rear is NULL (queue is empty):

Print "Queue is empty."

Return -1

Return the data of the rear Node

7)Empty:

Function isEmpty():

Return true if front is NULL (queue is empty), else return false

**PROGRAM TO EXECUTE STACKS USING LINKEDLISTS:**

#include <stdio.h>

#include <stdlib.h>

// Node structure for the queue

struct Node {

int data;

struct Node\* next;

};

struct Node\* front = NULL;

struct Node\* rear = NULL;

// Function to insert an element into the queue

void enqueue(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

newNode->next = NULL;

if (rear == NULL) {

front = rear = newNode;

} else {

rear->next = newNode;

rear = newNode;

}

printf("Inserted: %d\n", data);

}

// Function to delete an element from the queue

void dequeue() {

if (front == NULL) {

printf("Queue is empty. Cannot delete.\n");

return;

}

struct Node\* temp = front;

front = front->next;

if (front == NULL) {

rear = NULL;

}

int deletedItem = temp->data;

free(temp);

printf("Deleted: %d\n", deletedItem);

}

// Function to get the size of the queue

int size() {

struct Node\* current = front;

int count = 0;

while (current != NULL) {

count++;

current = current->next;

}

return count;

}

// Function to display the elements of the queue

void display() {

struct Node\* current = front;

if (current == NULL) {

printf("Queue is empty.\n");

return;

}

printf("Queue elements: ");

while (current != NULL) {

printf("%d ", current->data);

current = current->next;

}

printf("\n");

}

// Function to get the front element of the queue

int getFront() {

if (front == NULL) {

printf("Queue is empty.\n");

return -1;

}

return front->data;

}

// Function to get the rear element of the queue

int getRear() {

if (rear == NULL) {

printf("Queue is empty.\n");

return -1;

}

return rear->data;

}

// Function to check if the queue is empty

int isEmpty() {

return (front == NULL);

}

// Function to check if the queue is full (not applicable for linked list-based queues)

int isFull() {

return 0; // Linked list-based queues can't be full

}

int main() {

int choice, data;

while (1) {

printf("\nQueue Operations:\n");

printf("1. INSERT\n");

printf("2. DELETE\n");

printf("3. SIZE\n");

printf("4. Display\n");

printf("5. FRONT\n");

printf("6. REAR\n");

printf("7. IsEmpty\n");

printf("8. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter data to insert: ");

scanf("%d", &data);

enqueue(data);

break;

case 2:

dequeue();

break;

case 3:

printf("Queue size: %d\n", size());

break;

case 4:

display();

break;

case 5:

printf("Front element: %d\n", getFront());

break;

case 6:

printf("Rear element: %d\n", getRear());

break;

case 7:

if (isEmpty()) {

printf("Queue is empty.\n");

} else {

printf("Queue is not empty.\n");

}

break;

case 8:

printf("Exiting program.\n");

return 0;

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

**OUTPUT:**

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. Exit

Enter your choice: 1

Enter data to insert: 29

Inserted: 29

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. Exit

Enter your choice: 2

Deleted: 29

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. Exit

Enter your choice: 3

Queue size: 0

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. Exit

Enter your choice: 4

Queue is empty.

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. Exit

Enter your choice: 5

Queue is empty.

Front element: -1

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. Exit

Enter your choice: 6

Queue is empty.

Rear element: -1

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. Exit

Enter your choice: 7

Queue is empty.

Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

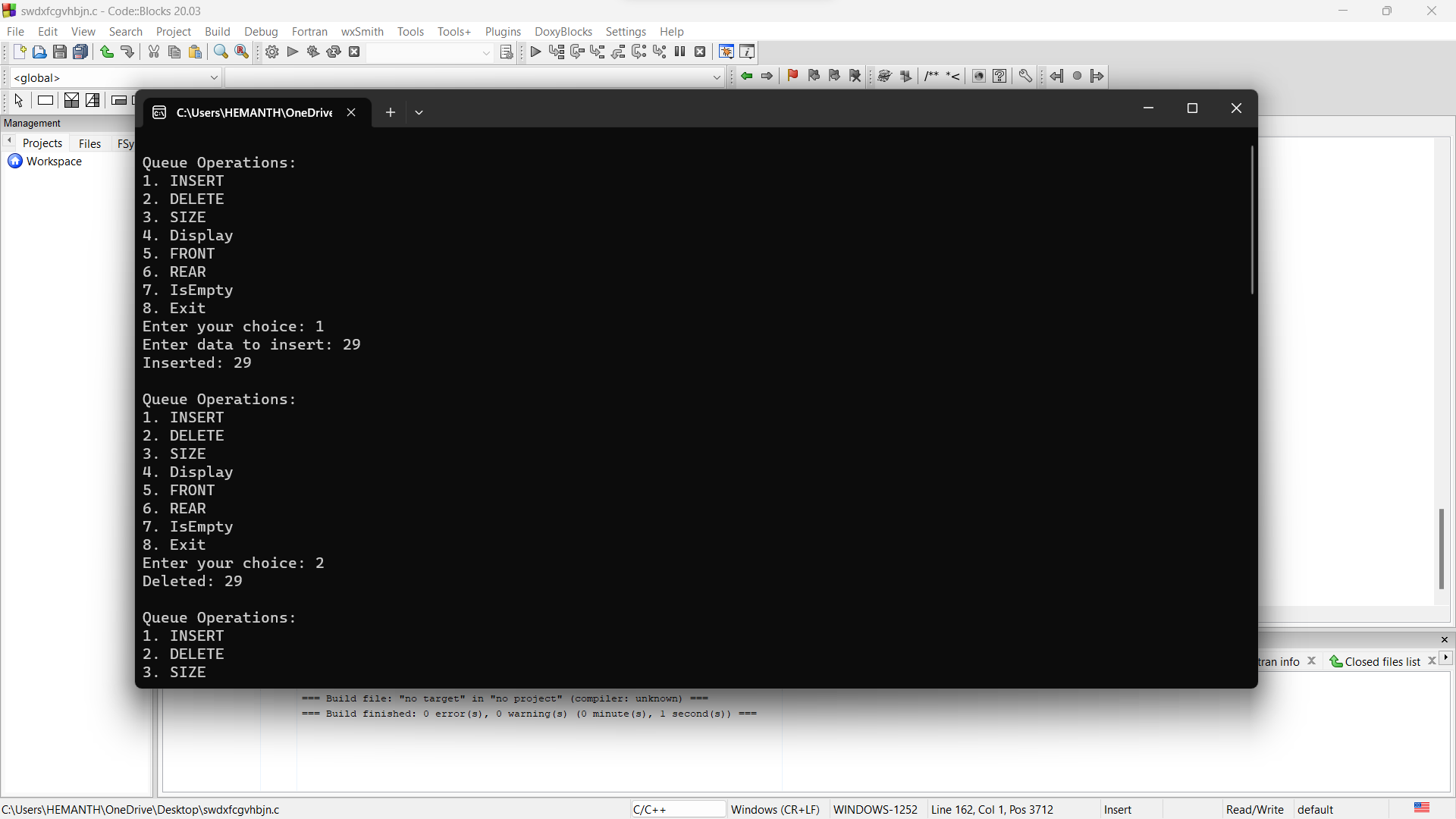
8. Exit

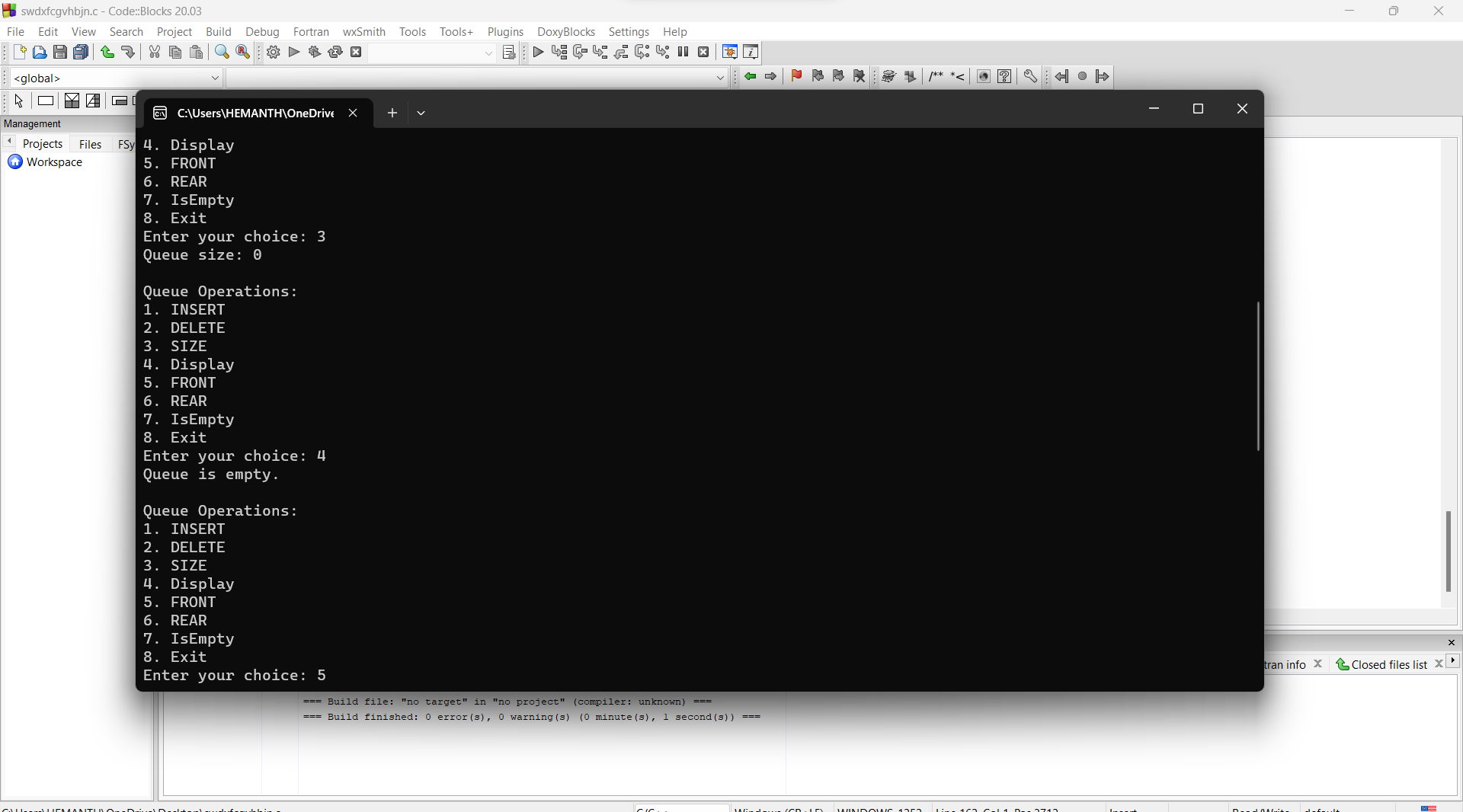
Enter your choice: 8

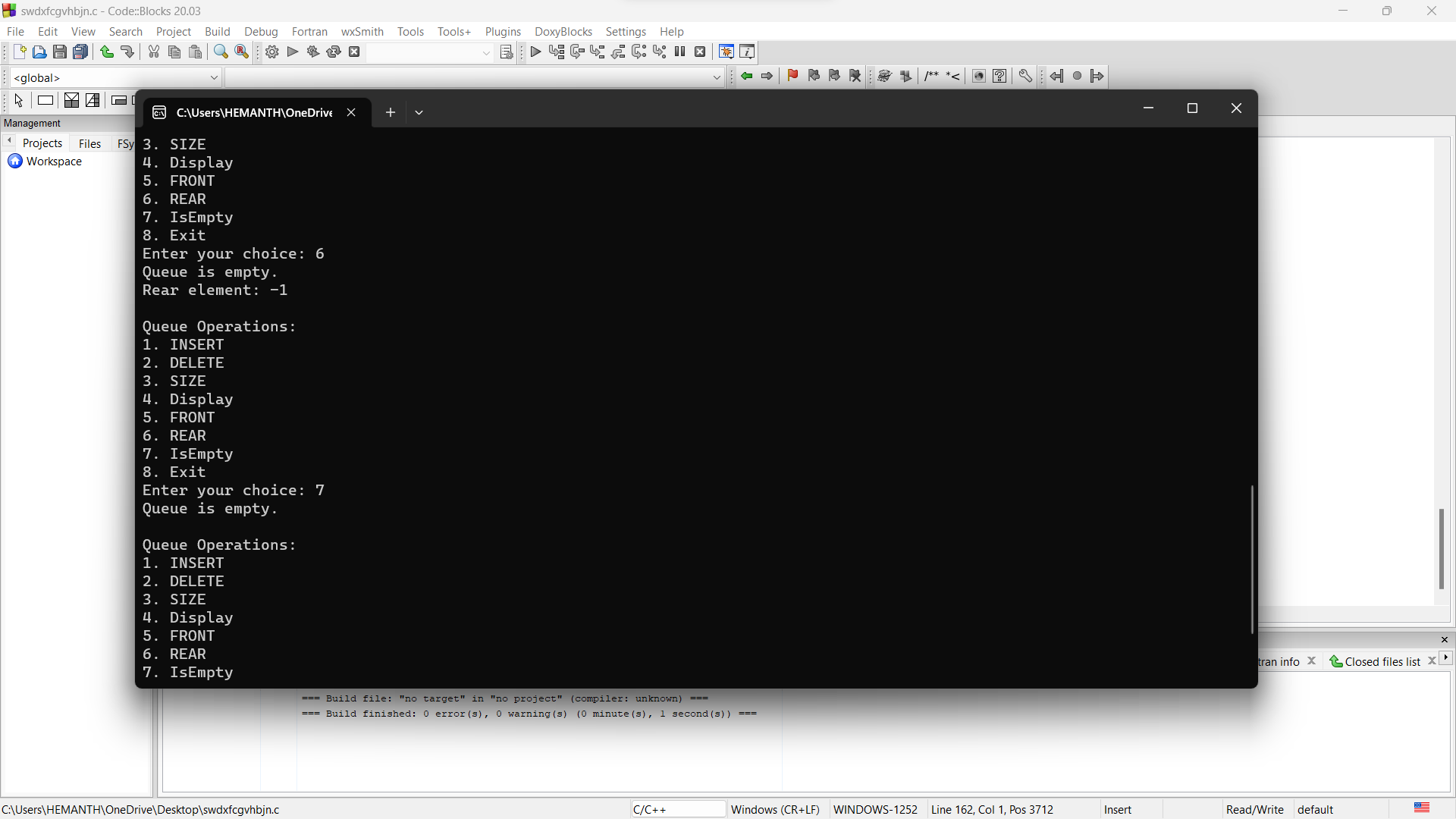
Exiting program.

Process returned 0 (0x0) execution time : 18.283 s

Press any key to continue.







**Operations on CIRCULAR QUEUES using Arrays:**

1.Enqueue:

Function enqueue(data):

a. If (front == 0 and rear == max\_size - 1) or (rear == (front - 1 + max\_size) % max\_size):

i. Print "Queue is full. Cannot insert."

ii. Return

b. If front == -1:

i. Set front and rear to 0

c. Increment rear (wrap around if needed using modulo)

d. Set queue[rear] to data

e. Print "Inserted: data"

2.Dequeue:

Function dequeue():

a. If front == -1:

i. Print "Queue is empty. Cannot delete."

ii. Return

b. Store queue[front] as deletedItem

c. If front == rear:

i. Set front and rear to -1

d. Else, increment front (wrap around if needed using modulo)

e. Print "Deleted: deletedItem"

3.Size:

Function size():

a. If front == -1:

i. Return 0

b. If front <= rear:

i. Return (rear - front + 1)

c. Return (max\_size - front + rear + 1)

4.Display:

Function display():

a. If front == -1:

i. Print "Queue is empty."

ii. Return

b. Print "Queue elements: "

c. If front <= rear:

i. Loop from i = front to rear (inclusive):

- Print queue[i] followed by a space

d. Else:

i. Loop from i = front to max\_size - 1:

- Print queue[i] followed by a space

ii. Loop from i = 0 to rear:

- Print queue[i] followed by a space

e. Print a newline

5.Front:

Function getFront():

a. If front == -1:

i. Print "Queue is empty."

ii. Return -1

b. Return queue[front]

6.Rear:

Function getRear():

a. If rear == -1:

i. Print "Queue is empty."

ii. Return -1

b. Return queue[rear]

7.Empty:

Function isEmpty():

a. If front == -1:

i. Return true

b. Else, return false

8. Full:

Function isFull():

a. If (rear + 1) % max\_size == front:

i. Return true

b. Else, return false

#include <stdio.h>

#include <stdlib.h>

int \*queue;

int front = -1;

int rear = -1;

int max\_size;

// Function to insert an element into the circular queue

void enqueue(int data) {

if ((front == 0 && rear == max\_size - 1) || (rear == (front - 1 + max\_size) % max\_size)) {

printf("Queue is full. Cannot insert.\n");

return;

}

if (front == -1) {

front = rear = 0;

} else {

rear = (rear + 1) % max\_size;

}

queue[rear] = data;

printf("Inserted: %d\n", data);

}

// Function to delete an element from the circular queue

void dequeue() {

if (front == -1) {

printf("Queue is empty. Cannot delete.\n");

return;

}

int deletedItem = queue[front];

if (front == rear) {

front = rear = -1;

} else {

front = (front + 1) % max\_size;

}

printf("Deleted: %d\n", deletedItem);

}

// Function to get the size of the circular queue

int size() {

if (front == -1) {

return 0;

}

if (front <= rear) {

return rear - front + 1;

}

return max\_size - front + rear + 1;

}

// Function to display the elements of the circular queue

void display() {

if (front == -1) {

printf("Queue is empty.\n");

return;

}

printf("Queue elements: ");

if (front <= rear) {

for (int i = front; i <= rear; i++) {

printf("%d ", queue[i]);

}

} else {

for (int i = front; i < max\_size; i++) {

printf("%d ", queue[i]);

}

for (int i = 0; i <= rear; i++) {

printf("%d ", queue[i]);

}

}

printf("\n");

}

// Function to get the front element of the circular queue

int getFront() {

if (front == -1) {

printf("Queue is empty.\n");

return -1;

}

return queue[front];

}

// Function to get the rear element of the circular queue

int getRear() {

if (rear == -1) {

printf("Queue is empty.\n");

return -1;

}

return queue[rear];

}

// Function to check if the circular queue is empty

int isEmpty() {

return (front == -1);

}

// Function to check if the circular queue is full

int isFull() {

return ((rear + 1) % max\_size == front);

}

int main() {

int choice, data;

printf("Enter the maximum size of the circular queue: ");

scanf("%d", &max\_size);

// Dynamically allocate memory for the circular queue

queue = (int \*)malloc(max\_size \* sizeof(int));

while (1) {

printf("\nCircular Queue Operations:\n");

printf("1. INSERT\n");

printf("2. DELETE\n");

printf("3. SIZE\n");

printf("4. Display\n");

printf("5. FRONT\n");

printf("6. REAR\n");

printf("7. IsEmpty\n");

printf("8. IsFull\n");

printf("9. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter data to insert: ");

scanf("%d", &data);

enqueue(data);

break;

case 2:

dequeue();

break;

case 3:

printf("Queue size: %d\n", size());

break;

case 4:

display();

break;

case 5:

printf("Front element: %d\n", getFront());

break;

case 6:

printf("Rear element: %d\n", getRear());

break;

case 7:

if (isEmpty()) {

printf("Queue is empty.\n");

} else {

printf("Queue is not empty.\n");

}

break;

case 8:

if (isFull()) {

printf("Queue is full.\n");

} else {

printf("Queue is not full.\n");

}

break;

case 9:

printf("Exiting program.\n");

free(queue); // Free the dynamically allocated memory

return 0;

default:

printf("Invalid choice. Please try again.\n");

}

}

return 0;

}

**OUTPUT:**

Enter the maximum size of the circular queue: 29

Circular Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 1

Enter data to insert: 29

Inserted: 29

Circular Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 1

Enter data to insert: 29

Inserted: 29

Circular Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 2

Deleted: 29

Circular Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 3

Queue size: 1

Circular Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 4

Queue elements: 29

Circular Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 5

Front element: 29

Circular Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 6

Rear element: 29

Circular Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 7

Queue is not empty.

Circular Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 8

Queue is not full.

Circular Queue Operations:

1. INSERT

2. DELETE

3. SIZE

4. Display

5. FRONT

6. REAR

7. IsEmpty

8. IsFull

9. Exit

Enter your choice: 9

Exiting program.

Process returned 0 (0x0) execution time : 26.655 s

Press any key to continue.

